Security and performance impact of client-side token storage methods

Gustav Fors
Abbas Radhi

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of bachelor's degree in software engineering. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Gustav Fors
E-mail: gufo19@student.bth.se

Abbas Radhi
E-mail: abra19@student.bth.se

University advisor:
Emil Folino
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Applications store more data than ever before, including sensitive information such as user data, credit card information, and company secrets. Due to the value of this data, malicious actors have a financial incentive to employ a variety of attacks against applications in order to gain access to it. As a consequence, application owners protect data behind authorization systems, with a common solution being token-based authentication systems in which the user's client receives and stores an access token after successful authentication. Developers seeking to create secure and effective applications face a number of questions. How do clients store these tokens and are they vulnerable to attack? What is the most secure way to store these tokens, and how do different storage methods impact the user experience?

The objective of this study is to answer these questions by comparing current storage methods available to developers of frontend applications. Literature was reviewed and an empirical study conducted so that comparisons could be made. Six storage options were found to be viable choices for review and ultimately it was concluded that In-memory storage with closures was the most secure storage option, but that this choice could have an impact on the usability of the application depending on the user desire for data persistence.

Keywords: storage, security, performance, tokens

Contents

Abstract **i**

1 Introduction **1**
1.1 On the content 1
1.1.1 Background 1
1.1.2 Purpose 3
1.1.3 Scope 3

2 Research Questions **5**
2.1 Defined research questions 5
2.1.1 RQ1: What storage options are available in modern frontend applications? 5
2.1.2 RQ2: Which vulnerabilities are associated with the options discovered in RQ1? 6
2.1.3 RQ3: How fast can data be retrieved from the options discovered in RQ1? 6
2.1.4 RQ4: Based on findings in RQ1, RQ2, RQ3 which storage option is most recommended for use in applications? 7

3 Research Method **8**
3.1 Method for answering the research questions 8
3.2 Preliminary study 9
3.3 Literature study 9
3.4 Empirical study 9

4 Literature Review **12**
4.1 Literature Overview 12
4.2 Client Storage Options 12
4.2.1 Cookies 12
4.2.2 Web Storage 14
4.2.3 Cache API 15
4.2.4 IndexedDB 15
4.2.5 WebSQL 17
4.2.6 In-memory 18
4.3 Security Concerns 19
4.3.1 Cross-Site Request Forgery 20
4.3.2 Cross-Site Scripting 20
4.3.3 Storage method specific vulnerabilities 21

4.3.4 Safest token storage method 22
4.4 Response times impact on user experience 22

5 Results and Analysis 24
5.1 Results 24
5.1.1 RQ1: What storage options are available in modern frontend applications? 24
5.1.2 RQ2: Which vulnerabilities are associated with the options discovered in RQ1? 25
5.1.3 RQ3: How fast can data be retrieved from the options discovered in RQ1? 25
5.1.4 RQ4: Based on findings in RQ1, RQ2, RQ3 which storage option is most recommended for use in applications? 26
5.2 Analysis 27
5.2.1 Storage options 27
5.2.2 Storage option vulnerabilities 27
5.2.3 Storage options user experience 30
5.2.4 Most recommended storage option 31

6 Conclusion 32
6.1 Summary of the study 32

7 Validity Threats 34

8 Future work 35

Chapter 1

Introduction

1.1 On the content

Token-based authentication is a widely-used technique for preventing unauthorized users from gaining access to specific system resources. The protocol enables users to verify their identity in exchange for a unique access token. While the token is valid, users can use it to access any resource for which it was issued, negating the need for the user to re-authenticate for each request. However, securely storing these tokens within a web client presents numerous challenges. This study compares various methods available for storing tokens in modern frontend applications and recommends the most secure and practical options based on the data findings.

1.1.1 Background

User authentication is perhaps the most critical security component of any web application. It is the security layer that guards against unauthorized application use and safeguards application data. There are many approaches to implementing authentication but with the increased popularity of microservices, mobile development and other decoupled system architectures, token-based authentication has emerged as one of the most widely used methods of identification [8].

Token-based authentication is described by Okta as a security protocol that enables users to identify themselves in exchange for a unique access token. Users can then access the website or application for which the token was issued without having to resubmit credentials each time they visit the web page, app, or other resource protected by that token [52]. This works by ensuring that each outgoing request from the resource contains the signed token which is then verified for authenticity by the server before a response is sent back [8].

A typical token-based authentication flow consists of a number of steps. A user attempts to read some data in a secure application. To authenticate, the client requests the user's credentials (typically a username and password). Once the user submits this data, the client sends it to the auth service which verifies the credentials and returns an access token to the client if they are valid. The client can now use the token it received from the auth service to request the desired data from the backend API. When the API receives the request, it will send the token to the auth service for validity verification and to confirm that the client is authorized to receive the

requested data. If this occurs, the API can now return the requested data to the client, which in turn is presented to the user.

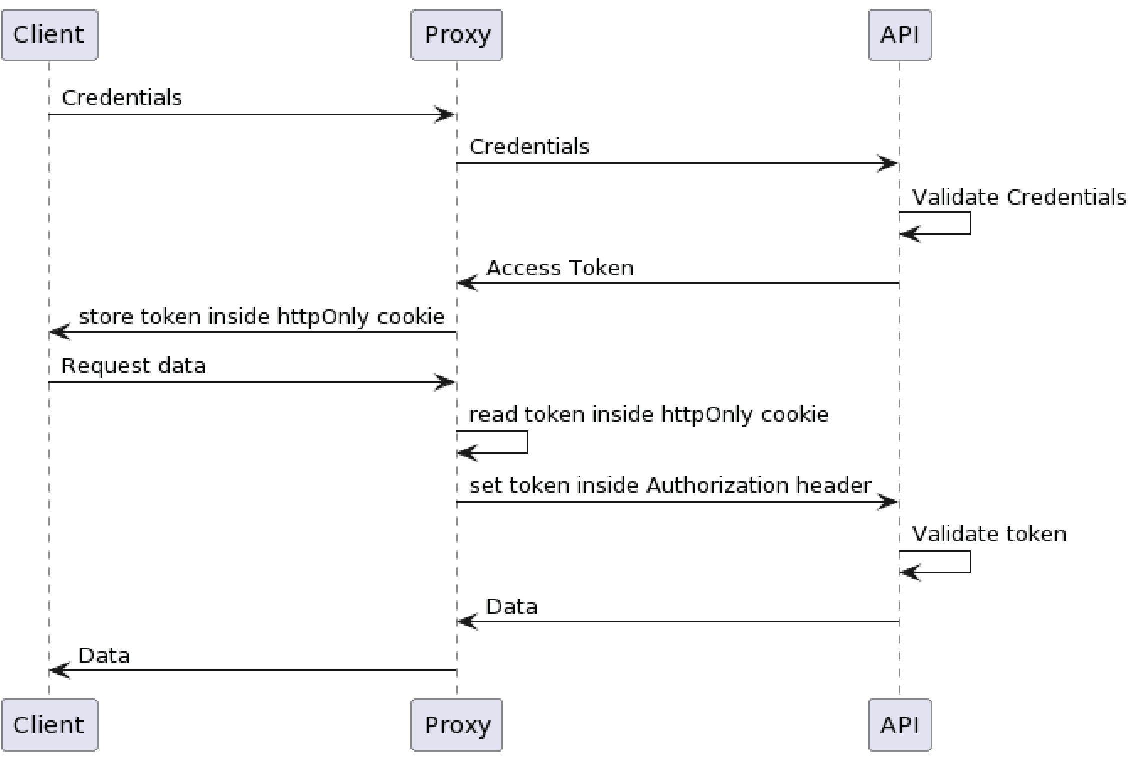


Figure 1.1: Token-based authentication flow.

There are numerous advantages to token-based authentication. As defined by Auth0 [8], tokens are completely stateless, which means that they contain all of the information required for authentication. This results in improved application scaling as the server is no longer required to store session state. Another advantage is the ability to decouple authentication/authorization logic into a single-responsibility auth service that can then be leveraged by other services rather than having to re-implement the same or similar logic multiple times. Additionally, access control can be included directly in the token payload, specifying user roles and permissions as well as the resources to which the user has access. This results in a reduction in server round trips and a performance improvement.

While token-based authentication has a number of advantages, it is not without drawbacks. A primary issue arises from the fact that tokens are completely stateless and as mentioned previously, all authentication information is encoded in the token. As a result, anyone who obtains access to the token can authenticate as the user for whom it was issued and carry out actions on their behalf. This raises serious security concerns, as malicious parties may attempt to retrieve other users' tokens and thus gain access to protected resources on the server. These attacks can have devastating consequences, as demonstrated by a report released by IBM Security which states that the global average cost of a data breach is $3.86 million due to reputational

damage, operational downtime, legal action and data loss [37]. Verizon stresses the critical nature of this issue, since the most common type of attack against web applications today is through stolen or brute-forced credentials such as access tokens and passwords [58]. Therefore, any application dealing with this type of authentication must ensure that token storage and transmission are done correctly and securely.

Unfortunately finding information and clear guidelines around how to securely store access tokens on a client can be difficult. Because tokens are simply a data string, they can be stored almost anywhere within an application. Those researching the subject will discover numerous contradictory recommendations for various storage options. Combined with the lack of a defined standard, this can make it challenging to select a storage option that is appropriate for the application. Common online recommendations are to use localStorage [32]. Other sources, such as Auth0's article on token storage [4] advise developers to store tokens in JavaScript variables. Some articles might advocate for storing tokens within cookies [31].

1.1.2 Purpose

The goal of this study is to fully clarify the topic of token-based authentication and then make recommendations based on an analysis of the currently available information on how to securely store tokens within frontend applications. The study will mainly benefit frontend developers and application owners, as they are more likely to work on or have a stake in projects that require this knowledge.

1.1.3 Scope

The study aims to compare token storage options found in RQ1 that meet two distinct criteria: having 90% user support and client API independence. These criteria have been established to exclude options that would be unsuitable for the vast majority of real-world applications or that are otherwise outside the scope of this study.

Browser support

Web development is a rapidly evolving industry, with new technologies released on a regular basis. One challenge with adopting new technologies is that they are frequently only supported by the latest web browsers, which means that a number of users will be unable to fully utilize the site as it was designed. There appears to be no publicly accepted consensus on the predefined browser compatibility percentage that developers should strive for; rather, the potential user base is subject to the decision of the site's owners who are responsible for choosing the technologies that will be used. For obvious reasons it is best practice to take steps to ensure that the largest proportion of users is able to access any site. For the purposes of this study, it was determined that the percentage of the global user base to support a technology being viable is 90%, implying that only ten out of every hundred users would be unable to use the application.

Client API Independence

Developers creating a client application may or may not have control over the API that interacts with the client. One example would be a client interacting with a third-party system developed by a completely different company who do not provide a means to make changes to their API. In contrast, the industry standard implementation for token-based authentication on the server side is to simply provide the client with a token which the client is then responsible for storing. This means that the API is unconcerned with how the token is stored; it is an entirely independent process occurring in the client. This study will only examine methods that are appropriate for this scenario in which the server issues a token that the client must store. Any method that requires the API and the client to share any kind of state will be excluded from consideration.

Chapter 2

Research Questions

2.1 Defined research questions

Four major questions must be investigated based on the scope of the study, these are RQ1 - RQ4.

2.1.1 RQ1: What storage options are available in modern frontend applications?

During the preliminary research, it was discovered that frontend applications can store data through a variety of storage options. It is necessary to identify the available storage options, their specification and usage for further evaluation. Determining these storage options allows the study to build a solid foundation and focus investigations for remaining research questions.

Expected outcome

Since the topic relates to frontend applications in which the user's web browser plays a significant role, the research expects to identify a number of options that can be viewed through the standard browser settings interface, typically targeted to developers. In the Chrome browser for example, these options would be visible in the Storage section of Developer Tools.

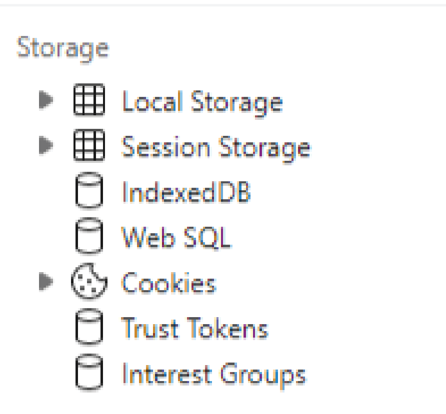


Figure 2.1: Google Chrome development tools storage interface.

In addition, other code based storage methods may be discovered during this research stage. Moreover, given that Web browsers can execute JavaScript code, it is reasonable to assume that variables are a valid storage option.

2.1.2 RQ2: Which vulnerabilities are associated with the options discovered in RQ1?

Given that the purpose of this study is to analyze the various storage options identified in RQ1 from a security perspective, the answer to this research question will have a substantial impact on the study's outcome. Although the study considers user experience and availability when determining RQ4, these factors are less important than security which is the primary consideration. Therefore, it is essential to identify the vulnerabilities associated with the various storage options in order to account for them when responding to RQ4.

Expected outcome

It is expected that there will be extensive examination of common and significant web-based attacks which developers can expect applications to be exposed to, especially for those storage options discovered while addressing RQ1. The study also expects to identify additional vulnerabilities that may be less well-known but still pose a significant security risk to the application.

2.1.3 RQ3: How fast can data be retrieved from the options discovered in RQ1?

A further consideration for storage option comparison is the speed which tokens can be retrieved from storage. Fast and responsive applications are essential in today's market; therefore, when selecting a storage option it may be beneficial to understand the performance impact of various storage methods in order to avoid unnecessarily throttling the application's performance. Depending on how responsive the application must be to end users, an option that provides a high level of security but takes a long time to extract tokens may be less advantageous than other options since it can have a heavy negative impact on user experience. This is a metric that developers should take into consideration when choosing a storage method.

Expected outcome

The study anticipates a significant difference in speed between the various storage options. Any option that stores data on disk should be considerably slower than options that store data directly in memory. It is also anticipated that more secure options will be slower than less secure ones, as it is reasonable to assume that their access mechanism is more complex, which could impact performance.

2.1.4 RQ4: Based on findings in RQ1, RQ2, RQ3 which storage option is most recommended for use in applications?

The study's objective is to provide developers with recommendations for implementing secure token storage within their application. To accomplish this, a comparison of the various storage methods will be conducted using the data collected in RQ1, RQ2, and RQ3 where the options with the best security, availability and speed results are shown.

Expected outcome

The results of RQ4 should be a clear indicator of the value of the differing storage options. These results will come from the research and analysis conducted in the study. The storage methods will be compared using the criteria defined in RQ1, RQ2 and RQ3, which are availability, security and speed performance.

Chapter 3

Research Method

3.1 Method for answering the research questions

The following section will present the methods used to conduct the literature study as well as the empirical study.

RQ1 is addressed by conducting a literature study. Within this study, published articles and/or documentation released by some major web browser vendors are reviewed with the intent of identifying the storage methods that exist in today's web browsers. Furthermore, each of the storage methods discovered are examined in conjunction with articles published by developers and documentation released by reliable sources. The goal is to understand the technical details of each method as well as the usage of each API responsible for accessing the data stored. This research question is critical to the study's outcome since it serves as the foundation for answering RQ2 and RQ3.

To answer RQ2, a literature study is conducted based on the results collected from RQ1. To address the known vulnerabilities of the storage methods, the study is conducted with the help of blog posts and articles written by developers and security experts. Also when available, reports are presented of known incidents of cyberattacks associated with these storage method vulnerabilities.

RQ3 is answered by performing an empirical study explained in Section 3.2.

RQ4 is answered by conducting a comparative analysis of RQ1, RQ2, and RQ3's collected results. The primary concern of this study is security, so this is the most important consideration when selecting candidates for RQ4. The second consideration is the effect on the user experience. If the most secure storage option has an unsatisfactory user experience, it should not be chosen as the recommendation. This means that the recommended option should be as secure as possible while still providing a satisfactory user experience. Measuring security involves examining each option's associated vulnerabilities to determine whether they can be exploited to gain direct access to the stored token or to perform malicious requests against the server that issued the token. The study also examines available protection methods for the various storage options; the option with the smallest attack surface is deemed the most secure.
Speed and persistence are also measures of the user experience. The study compares how long it takes to retrieve data from various storage options. This data is compared

to information gathered in a literature review on the impact of response times on user experience. The value of data persistence is significant, with persistent data favored over non-persistent options so that users do not have to constantly re-authenticate when using the application. The storage options in this study are compared in this metric through information collected in the literature review.

3.2 Preliminary study

The preliminary study was conducted by performing a keyword/search phrase search in a number of databases and search engines in order to find information on the topic. Search phrases used were: "secure token storage", "how to securely store tokens on a client", "best way of storing tokens".
Keywords used were: "token", "security", "client", "storage".
The databases/search engines searched were: Diva, Google Scholar, Google, Bing and DuckDuckGo.

3.3 Literature study

During the pre-study it was discovered that there is a strong lack of published research on the subject. As a result, this study will rely heavily on material published by developers including online articles, books, and user-generated content such as forum posts to find relevant literature.

To locate this literature, search engines such as Google, DuckDuckGo, and Bing will be used. Search engines will target relevant search phrases such as "client-side token storage", "JavaScript storage methods", "web browsers", "tokens", "web security", "Javascript attacks", "http", "https", "requests", "web attacks", "Javascript exploits" and "secure storage".

To be considered reliable, a source must be one of the leading non-profit organizations dedicated to documenting web standards, such as W3c [61], Mozilla [47], or OWASP Foundation [27], or another source whose claims can be verified by at least one additional source.

The results of the literature study can be found in Section 4.

3.4 Empirical study

The time it takes to extract tokens from storage using the various storage options discovered during RQ1 is measured in the empirical portion of the study.

The following setup is used to calculate the extraction time. To begin, a backend application built with Express JS, with two endpoints set up called "/token" and "/test" is created. When a GET request to the "/token" endpoint is received, the

backend application responds with a hard-coded token that can later be used in the "/test" endpoint request. The "/token" endpoint's sole purpose is to make it simple for test clients to obtain and store the token that is required as part of the "/test" endpoint request. To avoid any differences in data size for the token which could affect performance, the token is hard-coded rather than randomly generated. Because the token is hard-coded, the "/token" endpoint may appear unnecessary because the token could be coded directly into the client. However, this is not possible when dealing with certain types of cookies that cannot be set directly by the client, which is why this endpoint was implemented.

The actual measurement tests are carried out using the second endpoint, "/test." In the request it receives, the endpoint looks for an authorization header with a bearer token. The bearer's token value should be whatever the "/token" endpoint returned during the test's initialization phase. The endpoint will return a "200 OK" response if a valid token was provided.

During the preliminary study it was discovered that some storage options, such as HttpOnly cookies, are inaccessible by the client. For this reason, a backend application was used in the testing rather than directly measuring the time it takes the client to extract data from storage [26]. Because the experiment had to be structured in this way to account for all storage options, the result of this experiment is not only how long it takes to obtain the token from storage but also the time to attach it to a request, send it to the backend application and receive a response. Since this is the actual flow for any application implementing this sort of logic, the results are still applicable.

To carry out the tests, frontend applications were constructed for each storage method identified in RQ1. The frontend applications are both static websites as well as an application built in Next.js, a React framework used to enable server-side rendering [57]. This framework for testing was used to implement an API proxy since some methods of client-side storage cannot be set directly by the client and instead require a server in order to function as intended, e.g. HttpOnly cookies. As the focus of this study is scenarios in which the developer has access to an API with token-based authentication already implemented, an API proxy acts as a middleman between the two applications allowing for a HttpOnly cookie to be set (as explained in Section 5.2.2). Each client imports a JavaScript file that has an initialization and an execute function. The initialization function sends a request to the "/test" endpoint in order to receive a token that can later be used in the execute function. Once the token is returned it is stored on the client using their respective storage method and the initialization function is complete. The actual testing will then be carried out by the execute function. The function consists of a loop that sends requests using the fetch API to the "/test" endpoint and measures the time it takes from sending the request to receiving an OK response in milliseconds. The loop will repeat 1000 times for each storage method to obtain a more precise result. After each iteration the result is saved in an array of data. When the loop is finished, the sum of the numbers is divided by the total number of values in the array to get the average time in milliseconds for sending a request.

Back-end application	
cors	^2.8.5
express	^4.18.1

Table 3.1: Packages used for the back-end express.js testing application

Next.js application	
cookies	^0.8.0
http-proxy	^1.18.1
next	^12.1.6
react	^18.1.0
react-dom	^18.1.0

Table 3.2: Packages used for the Next.js application

The tests will be executed and performance measured on a total of four devices: two laptops and two mobile devices. The laptops host the testing application locally in order to minimize network latency. The mobile devices have access to the testing environments as they are hosted over the local network. To avoid network issues during the experiments, the same local network will be used throughout.

The test will be repeated for each client on the three major browsers: Chrome, Safari, and Edge [55]. The goal is to identify and account for any differences that may exist between different browser implementations. The sum of the averages from each browser is then divided by the number of browsers to get the overall average.

Device	**Laptop #1**	**Laptop #2**	**Mobile Device #1**	**Mobile Device #2**
Device name	Lenovo Thinkpad X1 Carbon	Lenovo Y50-70	iPhone XS Max	iPhone 11 Pro
Operating system	Windows 10 Version 20H2	Windows 10 Version 1909	iOS 15.4	iOS 15.2
CPU	Intel Core i7-7500U	Intel Core i7-4720HQ	A12	A13
Memory	16 GB LPDDR3-RAM 1866 Mhz	8 GB SO-DIMM DDR3 1600 Mhz	4 GB LPDDR4X	4 GB LPDDR4X
Disk ReadSpeed	3000 MB/s	750 MB/s	N/A	N/A
Disk Write Speed	1150 MB/s	500 MB/s	N/A	N/A
Browser Versions	Chrome 101 Safari 5.1.7 Edge 101	Chrome 98 Safari 5.1.7 Edge 101	Chrome 93 Safari 15 Edge 101	Chrome 101 Safari 15 Edge 101

Table 3.3: Hardware specifications for the laptop and mobile devices, as well as browser versions used during the experiments.

Chapter 4

Literature Review

4.1 Literature Overview

This literature review will provide an overview of the storage options available in web browsers using published material gathered from books, articles, blog posts and documentation. The primary focus of this section is to provide information about specifications, storage methods and known vulnerabilities of storage options which will form the basis of addressing RQ1 and RQ2.

4.2 Client Storage Options

Modern web browsers give websites multiple different storage options to choose from in order to save information in the user's browser with the purpose of extending the application's functionality and improving the user experience. In addition to storing data in the web browser, modern frontend applications can use server-side frameworks like NextJS, NuxtJS, and Angular to access additional features and storage options.

According to Mozilla, these storage options include Cookies, Web Storage (localStorage and sessionStorage), Cache API and IndexedDB [40]. In addition, Craig Buckler also refers to JavaScript Variables and WebSQL [11].

4.2.1 Cookies

A cookie is a small file containing information that is stored on the web browser. This cookie can be created using JavaScript on the client-side in a similar fashion to Web storage (Section 4.2.2). Alternatively, HTTP cookies can be generated and sent by the web server in the header of the HTTP response for the browser to store. Cookies have the ability to contain an expiration date and time and are often used to enhance the user experience by storing user states through sessions or user preferences. However, it can also be used to enable web tracking [21].

Although the storage capacity of a single cookie varies between browsers, it is recommended that the minimum storage capacity is set to 4096 bytes. As well as this, the recommended minimum number of cookies stored within a single domain is 50 [9]. They are domain specific, which by default makes them inaccessible from other domains. Permanent cookies persist after restart and are defined with an expiration

date. Session cookies on the other hand are defined without an expiration date and are erased when the session ends. A session is defined by the browser, although some browsers will allow session restoring, resulting in data stored indefinitely [45].

Settings can be adjusted to change the desired behavior of the cookie. Some of these options result in an increase in security and functionality. When creating a cookie, either set by JavaScript or by the server, a common vulnerability known as Cross-Site Scripting can be introduced (also known as XSS, explained in section 4.3.2). This is largely because the default behavior of JavaScript is to have read and write permissions on cookies. To mitigate the risk of such an attack, the *HttpOnly* flag is set. Assuming support by the web browser, cookies set by the web server that include the HttpOnly flag protect the cookie from being read or altered through client-side scripts [26]. HttpOnly cookies are available to the majority of web browsers currently in use, supported by 94.76% of the global user base [14].

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-8		2		3.1-4	10.1	3.2										
9-10	12-100	3-99	4-100	5-15.3	11.5-85	4-15.3		2.1-4.4.4	12-12.1				4-15.0			
11	101	100	101	15.4	86	15.4	all	101	64	101	100	12.12	16.0	10.4	7.12	2.5
		101-102	102-104	TP	87											

Figure 4.1: Web browser support for HttpOnly Cookies.

HTTP cookies are automatically sent by the browser on each subsequent request to the server. This behavior has a variety of uses and is often utilized in the context of user authorization. Unfortunately it can also lead to a class of attack known as Cross-Site Request Forgery, or CSRF [45]. HTTP cookies are automatically sent to the server regardless of their origin, often meaning that the server cannot differentiate between voluntary and involuntary user action. This can be problematic in some scenarios such as when an authenticated user is misled into performing unwanted actions (as explained in Section 4.3.1). To mitigate the risk of such an attack, the *SameSite* flag was first introduced to Google Chrome in 2016, followed by all other major browsers. The attribute asserts that a certain cookie is only sent when the request is initiated from the same domain that it is registered to. The SameSite attribute currently has reasonable browser support, being available for 93.47% of the global user base [16]. Another attribute that can be critical for protecting the contents of a cookie is the *secure* flag, which prevents the cookie from being sent over an unencrypted connection [19]. As cookies can be set by both the client and the server, two examples will be described.

On the client-side, cookies can be created, altered or removed with the help of the "document.cookie" property. For instance, to set a cookie with a name of "Joe" and a value of "Doe" simply assign the key and value as shown below. Giving the cookie another value is done in the same way as creating it [62].

```
document.cookie = "Joe=Doe";
```

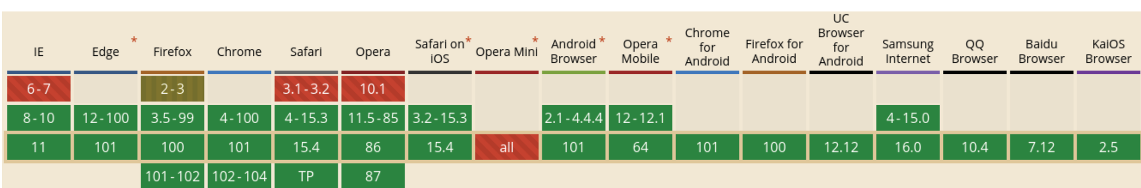
Listing 4.1: Creating a cookie on the client-side

On the server-side, cookies are set by including a "Set-Cookie" HTTP Header containing the content of the cookie. In the example below, a HTTP cookie is set with the name of "name" and "John" as value. An expiration date, HttpOnly, as well as the secure flag is set. Every part of the cookie is separated by semicolons. The HTTP response could look like this [28]:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=john; expires=Mon, 22-Jan-07 07:10:24 GMT; HttpOnly; secure;
Other-header: other-header-value
```

4.2.2 Web Storage

Web storage is a method of storing data on the web browser, similar to cookies. Web storage enables JavaScript to store data locally in the browser in the form of key/value pairs. There are two types of web storage: *sessionStorage* and *localStorage*. While the two are nearly identical and utilize the same API, the main difference between them is that localStorage is persistent whereas sessionStorage is not [25]. Data that is stored using the non persistent sessionStorage is only kept for the duration of an active session, also known as the browser or tab. This data is erased upon closing the active session. Information is not shared between tabs, even with the same domain. In contrast, data stored in localStorage is shared across tabs. Since localStorage is persistent, the information stored will not be erased when the browser or even operating system is closed as it does not have an expiration time [46]. The storage capacity of localStorage varies depending on the browser used, however most browsers allow at least five megabytes [33]. According to caniuse, Web storage is supported across a high number of browsers, with an estimated 97.09% user support [18].



IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
6-7		2-3		3.1-3.2	10.1											
8-10	12-100	3.5-99	4-100	4-15.3	11.5-85	3.2-15.3		2.1-4.4.4	12-12.1				4-15.0			
11	101	100	101	15.4	86	15.4	all	101	64	101	100	12.12	16.0	10.4	7.12	2.5
		101-102	102-104	TP	87											

Figure 4.2: Web browser support for Web Storage.

A code example by Mozilla shows how data is stored and retrieved from localStorage [39]. Data is set with the "setItem()" method that takes a key and value pair as arguments. The values must be strings.

```
localStorage.setItem('myCat', 'Tom');
```

The method "getItem()," which takes a key as an argument, is used to retrieve values. This is how to get the value set in the code line above.

```
const cat = localStorage.getItem('myCat');
```

Data can also be removed by passing the key to method "removeItem()".

```
localStorage.removeItem('myCat');
```

As localStorage and sessionStorage use the same API, the same methods can be called on the sessionStorage property.

4.2.3 Cache API

Cache API is a client-side storage option found in almost all common web browsers with 93.81% global user support [12].

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-15	2-38	4-39	3.1-10.1	10-26	3.2-10.3										
6-10	16-100	[1] 39-99	[2] 40-100	11-15.3	[3] 27-85	11-15.3		2.1-4.4.4	12-12.1				4-15.0			
11	101	[1] 100	[2] 101	15.4	[3] 86	15.4	all	[2] 101	[3] 64	[2] 101	100	12.12	16.0	10.4	7.12	2.5
		[1] 101-102	[2] 102-104	TP	[3] 87											

Figure 4.3: Web browser support for Cache API.

The primary function of the Cache API is storing network requests as well as responses; any resource that is addressable by URL can be cached. Having the ability to store responses is beneficial to increase performance and save resources, as it removes the need to ask the server for the same resources that it previously provided. The data is persistent and often is used for offline applications such as in Progressive Web Apps [30]. While the storage limit for Cache API varies between browsers, the maximum limit for most is between 50 megabytes to 33% of the available disk space [48]. Here is an example of how the Cache API might be utilized [36].

The Cache API is available through a global with "caches" property with JavaScript. A cache is opened through the "open()" method that takes a name as an argument, returning a Promise.

```
1 const cache = await caches.open('my-cache');
```

Listing 4.2: An example of opening a cache named "my-cache"

There are three methods for appending data to the cache: "add()", "addAll()" and "put()". For example, this is an example of caching a request to an external source:

```
1 cache.add('https://example.com/data.json');
```

Listing 4.3: Sending a request to an external source and caching the result

Retrieving a value from the cache is done through the "match()" method as shown below:

```
1 const response = await cache.match('https://example.com/data.json');
```

Listing 4.4: Retrieval of values that are saved in cache

4.2.4 IndexedDB

IndexedDB is a storage API found across a wide range of modern browsers and supported by 96.72% of the global user base [15].

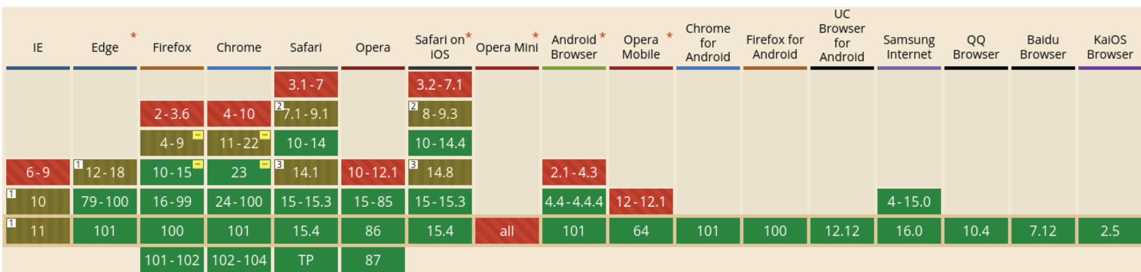


Figure 4.4: Web browser support for IndexedDB.

IndexedDB is created as an alternative storage method for storing large amounts of structured data on the client-side and also gives the ability to efficiently search. It allows for persistent storage of data in the form of key and value pairs, however unlike Web Storage it does not have a limited storage capacity, nor is it limited to only being able to store strings [23]. Instead it is possible to store any object supported by structured clone algorithms [42] including those of type Boolean, String, Date, File, Object and Array [43]. Transactions allow for interactions with the data stored in a database, including tasks like storing and retrieving data [60]. When a group of tasks require multiple alterations in the data, using a transaction gives the application some protection from failures that occur during the database interaction. In case of a failure during the tasks, transactions enable the database to rollback to a previous state, reverting all changes [5]. When a database is created it is only accessible within the domain or subdomain that it was created in. To demonstrate the usage of the API, we use the following example [3].

When opening a connection to a database, the "open()" method is called and the arguments passed are the database name and database version. The "onupgradeneeded" event handles initialization of object stores and indexes. IndexedDB uses collections of data known as object stores, which can be created with the function "createObjectStore()". Indexes are used to retrieve data and are created using the "createIndex()" method.

```
1  const request = indexedDB.open("CRM", 1);
2
3  request.onupgradeneeded = (event) => {
4    let db = event.target.result;
5    let store = db.createObjectStore("Contacts", {
6      autoIncrement: true,
7    });
8
9    let index = store.createIndex("email", "email", {
10     unique: true,
11   });
12 };
```

Listing 4.5: Opening a database connection and creating an object store and an index.

To insert data into object stores, a transaction is first created and the object store is obtained. The "put()" method is used to update or insert a new record into the

database. Lastly, the database connection is closed.

```
request.onsuccess = (event) => {
  const db = event.target.result;
  const txn = db.transaction("Contacts", "readwrite");
  const store = txn.objectStore("Contacts");

  store.put({email: "john.doe@example.com"});

  txn.oncomplete = function () {
    db.close();
  };
};
```

Listing 4.6: Inserting data into an object store.

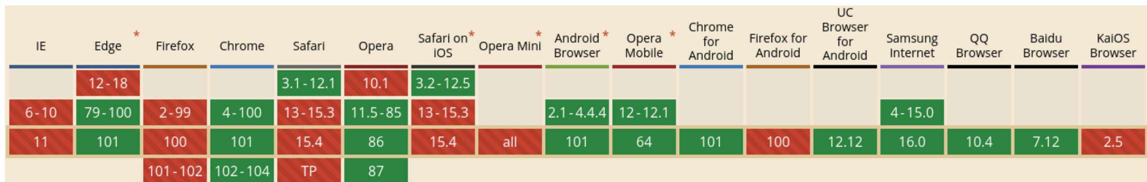
Transactions are used when retrieving data from a database. The object store is obtained once more, followed by the "get()" method, which queries a record by passing an identifier as a parameter.

```
request.onsuccess = (event) => {
  const db = event.target.result;
  const txn = db.transaction("Contacts", "readonly");
  const store = txn.objectStore("Contacts");

  let query = store.get(id);

  query.onsuccess = (event) => {
    if (event.target.result) {
      console.table(event.target.result);
    }
  };

  txn.oncomplete = function () {
    db.close();
  };
};
```

Listing 4.7: Retrieving data from an object store.

4.2.5 WebSQL

WebSQL is a client-side storage API supporting more complex relational databases and allows for the use of transactions and various types of queries, similar to those found in server-side databases [54]. Despite the fact that WebSQL has been abandoned in the favor of Web Storage and IndexedDB [59], with a substandard level of browser support of 75.86% global user base it remains implemented in some of the most recent web browsers [17].



IE	Edge	Firefox	Chrome	Safari	Opera	Safari on iOS	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-18			3.1-12.1	10.1	3.2-12.5										
6-10	79-100	2-99	4-100	13-15.3	11.5-85	13-15.3		2.1-4.4.4	12-12.1				4-15.0			
11	101	100	101	15.4	86	15.4	all	101	64	101	100	12.12	16.0	10.4	7.12	2.5
		101-102	102-104	TP	87											

Figure 4.5: Web browser support for WebSQL.

Data is persistent when using WebSQL which means that data is not deleted when the browser is restarted. WebSQL typically does not have a storage limit, however some browsers will ask the user for permission to increase storage capacity [49]. A code example demonstrates this [2].

The "openDatabase" function to create a database accepts a number of parameters, such as the database name, version number, description, database storage capacity, and a callback function. Queries are carried out within transactions. As shown below, a database is created and then a table is created within a transaction.

```
1 const db = openDatabase("db", "1.0", "example DB", 2 * 1024 * 1024);
2
3 db.transaction(function (tx) {
4   tx.executeSql("CREATE TABLE IF NOT EXISTS LOGS (id unique, log)");
5 });
```

Listing 4.8: Creating a database and and a table called LOGS in WebSQL.

In the case of a SELECT query to the database, a callback function is sent to handle the results.

```
1 db.transaction(function (tx) {
2   tx.executeSql(
3     "SELECT * FROM LOGS",
4     [],
5     function (tx, results) {
6       // handle ‘results‘
7     },
8     null
9   );
10 });
```

Listing 4.9: An example of retrieving data from the database with a SELECT query.

4.2.6 In-memory

98.13% of browsers in use today have JavaScript engines built-in [13].

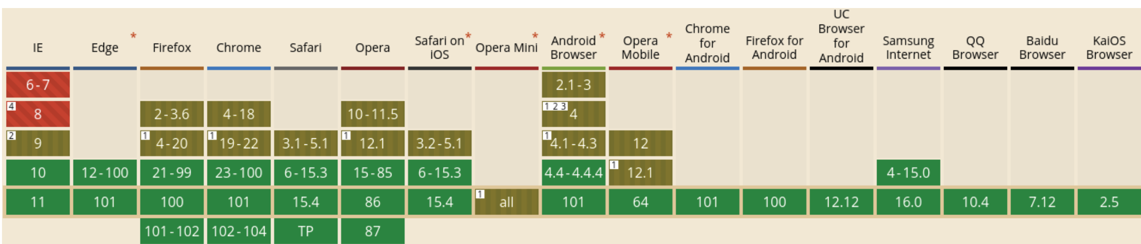


Figure 4.6: Web browser support for ES5.

These engines are software components that execute JavaScript code [6] and provide the ability for non-persistent in-memory data storage using variables. Variables are abstract storage locations with a symbolic name referencing a memory address that contains some amount of data called a value. Variables can store many different data types including integers and strings, with a memory capacity for the string datatype of $2^{53} - 1$ elements [7] [24].This makes variables with the string type a valid option for storing access tokens. In fact, storing access tokens in variables for single-page applications is recommended by Auth0 as long as closures are used [4]. In JavaScript, closures are created every time a function is created. A closure is a combination of a function bundled together with references to its surrounding state, also known as the lexical environment [41]. This means that variables defined inside a function are not accessible from outside the function scope, but variables defined outside a function are accessible from inside the function scope. This is important, because if access tokens are stored in the global scope (therefore outside of a function closure) it will be vulnerable to XSS attacks since the variable is accessible. Defining the variable holding the access token inside of a function closure makes it no longer accessible through XSS attacks since the variable is inaccessible from outside the scope.

```
1 <script>
2     const protectedVariable = () => {
3       let token = "secret";
4     };
5 </script>
```

Listing 4.10: An example of storing a token inside a closure.

Storing tokens in variables without closures offers no additional security benefits over options like Web Storage (localstorage/sessionstorage) but Web storage also has the added benefits of persistent storage, likely making it a better alternative.

4.3 Security Concerns

Two types of attacks are most commonly encountered by users of the storage methods detailed in this study. These are known as Cross-Site Request Forgery attacks (or CSRF attacks) and Cross-Site Scripting attacks (referred to as XSS attacks) [44].

4.3.1 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF for short) is an attack that takes advantage of the automatic inclusion of credentials along with HTTP requests, including HTTP cookies. It is especially dangerous when the request is made to a domain in which a user is already authenticated, as the receiving server is not able to differentiate between voluntary and involuntary requests from the user's client. Malicious requests often target functionality that may benefit the attacker or cause damage to the victim. A cross-site request forgery attack can be performed in a variety of ways. The primary objective of the attack is to trick the target or the target's browser into sending a request to a domain in which the user is authenticated [34]. Social engineering attacks can be used to perform cross-site request forgery attacks by misleading the victim into overlooking security threats and accidentally giving away sensitive information. One example of such a social engineering approach is to send an email or link that automatically includes sensitive credentials in the request by the browser when the target clicks it [29]. This method may be easily noticeable by the target due to the fact that the browser is redirecting them to another page. However, other more subtle techniques can be used that are harder to detect, employing the use of embedded images, scripts, forms or other methods that can make HTTP requests [20]. A hypothetical example of a banking cross-site request forgery attack through an embedded image is now described. Assume the victim is already authenticated in the hypothetical banking website "bank.com". The victim enters a website with an image that references a URL intended to make a malicious request.

```
<img src="https://bank.com/transfer?amount=5000&to_account=12345678" />
```

Upon entering the website, the browser sends a GET request to the URL referenced in the image, which results in the browser including valid credentials along with the request. If the banking website does not have CSRF protection implemented, the request is authorized since the receiving server cannot differentiate it from a legitimate request, allowing for a transfer of funds to another account. If the server were to accept POST requests instead, the image tag could be replaced with a fake form that sends a request to the target when submitted. The victim does not even need to submit the form themselves, it can be done automatically via JavaScript [20].

As this vulnerability mostly revolves around HTTP cookies, there are a few security settings implemented to mitigate the risk of such attacks. These settings are explained in Section 4.2.1.

4.3.2 Cross-Site Scripting

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. The end user's browser has no way to know that the script should not be trusted, and thus will execute the script. Because the browser thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser. The flaws that allow these attacks to succeed are pervasive and exist wherever a web application uses user input without validating or encoding

it within the output it generates [35]. As a result, access tokens stored in a manner that is accessible via JavaScript, such as variables without closures, Web Storage, cookies without the HttpOnly attribute set, Cache API, IndexedDB, and WebSQL, are vulnerable to this exploit, and malicious users will be able to extract tokens from other users using cross-site scripting.

Cross-Site Scripting attacks differ in complexity, but a simple illustration comes from Sebastian Peyrott in the JWT Handbook who describes an example of an XSS attack sourced from a public site's comment section. Each comment that is added is stored in the backend and also displayed to users in the comments section. If these comments are not sanitized by the backend, it is possible for an attacker to submit a comment that could be interpreted as a <script> tag by the user's browser. In this way, arbitrary Javascript code could be executed, allowing credentials that have been stored as cookies or in local storage to be stolen [53].

Protecting against cross-site scripting attacks requires validating all output, escaping and sanitizing potentially malicious code. This is known to offer perfect injection resistance. Any output that does not undergo this sanitization procedure is a potential vulnerability that should be addressed. Frameworks such as React and Angular can be utilized by developers to automate this procedure. However, these frameworks are not perfect as cross-site scripting vulnerabilities may still exist even when using them [1].

Another important factor to consider is thoroughly vetting any third-party libraries, as they may contain cross-site scripting vulnerabilities which could therefore compromise the application.

4.3.3 Storage method specific vulnerabilities

A common feature among the storage methods discussed is that almost all have a client-side storage API associated with them such as IndexedDB and Web Storage, allowing them to be written, modified and read through JavaScript.

As discussed in Section 4.3.2, cross-site scripting attacks allow an attacker to inject and execute malicious JavaScript code into the target's web browser, giving them almost full control over the client-side [44]. As many of the client-side storage alternatives in this study are used though their respective JavaScript API, most of them are susceptible to this vulnerability to some extent. These storage methods include Cookies, Web Storage, Cache API, IndexedDB, WebSQL and In-memory storage.

Client-side storage alternatives that are not controlled through a JavaScript API are normally not susceptible to cross-site scripting attacks. While In-memory storage is especially vulnerable to this type of attack, when implemented correctly a successful attack is difficult to perform. Although JavaScript variables are usually readable by very simple JavaScript code, when stored within scopes and closures they become invisible to the global scope, effectively making them unreachable [4]. HTTP cookies with the HttpOnly flag set cannot be read or written by JavaScript, thus lowering

the risk of a cross-site scripting attack. However, this does make them vulnerable to another type of attack called Cross-Site Request Forgery [45]. As HTTP cookies are automatically included in each request they can be targeted by tricking the victim or web browser into sending unwanted requests [38]. It is critical to be aware that the risk of a cross-site scripting attack on a HttpOnly cookie is not completely eliminated because HTTP requests can also be generated through JavaScript [64].

4.3.4 Safest token storage method

According to Auth0, the most secure storage method is In-memory storage, in the form of a JavaScript variable. When storing a token, it is recommended to use JavaScript closures so that they are not accessible in the global scope. It is also noted that In-memory storage does not provide persistent storage, meaning data is removed on page refreshes and is not shared between tabs [4]. A comparison of some storage methods was conducted by Bogdanov where it was concluded that tokens should be saved In-memory, due to local storage being prone to cross-site scripting attacks, and HttpOnly cookies being subject to cross-site request forgery attacks [10].

There are recommendations from online sources to use localStorage, such as Washburn Jr.'s article on the best way to store tokens due to the persistence [32]. Some developers argue against IndexedDB, claiming that it requires more code and does not perform as well because it is designed for larger amounts of data [56]. It is also susceptible to the same vulnerabilities as Web Storage [65].

A general consensus observed in reliable sources is in favor of cookies. An article written by Copes highlights the importance of using HttpOnly over other methods such as localStorage [22]. The author emphasises that localStorage is unsafe from a cross-site scripting attack in the event of a compromised third-party script and instead strongly suggests storage inside a HttpOnly cookie as it cannot be read by JavaScript. This is further acknowledged by Parecki and Waite who also suggest using HttpOnly cookies for interacting with an API as JavaScript cannot access a token that is stored inside of them [63]. Wirantono shares a similar observation, advising the use of HttpOnly over localStorage, adding that although they are susceptible to cross-site request forgery attacks, the vulnerability can be mitigated by using the sameSite flag [64].

4.4 Response times impact on user experience

An important consideration for evaluating the performance of a token is the speed that it can be retrieved from storage. Responsiveness is an essential factor influencing user perception of modern systems. As such, this topic was investigated and researched to determine the impact of different storage methods on the speed of token retrieval.

Studies on timing as part of the UI experience have reached conclusions that have been consistent for several decades and can thus be considered relevant to both stan-

dard programs and web-based applications [50]. Human perception allows for three primary time limits which users recognize and respond to in different ways.

0.1 second
At this time limit, users will feel that the system is responding directly to their actions and there is no further need for additional feedback. If the screen interface responds to input in a time longer than this, users could sense that the response is not due directly to their own actions but is the consequence of system processing and subsequent action.

1 second
At this time limit or less, users will notice the delay without a disruption to their train of thought which will still not require special feedback. This delay will be perceived as the system performing tasks in response to the user input, but it sacrifices the sensation of immediate feedback. For web pages this implies that the loading of a new page should take no longer than this time to prevent users feeling constrained [51]. If an operation takes longer than 1 second, then users may perceive the system as sluggish and this could have an impact on the usability of the application.

10 seconds
This time limit is close to the maximum that a user will maintain focus on a task waiting for it to complete. The delay is noticeable and may cause impatience as the user's flow is disrupted. Special feedback is necessary here to keep users informed of the progress of the operation and ideally there should be a way for users to cancel the task [50]. This is especially important if the amount of time taken to finish is variable, since users feel more comfortable having an idea of the length of time they may be waiting.

Chapter 5

Results and Analysis

5.1 Results

5.1.1 RQ1: What storage options are available in modern frontend applications?

The study found seven storage options currently available for use in modern frontend applications. These are Cookies, WebStorage (which includes both LocalStorage and SessionStorage), Cache API, IndexedDB, WebSQL and In-memory storage using JavaScript variables. In order for a storage option to be deemed viable for use in the context of this study it had to meet two criteria. Firstly it had to exceed 90% user support so as not to negatively impact on user accessibility. The second criteria is that the frontend application itself must be able to add and retrieve data from storage independently of any third party. Of the seven storage options discovered, six passed these criteria and were deemed viable for consideration in the study. The only option which was not able to pass the criteria is WebSQL with only 75.86% user support. It was also discovered that this technology has been abandoned so support will continue to decline in the future. All other viable storage options also passed the client independence criteria.

Storage option	Storage capacity	Browser support	Set by the client	Persistent	Vulnerability
Cookies	$\approx 4KB$	>94.76%	Yes	Yes	XSS
Cache API	50MB - 33%	93.81%	Yes	Yes	XSS
HttpOnly Cookies	$\approx 4KB$	94.76%	No	Yes	XSS & CSRF
IndexedDB	Unlimited	96.72%	Yes	Yes	XSS
localStorage	>5MB	97.09%	Yes	Yes	XSS
sessionStorage	>5MB	97.09%	Yes	No	XSS
WebSQL	Unlimited	75.86%	Yes	Yes	XSS
In-memory (JS ES5)	$2^{53} - 1$ elem.	98.13%	Yes	No	XSS

Table 5.1: Comparison of all storage options discussed

5.1.2 RQ2: Which vulnerabilities are associated with the options discovered in RQ1?

The results revealed two significant security flaws in the storage options discovered in RQ1. The first concern is Cross-Site Scripting attacks (XSS), which are a form of injection attack in which the perpetrator can exploit unsanitized user input to inject malicious code into the user's client, potentially granting them access to stored tokens and allowing them to send requests on the user's behalf. Cross-Site Request Forgery (CSRF) is the second type of attack, in which the user is tricked into sending unwanted requests on behalf of the attacker. The attack takes advantage of the fact that web browsers include all cookies used by a domain in every request sent to that domain. As a result, whenever a user is tricked into sending a request, the forged request appears legitimate to the web server, which executes the action even though it was not intended by the user. All storage options were discovered to be susceptible to Cross-Site Scripting attacks, while only cookies were vulnerable to Cross-Site Request Forgery attacks. In addition, it was discovered that two of the storage options (cookies and In-memory storage using variables) can be configured to provide superior protection against XSS attacks compared to the other storage options. For developers using In-memory storage, closures can be used to protect data. Whenever a variable is defined inside a closure, it is no longer accessible via XSS, which significantly reduces the risk of exposed tokens. Similarly, with cookies, the HttpOnly flag can be set to prevent JavaScript from accessing the cookie's value. However, since cookies are automatically included in requests, attackers would still be able to forge requests even if they lacked access to the actual token. This is not the case with In-memory storage utilizing closures.

5.1.3 RQ3: How fast can data be retrieved from the options discovered in RQ1?

The access speed of a storage method is of particular interest in this study. Because the discovered storage methods function differently and have different APIs, it was theorized that some options would have access speeds that could be significant to an application's overall performance. Cache API, Cookies, HttpOnly Cookies, IndexedDB, localStorage, sessionStorage, and In-memory storage were among the methods tested.

The tests were carried out on four different devices: two laptop computers and two mobile phones. The average access speeds for various storage options on various web browsers are presented in the tables below. Time results are measured in milliseconds.

In-depth overview of the results

Device	Storage Method	Chrome	Safari	Edge	Average
Lenovo X1 (Experiment 1)	Cookies	3.515	3.176	2.719	≈ 3.13
	Cache API	4.021	4.363	5.432	≈ 4.6
	HttpOnly Cookies	8.154	7.34	6.432	≈ 7.31
	IndexedDB	11.12	7.785	8.678	≈ 9.2
	localStorage	3.38	2.73	3.189	≈ 3.1
	sessionStorage	3.073	3.22	3.29	≈ 3.2
	In-memory	3.532	2.744	3.153	≈ 3.14
Lenovo Y50-70 (Experiment 2)	Cookies	5.37	4.49	5.923	≈ 5.26
	Cache API	5.739	5.4	6.27	≈ 5.8
	HttpOnly Cookies	11.38	11.046	9.303	≈ 10.58
	IndexedDB	8.878	9.47	8.719	≈ 9.02
	localStorage	5.424	5.132	5.94	≈ 5.5
	sessionStorage	5.395	4.912	5.41	≈ 5.24
	In-memory	4.654	5.24	5.412	≈ 5.1

Table 5.2: Performance tests on laptop computers demonstrate the speed of various storage mechanisms in different web browsers.

Device	Storage Method	Chrome	Safari	Edge	Average
iPhone XS Max (Experiment 3)	Cookies	8.543	7.932	8.231	≈ 8.24
	Cache API	8.6	7.63	8.434	≈ 8.22
	HttpOnly Cookies	14.213	16.361	17.4	≈ 15.6
	IndexedDB	16.48	18.434	15.18	≈ 16.7
	localStorage	8.732	8.346	8.9	≈ 8.66
	sessionStorage	8.345	8.397	9.012	≈ 8.58
	In-memory	7.27	8.443	7.376	≈ 7.7
iPhone 11 Pro (Experiment 4)	Cookies	8.543	8.157	8.94	≈ 8.55
	Cache API	7.55	7.4	8.123	≈ 7.69
	HttpOnly Cookies	15.471	16.361	16.1	≈ 15.98
	IndexedDB	16.45	17.08	17.535	≈ 17
	localStorage	6.922	6.74	7.24	≈ 6.97
	sessionStorage	6.8	6.54	7.347	≈ 6.9
	In-memory	7.089	6.57	6.34	≈ 6.67

Table 5.3: Performance studies on mobile show how different storage methods perform in different web browsers.

5.1.4 RQ4: Based on findings in RQ1, RQ2, RQ3 which storage option is most recommended for use in applications?

In-memory storage with closures is the most recommended storage method, according to the comparative analysis performed in Section 5.2.4. If correctly implemented, this method provides the highest level of security because it is immune to the Cross-Site Scripting and Cross-Site Request Forgery attacks described in RQ2. According to the

results of RQ3, it is also one of the quickest methods available. The only disadvantage of this method in comparison to others is that it does not provide persistent storage. This means that every time the user reloads the application, the token data is lost and the user must re-authenticate. This can have a significant impact on the user experience, so if this is a major concern when developing the application, it is advised to utilize cookies with the HttpOnly and SameSite flags set instead. This method is slightly less secure than In-memory storage with closures because it is susceptible to Cross-Site Scripting attacks, though to a lesser extent than the other options. Using any of the other options, an attacker could gain access to the actual token using Cross-Site Scripting. However, with HttpOnly cookies, an attacker can never obtain the actual token value, but can still send requests containing the token. This is because cookies are automatically sent along with the request. However, it is more secure than the other options and offers persistent storage so that users do not have to re-authenticate every time they reload the application.

5.2 Analysis

This chapter examines the theories found in the literature review as well as the empirical data collected in order to answer questions regarding the storage options for frontend applications, their associated availability, vulnerabilities and data access speed.

5.2.1 Storage options

During the literature review in Section 4.1 the first task was to identify which storage options are available in frontend applications for the purpose of storing access tokens. The review concluded that there are seven storage options available: Cookies, Web Storage, (LocalStorage, SessionStorage), Cache API, IndexedDB and In-memory storage using variables. Of these, six were deemed suitable for use based on the limitations set for this study - a storage option must have at least 90% user support and the application itself must be able to store the token independent of any third party. The only option that did not match the support criteria is WebSQL with 75.86% support. This leaves a number of choices to consider for vulnerability inspection and empirical study.

5.2.2 Storage option vulnerabilities

The literature review showed that there were two vulnerabilities associated with the the selected token storage options. These are Cross-Site Scripting attacks (referred to as XSS attacks) and Cross-Site Request Forgery attacks (or CSRF attacks). XSS attacks are a type of injection attack which can occur when the application is not properly sanitizing user generated output. Since this attack allows malicious users to execute JavaScript code in the victim's browser it means that any access token or other sensitive data accessible through JavaScript is compromised and can be stolen by the attacker. The other form of attack called Cross-Site Request Forgery leverages

the fact that web browsers automatically and invisibly send along any cookies used by a domain in any request sent to that domain. Attackers exploit this by trying to trick their target's browsers into sending unintended requests on behalf of the attacker. The forged request will appear legitimate to the web server and will thus perform the requested action. To mitigate the risk of CSRF attacks it is important to use the SameSite flag for cookies. This flag makes it so cookies are only sent when the request is initiated from the same domain that it is registered to.

Of the two attacks described above, Cross-Site Scripting attacks are the most threatening. Every storage method in this study can be the subject to this form of attack, and since successful attackers are able to execute JavaScript code directly on the client, it can be used to access the actual token directly. In contrast, Cross-Site Request Forgery attacks are only able to target cookies. In addition, CSRF attacks do not give the attacker direct access to the token but rather trick the user into making unintended requests which then automatically include the token.

If there is an exploit that allows the application to be attacked using XSS, five out of the seven original storage options would have no additional protections in place to safeguard data, which would allow attackers to directly access the tokens stored using JavaScript through the storage specific APIs. The two safer options even through a successful XSS attack are In-memory storage using closures as well as Cookies using the HttpOnly flag.

When working with In-memory storage, JavaScript is only able to directly reach variables defined within the global scope meaning that if you want to protect variables you must define them within a private scope commonly referred to as a closure. A closure is created whenever a JavaScript function is created. This means to protect a variable it must be defined inside a function. A common way of doing this is by creating an application closure which holds the application logic and variables. This allows the application to function while protecting from XSS attacks since they are not able to directly reach into the application scope.

The other more protective option is to use HttpOnly cookies, these are a special kind of cookie that is not at all readable by JavaScript making it impossible to access its value through XSS attacks. It is absolutely vital that the HttpOnly flag is set - without this it is just a regular cookie that can still be read by JavaScript, making it no safer than any other storage option. HttpOnly cookies do possess one drawback. Since they are not writable by JavaScript they actually need to be set by a server. A criteria for this study was client independence meaning the client should not depend on some special configuration on the API server as developers may not even be in control of the API. Instead, the study assumes that the API simply returns an access token as part of the JSON response body which is standard practice for token-based authentication systems.

This does not mean that HttpOnly cookies cannot be used in the context of this study. An additional server component can be added as part of the frontend application. This server component can act as a middleman between the client and

the API, ensuring that all communication between the two is routed through it. By creating this sort of server it is possible to configure it to store and read the token using HttpOnly cookies on the client.

Considering this approach in more detail, consider the following scenario: you have an API you want to interact with, but you have no control over how it is implemented. You will receive a token after successfully authenticating and you must store it on the client. If you want to store the token inside an HttpOnly cookie, one option is to set up a proxy server to route all the communication between the client and the API through. This means that instead of calling the API directly, the client calls the proxy server which then calls the API on the client's behalf. Similarly, the API does not respond directly to the client, but rather to the proxy, which then forwards it to the client.

Having the proxy in this key position where all traffic flows through it allows developers to implement robust security systems for their frontend applications. For the purposes of this study, it allows developers to store access tokens on the client using HttpOnly cookies.

It is now possible to walk through an authentication flow based on the scenario described above. In this scenario a client wants to use the API to get data. As a first step, the client sends an authentication request to the proxy, including their credentials. The request is then forwarded to the API, which verifies the credentials and returns a token. The response from the API is sent to the proxy because the API received the request from the proxy rather than the client. The proxy now has the access token, but instead of forwarding it to the client, the proxy, as a server, can store it directly on the client inside a HttpOnly cookie. This means that the client now has the token stored, but is unable to read it and as a result is completely protected from XSS attacks.

Now that the token has been safely stored on the client, the next step is to retrieve it from the client and send it to the API in the format it expects, such as a Bearer token in the Authorization header of the request. The client has no way of accessing this token, making it impossible for the client to directly make an API request. Instead, the client must make the request through the proxy, which can read the token from the cookie because it is the entity that sent the cookie containing the token. Once the token has been extracted, the proxy can create the correct request by appending the Authorization header to the outgoing request, which includes the token as a Bearer. Since this is the correct format which the API expects it will return the data to the proxy which can then forward it to the client.

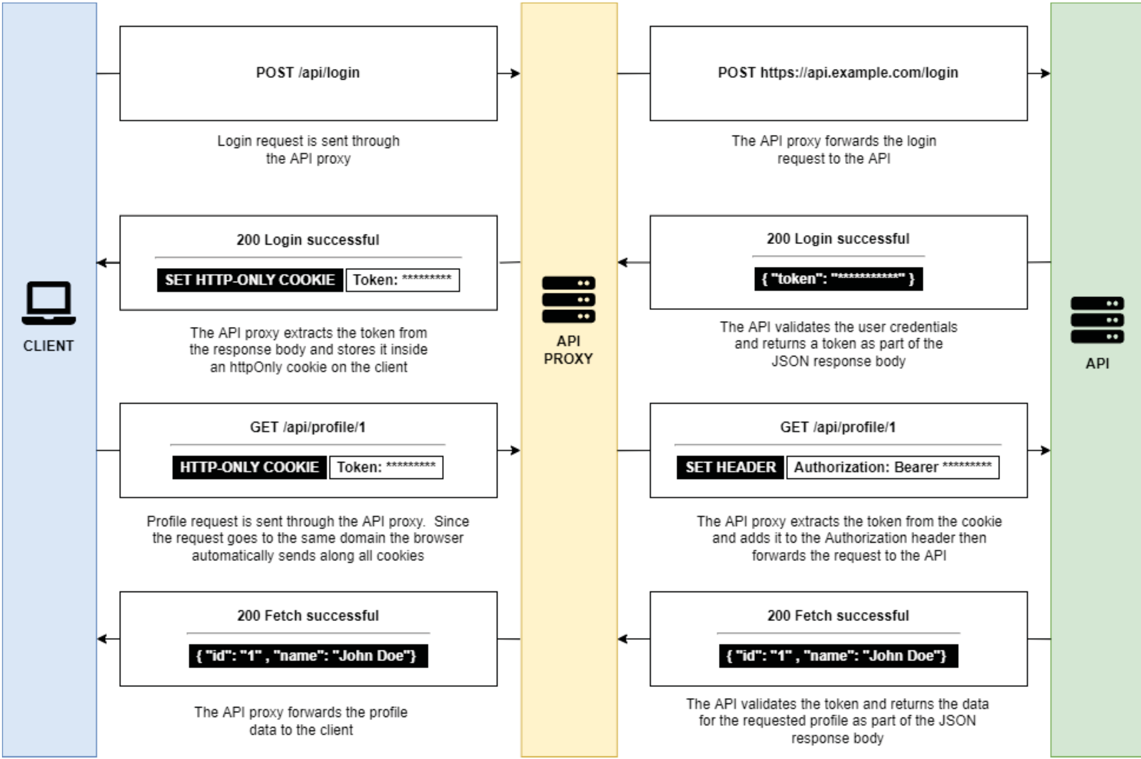


Figure 5.1: Authentication flow using HttpOnly cookies and API proxy.

Although both In-memory storage using closures and HttpOnly cookies are both safer ways of storing tokens than other alternatives, when contrasted against each other this study concluded that In-memory storage using closures is the better option in terms of security. Both options protect the token from being directly accessed via XSS attacks, however with HttpOnly cookies attackers would still be able to send malicious requests since cookies will automatically be sent along with the requests which then authorize it. This would not be the case with the In-memory storage option using closures, which means the attack surface is smaller and therefore making it a safer option.

5.2.3 Storage options user experience

Through both the literature review and also an empirical study, this study also examined the user experience using the different storage options. This was measured through speed and data persistence. The empirical study recorded how long it takes to withdraw a token from storage using the different options, attach it to a request, send it to the API and receive a response back. The literature review investigated how response times affect users of an application and whether the different response times measured in the empirical study would yield different results in terms of user experience.

During the empirical study the following average speeds out of all experiments were measured for the different storage options: Cache API 6.5ms, Cookie 6.3ms, In-

dexedDB 13ms, Localstorage 6.1ms, SessionStorage 6ms, In-memory storage 5.7ms, HttpOnly Cookie 12.4ms.

Looking at the results it can be seen that there is not a great variance in speed between the methods, although IndexedDB and HttpOnly Cookies are noticeably slower than the other options. This is reasonable since one is dealing with a fully fledged database system using transactions and the other is sending requests through a proxy. All storage options fall within the 0.1 second interval meaning that, based on the information from the literature review, users will feel that the system is responding directly to their actions and there will be no further need for additional feedback to the user. Although IndexedDB and HttpOnly cookies are slower than the other options, it is not enough to impact the users' experience. In fact, any of the storage options could be selected without having a negative impact on user experience.

One additional factor when dealing with user experience is data persistence. During the literature review it was discovered that Cache API, Cookies, IndexedDB and LocalStorage offer persistent storage meaning the user data will not be lost upon restarting the application as opposed to In-memory storage and SessionStorage where data would be lost. As a result, using the persistent options would offer a better user experience since the user would not have to re-authenticate when restarting the application.

5.2.4 Most recommended storage option

When considering the criteria of security, availability and user experience, the most recommended option for storing access tokens would be either HttpOnly Cookies or In-memory storage using closures. LocalStorage, SessionStorage, Cache API, WebSQL, Cookies and IndexedDB are all similar and offer no obvious advantages over the recommended choices. Furthermore, these others come with the considerable flaw of being very vulnerable to XSS attacks, thus making them clearly worse options. In-memory storage using closures and HttpOnly cookies are both better protected from XSS attacks, but In-memory storage using closures offers slightly better protection since, this option is not susceptible to CSRF attacks as HttpOnly cookies are. This comes at a cost however - In-memory storage using closures is not persistent, so data would be lost between sessions (unlike HttpOnly cookies). The conclusion of the analysis is that if user experience is the primary concern then HttpOnly cookies are the best available storage option. If the most important consideration is to make the application as safe as possible, then In-memory storage using closures would be the best choice.

Chapter 6

Conclusion

6.1 Summary of the study

It was determined that the viable storage options available in modern frontend applications are Cookies, Local storage, Session storage, In memory storage using JavaScript variables, Cache API and IndexedDB. All storage options are vulnerable to Cross-Site Scripting attacks (XSS) and cookies have an additional vulnerability to Cross-Site Request Forgery attacks (CSRF). The degree to which the options are exposed to the vulnerabilities differs and In-memory storage as well as Cookies can be configured in a way that offers superior protection against XSS attacks compared to the other options. Cookies can also be configured to be protected from CSRF attacks. To protect against XSS using In-memory storage, all variables must be defined inside a closure as this makes them inaccessible through XSS. Similarly, Cookies can also be protected from XSS by setting the flag HttpOnly, which makes it so that the cookie cannot be read by JavaScript. To protect cookies from CSRF attacks, another flag can be set called SameSite which blocks cookies from being sent when the request is sent from a different domain. These configuration settings make the two options superior in terms of security.

When considering user experience, it was found that there is no significant difference in the speed at which tokens can be retrieved from storage between the different options. In order for a user to be negatively impacted by the delay it had to exceed 100ms; empirical testing concluded that all storage options were consistently performing under 20ms. However another issue impacting user experience is data persistence. It was discovered that Cookies, Local storage, Cache API and IndexedDB offer persistent storage, while others do not. If data is not persistently stored, this may impact the user experience since reloading the application would force the user to re-authenticate .

When comparing In-memory storage using closures and HttpOnly cookies it is clear that the former is the more secure option since it is immune to XSS attacks. Meanwhile, HttpOnly cookies offer some protection, in that it is impossible to access the token directly using XSS. However, requests can still be forged through XSS which will automatically include the token in the request, even though it cannot directly be accessed. This is possible since cookies are automatically passed along with requests. Although this flaw exists, cookies are still more secure than the other options since they expose direct access to the token via XSS attacks. However, HttpOnly cookies

do offer persistent storage, meaning users would not have to re-authenticate when reloading the application, thus leading to a better user experience.

Although user experience is an important aspect to consider, security is regularly deemed more critical. Since the user experience of In-memory storage was not deemed significantly worse than other options, this study recommends it as the best storage option for securely storing tokens in frontend applications.

Chapter 7

Validity Threats

As the authors could not find other studies similar to the one conducted in this paper, they were not able to compare their results to other conclusions in order to note potential inconsistencies or inaccuracies revolving around the results.

While conducting the experiments, there may have been inconsistencies in the results due to running the tests over a network. The authors attempted to run the tests in environments that were as isolated as possible. This included closing all unnecessary tasks that are taking up resources, as well as running the browsers in private mode. However there is always the potential that variables that were not accounted for influenced the result outcome. These variables include other background processes taking up resources on the computers and unreliable network connections.

Chapter 8
Future work

This study focused on technologies for storing data currently implemented in modern web browsers. However, with the rapid pace of evolving technology, it would be valuable to maintain a view on new technologies and techniques to consider their appropriateness for storing sensitive data such as tokens. It would also be worthwhile delving deeper into any associated vulnerabilities as part of a greater evaluation of their advantages and disadvantages. One interesting topic to explore in a future study is whether it is possible to combine multiple storage methods in order to minimize the attack surface, making storage methods more difficult to exploit.

The current lack of a standardized storage method for sensitive data may need to be addressed in future research by which time this may have changed. It is also possible that newer attack methods may render current technologies like those discussed in this study less secure or even obsolete.

Future research could also expand on the empirical study, as potential exists for many different experiments to be conducted, such as testing on a wider range of storage methods, web browsers and devices. It may also be of interest to create a risk assessment matrix, where the risk is evaluated by comparing variables such as attack vectors of the storage methods. Input from security experts through conducting surveys and interviews would also be factored into this assessment. This will be beneficial in order to get a better understanding and a more accurate and fair evaluation of the storage methods discussed.

Bibliography

[1] "Cross site scripting prevention cheat sheet," last accessed 17 April 2022. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[2] "Html5 - web sql database," last accessed 2 May 2022. [Online]. Available: https://www.tutorialspoint.com/html5/html5_web_sql.html

[3] "Javascript indexeddb," last accessed 16 April 2022. [Online]. Available: https://www.javascripttutorial.net/web-apis/javascript-indexeddb/

[4] "Token storage," last accessed 22 April 2022. [Online]. Available: https://auth0.com/docs/secure/security-guidance/data-security/token-storage

[5] *Client-Side Data Storage*. O'Reilly Media, Inc, 2016.

[6] "Javascript engine," 2022, last accessed 19 April 2022. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript_engine

[7] "Variable (computer science)," 2022, last accessed 15 April 2022. [Online]. Available: https://en.wikipedia.org/wiki/JavaScript_engine

[8] Auth0, "Token based authentication made easy," n.d., last accessed 06 Mars 2022. [Online]. Available: https://auth0.com/learn/token-based-authentication-made-easy/

[9] A. Barth, "Http state management mechanism," 2011, last accessed 15 April 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6265

[10] A. Bogdanov, "The right way to store your json web token." 2021, last accessed 1 May 2022. [Online]. Available: https://motion-software.com/blog/store-your-json-web-token

[11] C. Buckler, "10 client-side storage options and when to use them," 2021, last accessed 8 May 2022. [Online]. Available: https://www.sitepoint.com/client-side-storage-options-comparison/

[12] caniuse, "Cache api," last accessed 10 March 2022. [Online]. Available: https://caniuse.com/mdn-api_cache

[13] ——, "Ecmascript 5," last accessed 4 April 2022. [Online]. Available: https://caniuse.com/es5

[14] ——, "headers http header: Set-cookie," last accessed 16 April 2022. [Online]. Available: https://caniuse.com/mdn-http_headers_set-cookie_httponly

[15] ——, "Indexeddb," last accessed 3 March 2022. [Online]. Available: https://caniuse.com/indexeddb

[16] ——, "'samesite' cookie attribute," last accessed 16 April 2022. [Online].

Available: https://caniuse.com/same-site-cookie-attribute

[17] ——, “Web sql database,” last accessed 17 April 2022. [Online]. Available: https://caniuse.com/sql-storage

[18] ——, “Web storage - name/value pairs,” last accessed 12 March 2022. [Online]. Available: https://caniuse.com/namevalue-storage

[19] M. Coates, “Secure cookie attribute,” last accessed 3 April 2022. [Online]. Available: https://owasp.org/www-community/controls/SecureCookieAttribute

[20] Codepath, “Cross site request forgery,” last accessed 19 April 2022. [Online]. Available: https://guides.codepath.com/websecurity/Cross-Site-Request-Forgery

[21] Cookies, “What are cookies? | cookies definition,” last accessed 16 April 2022. [Online]. Available: https://www.cloudflare.com/learning/privacy/what-are-cookies/

[22] F. Copes, “Jwt authentication: Best practices and when to use it,” 2021, last accessed 15 March 2022. [Online]. Available: https://blog.logrocket.com/jwt-authentication-best-practices

[23] A. Echamea, *Mastering Backbone.js*. Packt Publishing, 2016.

[24] ecma international, “Ecma-262,” 2020, last accessed 17 May 2022. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-262_11th_edition_june_2020.pdf

[25] C. Ford, “Introduction to localstorage and sessionstorage,” last accessed 12 March 2022. [Online]. Available: https://www.digitalocean.com/community/tutorials/js-introduction-localstorage-sessionstorage

[26] O. Foundation, “Httponly - set-cookie http response header,” last accessed 14 April 2022. [Online]. Available: https://owasp.org/www-community/HttpOnly

[27] ——, n.d., last accessed 06 Mars 2022. [Online]. Available: https://owasp.org/

[28] M. Frisbie, *Professional JavaScript for Web Developers*. Wiley, 2019, ch. 25.

[29] Impreva, “Cross site request forgery (csrf) attack,” last accessed 29 April 2022. [Online]. Available: https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/

[30] J. Irabor, “Working with the javascript cache api,” 2022, last accessed 15 March 2022. [Online]. Available: https://blog.logrocket.com/javascript-cache-api

[31] B. V. J, “Where should you store tokens?” 2021, last accessed 17 May 2022. [Online]. Available: https://balavishnuvj.com/blog/where-to-store-auth-tokens/

[32] M. W. Jr, “What’s the best way to store tokens in redux?” 2017, last accessed 17 May 2022. [Online]. Available: https://michaelwashburnjr.com/blog/best-way-to-store-tokens-redux

[33] I. Kantor, “Localstorage, sessionstorage,” 2022, last accessed 12 March 2022. [Online]. Available: https://javascript.info/localstorage

[34] KirstenS, “Cross site request forgery (csrf),” 2021, last accessed 17 April 2022. [Online]. Available: https://owasp.org/www-community/attacks/csrf

[35] ——, "Cross site scripting (xss)," 2021, last accessed 17 April 2022. [Online]. Available: https://owasp.org/www-community/attacks/xss/

[36] P. LePage, "The cache api: A quick guide," 2020, last accessed 8 March 2022. [Online]. Available: https://medium.com/@zaffarabbasmughal/browsers-support-and-limitation-towards-pwa-2e6c58cd16d5

[37] I. S. . P. I. LLC., "Cost of a data breach report 2020," accessed 07 March 2022. [Online]. Available: https://www.capita.com/sites/g/files/nginej291/files/2020-08/Ponemon-Global-Cost-of-Data-Breach-Study-2020.pdf

[38] Microsoft, "Asp.net cookies overview," 2014, last accessed 1 May 2022. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/ms178194(v=vs.140)

[39] Mozilla, "Window.localstorage," last accessed 15 March 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

[40] ——, "Client-side storage - learn web development," 2022, last accessed 8 May 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage

[41] ——, "Closures," 2022, last accessed 17 April 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

[42] ——, "Indexeddb api," 2022, last accessed 3 May 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[43] ——, "The structured clone algorithm," 2022, last accessed 3 May 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm

[44] ——, "Types of attacks - web security," 2022, last accessed 17 April 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks

[45] ——, "Using http cookies," 2022, last accessed 16 April 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[46] ——, "Web storage api," 2022, last accessed 2 May 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[47] ——, n.d., last accessed 06 Mars 2022. [Online]. Available: https://foundation.mozilla.org/en/

[48] Z. Mughal, "Browsers support and limitation towards pwa," 2019, last accessed 9 March 2022. [Online]. Available: https://medium.com/@zaffarabbasmughal/browsers-support-and-limitation-towards-pwa-2e6c58cd16d5

[49] nevf, "Chrome (webkit) websql database maximum file size?" 2010, last accessed 17 April 2022. [Online]. Available: https://stackoverflow.com/questions/4480879/chrome-webkit-websql-database-maximum-file-size

[50] J. Nielsen, "Response times: The 3 important limits," 1993, last accessed 14 May 2022. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/

[51] ——, "Powers of 10: Time scales in user experience," 2009, last

accessed 14 May 2022. [Online]. Available: https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/

[52] Okta, "What is token-based authentication?" n.d., last accessed 06 Mars 2022. [Online]. Available: https://www.okta.com/identity-101/what-is-token-based-authentication/

[53] S. E. Peyrott, *The JWT Handbook*. Auth0 Inc., 2018.

[54] J. Priest, "Chapter 16 - offline web applications," last accessed 8 March 2022. [Online]. Available: https://james-priest.github.io/100-days-of-code-log-r2/CH16-Offline1-WebSQL.html

[55] statcounter, "Desktop browser market share worldwide," 2022, last accessed 14 May 2022. [Online]. Available: https://gs.statcounter.com/browser-market-share/desktop/worldwide

[56] H. Sum, "Should i store jwt tokens in indexeddb?" 2020, last accessed 17 May 2022. [Online]. Available: https://stackoverflow.com/questions/62360200/should-i-store-jwt-tokens-in-indexeddb

[57] I. Vercel, "What is next.js?" last accessed 17 May 2022. [Online]. Available: https://nextjs.org/learn/foundations/about-nextjs/what-is-nextjs

[58] Verizon, "Data breach investigations report," 2020, last accessed07 March 2022. [Online]. Available: https://enterprise.verizon.com/resources/executivebriefs/2020-dbir-executive-brief.pdfx

[59] W3C, "Web sql database," 2010, last accessed 16 April 2022. [Online]. Available: https://www.w3.org/TR/webdatabase/

[60] ——, "Indexed database api 3.0," 2021, last accessed 8 March 2022. [Online]. Available: https://www.w3.org/TR/IndexedDB/

[61] ——, "W3c process document," n.d., last accessed 06 Mars 2022. [Online]. Available: https://www.w3.org/Consortium/Process/Process-19991111/background.html

[62] W3Schools, "Javascript cookies," last accessed 12 April 2022. [Online]. Available: https://www.w3schools.com/js/js_cookies.asp

[63] A. P. . D. Waite, "Oauth 2.0 for browser-based apps," 2020, last accessed 2 May 2022. [Online]. Available: https://datatracker.ietf.org/doc/pdf/draft-ietf-oauth-browser-based-apps-03.pdf

[64] M. Wirantono, "Localstorage vs. cookies: All you need to know about storing jwt tokens securely in the front-end," 2020, last accessed 2 May 2022. [Online]. Available: https://indepth.dev/posts/1382/localstorage-vs-cookies

[65] D. Woda, "Should i store jwt tokens in indexeddb?" 2021, last accessed 17 May 2022. [Online]. Available: https://community.auth0.com/t/should-i-store-jwt-tokens-in-indexeddb/63100/3

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden