

Implementation of a component to manage authorization for a web application

VICTOR PAULSEN

BACHELOR'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Implementation of a component to manage authorization for a web application

Bachelor's Thesis

by

Victor Paulsen

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden



LUND UNIVERSITY

Klarna.

2021

Abstract

This thesis was written in collaboration with Klarna in Giessen with the purpose of developing a component to deal with authentication in a web application. The component is intended to interact with the authorization service from Okta with the correct type of authorization flow to allow or disallow access to the web application's resources. At first, knowledge was gathered regarding the web application, the underlying code and its current safety measurements. Then, different types of authorization flows that Okta was supporting at the time were evaluated. The component was built with the programming language Java together with the framework Spring and consisted of three classes:

- one REST API with an endpoint to which Okta sends the required data in case of a successful login via Okta's portal.
- one class that manages all the logic for authorization.
- one helper class that translates the response from Okta's API into usable Java objects.

Authorization code flow was implemented in this solution together with logic for both access and refresh tokens (see 2.3).

Keywords

SSO, Auth, Spring Boot, Okta, Web Application, Security, OAuth2.0, OIDC, Authorization code flow, API.

Sammanfattning

Examensarbetet utfördes i samarbete med Klarna i Giessen med syftet att utveckla en komponent i en del av en webbapplikation som hanterar autentisering. Komponenten skulle interagera med auktoriseringstjänsten försedd från Okta tillsammans med korrekt auktoriseringsflöde som en lösning för tillåten användning av webbapplikationen. Arbetet inleddes med en undersökning av webbapplikationen, bakomliggande kod och dåvarande säkerhetsåtgärder. Därefter evaluerades olika typer av auktoriseringsflöden som Okta stödde. Komponenten blev byggd i programmeringsspråket Java tillsammans med ramverket Spring och bestod av tre klasser:

- ett REST API med en kommunikationspunkt som Okta överför essentiell information till efter en lyckad inloggning via Oktas portal.
- en klass som hanterar all logik angående auktorisering.
- en klass som översatte all respons från Okta's API till användbara objekt.

Flödet "Authorization code flow" var det som slutligen användes i komponenten tillsammans med logik för både access tokens och refresh tokens (se 2.3).

Nyckelord

SSO, Auth, Spring Boot, Okta, Webbapplikation, Säkerhet, OAuth2.0, OIDC, Authorization code flow, API.

Contents

Preface	8
1 Introduction	9
1.1 Background	9
1.2 Purpose	10
1.3 Goal formulation	10
1.4 Problem formulation	11
1.5 Motivation of thesis	11
1.6 Boundaries	12
2 Technical background	13
2.1 Spring Framework and Spring Boot	13
2.2 JSON web tokens	14
2.3 OAuth 2.0	15
2.4 OpenID Connect	16
2.5 Okta	16
2.6 Bitbucket	16
3 Method	17
3.1 Eliciting	17
3.2 Research phase	18
3.3 Implementation phase	18
3.4 Feedback phase	18
3.5 Validation phase	19
3.6 Communication	19

3.7 Changes during the project	19
3.8 Source criticism	19
4 Analysis	23
4.1 Elicitation results	23
4.2 Former authentication handling	25
4.3 Choice of authorization flow	26
4.4 Encountered problems and solutions	27
5 Results	29
5.1 Starting the authorization code flow	29
5.2 Acquiring tokens with the authorization code	30
5.3 Handling the authorization from the client	31
5.4 Logout	32
5.5 Design	33
6 Summary	35
6.1 How does the current solution for authentication and authorization look for the web application?	35
6.2 What are the requirements in terms of security related to an Okta single sign-on solution?	35
6.3 How is a user-friendly authentication solution implemented?	36
6.4 What is one way to implement single sign-on via Okta for a web application?	36
6.5 Reflection on ethical aspects	37
6.6 Future development opportunities	37
7 Terminology	39
8 References	41
9 Appendices	43

Appendix A. Screenshots of the login flow	43
Appendix B. JWT	45

Preface

This Bachelor's thesis would not exist without the support and guidance of my supervisor Bertil Lindvall, my examiner Erik Larsson, and my contact persons at Klarna - Marc Siewert among other colleagues. You all have my deepest gratitude.

Victor Paulsen

1 Introduction

This chapter introduces the company for which the project was performed, the purpose of the work, the goal as well as the motivation behind it. Lastly, the boundaries set for this project are discussed.

1.1 Background

Klarna is a Swedish company within the financial technology sector. They provide online financial products and services for the secure payment of physical and digital goods on the Internet. Klarna, initially named Kreditor and changed to Klarna in 2009, was founded in 2005 by Sebastian Siemiatkowski, Niklas Adalberth and Victor Jacobsson. The company currently has more than 4,000 employees in seventeen countries, most of them located in Europe. This current project was carried out in the office at Gießen, Germany and officially performed with Sofort GmbH, which was acquired by Klarna in 2014 and is responsible for engineering solutions.

Internally, a department team within Sofort GmbH was using a company-developed web application that allowed for categorizing transactions. This application was only intended for the members of the team. The application provided necessary data for another application that gave customers an overview of what they spent their money on and in that way tried to promote economically wise decisions. The application was in an early developmental stage and had room for improvements in different areas. One of the improvements of interest was to integrate the application with Okta so that it would use the employees' central set of credentials and thus be more user friendly, not having to remember several different credentials; a long-term goal was to use the same authentication methods for most of the company's web applications. The planned users of the web application were to be the internal members of a certain department team.

The application's backend was built with the help of Spring Framework, while the frontend was built with the framework React.

The component to be created would need to comply with these techniques.

This report presents the implementation of gaining authorization via Okta with their OIDC and OAuth2.0 API in a Spring Boot application.

1.2 Purpose

The purpose of this project is to upgrade and document the web application's features regarding authentication and authorization to more easily implement other web applications in the future. Klarna's requirements in terms of security, ease of use and flexibility are taken into consideration.

The expected result is the ability to log in with a central set of credentials that an employee at Klarna has, and once they are authenticated, they are not disrupted or redirected for reauthorization for a longer time.

1.3 Goal formulation

The goal of this thesis work is to enable single sign-on (SSO) authorization via Okta for a web application that currently performs authorization via stored credentials in the application's backend. A minimal viable product (MVP) is constructed as a component for the web application; it handles the login based on the central set of credentials currently stored for the employees.

The component's functionality is meant to provide resource access to a successfully authenticated user when the application is assigned to them. Additionally, it is designed to deny resource access if the user is either not authenticated or does not have the application assigned to them. Furthermore, the component keeps an active user logged in without having to redirect them to reauthorize or revalidate their issued tokens.

1.4 Problem formulation

The following questions are answered:

1. What is the current solution for authentication and authorization for the web application?
2. What are the requirements in terms of security related to a SSO solution?
3. How is a user-friendly authentication solution implemented?
4. What is one way to implement SSO via Okta for a web application based on Spring Boot?

1.5 Motivation of thesis

I chose this thesis topic because it is an interesting area to me as it mostly regards security and web development. The thesis implements a practical solution with underlying theoretical research. What also motivates me is the fact that the results could provide a potential guideline for developers who want to implement a similar solution. Furthermore, it could encourage the general audience and the information technology branch to make use of more secure and practical ways of dealing with authentication and authorization.

The company's goal is to have a standardized central login method for most of their applications to increase the ease of monitoring and useability, one that is less error prone due to not having to process many login credentials. Furthermore, having a central set of credentials would also facilitate their process of access reviews. Access reviews carries out with inspecting access assignments for applications and deactivating it for users who would not have a reason to be able to use them anymore. For example, when an employee would leave a company or moved to a different internal team. When employees of Klarna log in via Okta, multi-factor authentication is used. It is a company policy because it is safer from a security point of view and this pushes for integrating all their web applications with Okta.

1.6 Boundaries

The solution is only to be tried out for a web application based on Spring Boot. Spring's Security framework was not acceptable as it would create issues for the web application. This required the implementation of the authorization flow to make use of Okta's OIDC and OAuth 2.0 API instead. The component's functionality only needed to be validated for the web browser Google Chrome, because of Google Chrome being Klarna's internal web browser choice.

A more thorough validation to test the security of the component were to be made on Klarna's end, this thesis covers following validation.

1. Assigned user can log in.
2. Unassigned user can't log in.
3. Security flags of cookies are set.
4. Tokens are invalid and rotated after a set time.

2 Technical background

The technical background includes all techniques used in developing the component for the targeted web application.

2.1 Spring Framework and Spring Boot

Spring Framework: Spring Framework is a tool for creating and structuring Java applications by building a structure with common patterns. Framework helps with various functionalities, two of them being major points in the context of this thesis project:

Application context and dependency injection:

- Spring initializes and manages object instances and connects them together.
- Spring injects references into objects and ensures that every object has references to all the objects they require. It only creates one instance of classes with business logic methods, while it creates one or many instances of objects that hold data, depending on the need [2].

Spring MVC:

- This is used to create an API endpoint that Okta's servers can communicate with through HTTP GET requests [3]. Picture 1 shows how the endpoint looks in code, where the PostMapping annotation sets the endpoint-URI to `www.baseURI.com/form`.

```

@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
                                   @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}

```

Picture 1. Example of creating an API endpoint with Spring's MVC Framework.

Source: Screenshot taken from [3]

Spring Boot: This provides features to create production-grade Spring-based applications that are set up to run without needing much configuration. Spring Boot can embed Tomcat, Jetty or Undertow without needing to deploy WAR files [4]. It is used in the web application's backend, embedded with Tomcat.

2.2 JSON web tokens

A Java Script Object Notation (JSON) web token (JWT) is used for sharing data between two points, including ways to ensure confidentiality. These tokens can be signed and encoded with various algorithms. A JWT contains three parts:

- **Header:** Information about the algorithm and type of token being used.
- **Payload:** Contains data in the form of name-value pairs.
- **Signature:** Validates the token in a secure way.

An example of how a JWT looks is presented in Appendix B.

2.3 OAuth 2.0

OAuth 2.0 is an open-standard authorization protocol issued by the Internet Engineering Task Force (IETF) that allows third-party access to limited resources in one of two ways:

- On behalf of a resource owner by setting up an approval interaction between the resource owner and the HTTP service.
- By allowing the third-party application to obtain access on its own behalf.

In the early ages of client-server authorization models, it was normal for the client to store credentials from a resource owner to access resources from a server in their name. This can be illustrated through the following scenario: “Give me the key to your house, and I promise I will only go in and get your jacket.”; Where the person asking is the client, the key is the resource owners’ credentials, and the house is the resource server. This was a non-secure solution that needed improvement, and that is what gave life to the OAuth2.0 protocol in which it does not store any resource owner credentials on the clients’ side. Instead, it trades the following information:

- What access the client will have on behalf of the resource owner.
- A time limit on how long the access will be available.
- Other access attributes such as name, issuer and similar values.

The client uses this information, contained in what is called an access token, to access the protected resources on the resource server. The access token is issued by an authorization server when the resource owner approves the access the client will receive. This type of solution can be compared to a valet key for a car. When one gives a valet key to a car park attendant, the attendant cannot access the glove compartment or any other areas that might store valuables; they

are restricted to resources that enable them to do their job and nothing more [6].

Okta acts as the authorization server in this thesis work, issuing and validating all the tokens. Okta implements the OAuth2 protocol, and the format of their tokens are of JWT.

2.4 OpenID Connect

Since the OAuth2.0 protocol does not handle authentication, it is instead extended with the OIDC protocol, which enables applications to verify the identity of a user as well as to obtain basic profile information about the user [7].

2.5 Okta

Okta is an identity and access management company based in San Francisco. They provide cloud-based software that can be integrated with applications for the purpose of managing and authenticating users [14]. Their most relevant product regarding this thesis is SSO, which enables users to log in to a variety of applications with the same set of credentials. Important to understand is that OpenID providers need to have a certificate. Okta had a certificate during the time of this thesis project and acted as an authorization server.

2.6 Bitbucket

Bitbucket is a web-based storage service for coding projects that allows users to do basic Git operations while controlling read and right access to the code.

3 Method

This chapter describes how this thesis work was planned and executed. It is divided into several phases with a describing subchapter for each of them. This was done so that phases could overlap each other or continue during the entire project.

3.1 Eliciting

As with most projects, requirements had to be determined. The eliciting phase was used to determine the identity platform with which to integrate the component as well as other possible requirements that could follow. It is important to understand that this was an in-house project.

Firstly, requirements were elicited through brainstorming with the team that created the application. The question addressed was “How would you want your application’s authentication experience to be?” Requirements obtained from this brainstorming session were mostly regarding user experience, but there were also some implementation requirements mentioned. Brainstorming is one of several eliciting techniques advocated by Soren Lauesen [9].

Once the requirements were noted, a Jira [18] ticket was filled out and sent to an internal team for access management. (Jira Service Management is a software that was used as a virtual service desk). The ticket contained the following information:

- Type of application.
- In what environment the SSO integration was to be created in, with the options non-production or production.
- Whether OIDC or SAML would be used for authentication. SAML is another open standard protocol that can be used to deal with authentication and authorization. SAML uses different technologies but can achieve similar results [19].

The response to this form included Okta attributes to use in setup as well as some other requirements.

3.2 Research phase

Following the identification of requirements, a research phase was initiated. Reading documentation and guides provided by Okta helped to gain knowledge about the matter. Very basic applications with various methods of implementation were created to gain knowledge regarding how they turned out in practice. Furthermore, there were important resources in the official documentation of OAuth2.0 provided by IETF [6].

3.3 Implementation phase

Once the research reached a stage where sufficient knowledge and the necessary Okta credentials for the application were achieved, an attempt to implement a solution was made. The first prototype was only intended to confirm what kind of implementation type would work for the application. After that, code review with feedback was conducted.

The Java programming language was used in writing together with the IDE IntelliJ IDEA Ultimate and the build automation tool Gradle; which had some scripts with which to synchronize the React-built frontend with the backend. The implementation and feedback phases resulted in several iterations. A good design was made once the required functionality was attained.

3.4 Feedback phase

Version control was achieved through Git with Bitbucket, and a code review was conducted after every code commit. The feedback was provided by the senior engineers in the team through comments

on the pull requests and by following up on open discussions in Google Meet.

3.5 Validation phase

The validation was to be performed in three ways:

- Manually through demonstration, using the application while analyzing with the web browser's developer console.
- Code analysis by team members.
- Approval from customized Spring Boot test classes.

3.6 Communication

Communication with the company was carried out through Slack and Google Meet due to restrictions caused by the COVID-19 pandemic. The web application team had an online office environment in Google Meet where team members could check in and communicate with one another while working. For requests or communications with other teams within Klarna, Slack was used for more casual conversations, while Jira was used to create requests in the form of tickets. Every Friday, there was a retrospective held via Google Meet to update the stakeholders.

3.7 Changes during the project

More thorough work was planned for security testing. However, since the component had to be tailored with Okta's OIDC and OAuth 2.0 API, the implementation was more time consuming than expected. The security validation was therefore performed on Klarna's end.

3.8 Source criticism

Information about the references is presented in this section for the interested reader regarding the trustworthiness of all the sources.

Spring

Spring was founded by the creators of the Spring Framework; all their projects are free to use and open-source with an active community. Spring's products are widely used within Java Enterprise applications.

Internet Engineering Task Force

“The goal of the IETF is to make the Internet work better” is officially stated in the document Request for Comments (RFC) 3935 [17]. The IETF includes researchers and network developers, and it reaches its goal by stating and documenting standards for the Internet's infrastructure. These published standards aim for improved usability of the Internet, including the improvement of information exchange between different devices.

OpenID Foundation

As is the IETF, the OpenID Foundation is an internationally orientated group. Individuals as well as companies share ideas, experiences and interests for optimizing OpenID technologies. They are represented as a public trust organization.

Software Requirements: Styles and Techniques

The author of the book Software Requirements: Styles and Techniques, Soren Lauesen, is a professor emeritus of computer science at IT University of Copenhagen. The book is also used in the required engineering course ETSF30 at Lunds Tekniska Högskola (LTH).

UML Syntax

Lennart Andersson, the author of this document, was a professor in the computer science department at LTH when he published it.

The Open Web Application Security Project

Declared as a nonprofit foundation whose mission is to improve software security, the Open Web Application Security Project (OWASP) has tens of thousands of members who develop open-source software projects to help forward their mission. Several corporations have supported the OWASP and their work for over two decades.

Google Guava

Google Guava was Founded by Kevin Bourrillion, a senior engineer and Jared Levy, a senior staff software engineer. It is a set of common libraries for Java that carries the Apache License 2.0, meaning it is open-source. It was and is developed mostly by engineers from Google.

Okta

Okta, described in 2.5, provided practical information for the OAuth2.0 framework and the OIDC protocol. The information was validated against the IETF's official document as their information needs to be correct for the service to be safely implemented. The information was correct, but a more in-depth knowledge was gained from the IETF's official document.

134 Cybersecurity Statistics and trends for 2021

134 Cybersecurity Statistics and trends for 2021 is a blog post authored by Rob Sobers, a software engineer specializing in web security. He provided valid sources for all statistics provided in the

post. The post was published by Varonis which is a company that focuses on providing security software for organizations.

4 Analysis

4.1 Elicitation results

The results from the elicitations translate into the following specifications:

1. Requirements for authorization and authentication
 - 1.1. An authorization server provided by Okta will be used.
 - 1.2. Authorization code flow will be used.
 - 1.3. The implementation of the flows will follow Okta's implementation guidelines and documentation. Eventual exceptions will be communicated and evaluated before proceeding.
2. Requirements for the component
 - 2.1. Code for the component will be version controlled in Bitbucket with follow-up to each feedback comment.
 - 2.2. The component's classes and methods will be documented according to Javadoc standards.
 - 2.3. The component will not disrupt the possibility of using the older authorization components.
 - 2.4. The component will classify the larger responsibilities into separate classes.

- 2.5. The component will set all the tokens as cookies.
 - 2.6. The component will be initialized in accordance with Spring Framework's dependency injection.
 - 2.7. The component will wait five minutes until it revalidates an access token.
 - 2.8. The component will have an API endpoint that handles user logouts.
3. Security requirements
- 3.1 The tokens will be secured as follows:
 - 3.1.1. The tokens stored as cookies must have the HttpOnly attribute.
 - 3.1.2. The tokens stored as cookies must have the Secure attribute.
 - 3.1.3. The tokens stored as cookies must have the SameSite attribute set to Strict where possible and Lax where Strict is not possible.
 - 3.2. The refresh token will be revoked and replaced each time it issues a new access token.
 - 3.3. The implementation of the OAuth2.0 flow will include the recommended attribute "state" for the code generation.

3.3.1. State will be generated by a cryptographically strong random number generator and signed with a message authentication code (MAC).

3.4. No secret credentials will be stored in the created classes.

4. Project requirements

4.1. A retrospective update will be provided each Friday.

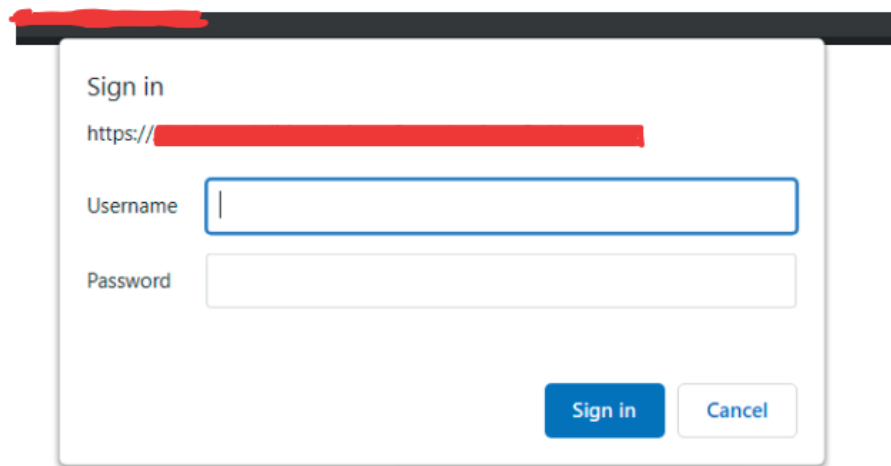
There was no need for a project board to follow the project's process because of the following:

- The work was performed alone, so there was no need to distribute different tasks.
- A retrospective update was provided each Friday.
- There was the possibility to attend a virtual office environment where communications could be carried out.
- The progress could be observed in Bitbucket.

These fulfilled the need for updating the stakeholders at Klarna.

4.2 Former authentication handling

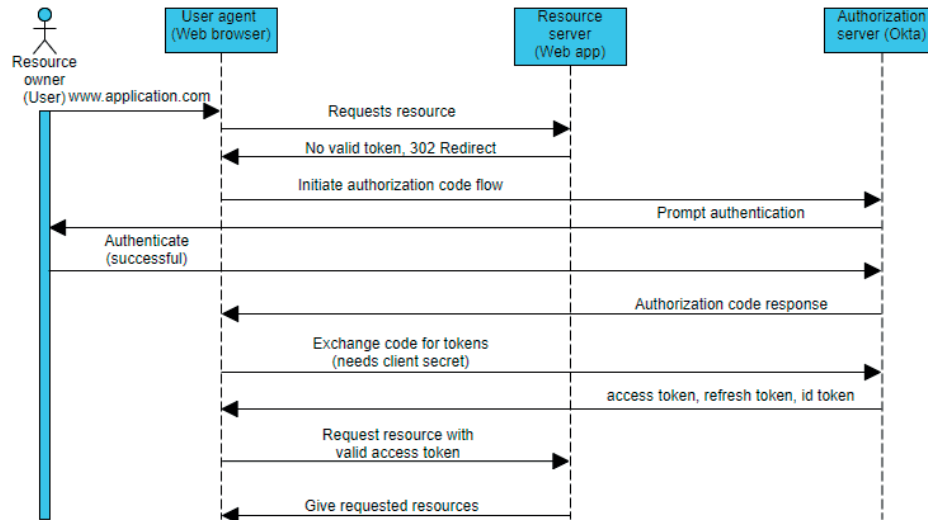
The former authentication consisted of having a list of created usernames and passwords for each user who needed access to the web application; this list was stored in a credential file inside the project. The web application was only reachable from Klarna's internal network but still needed an upgrade in terms of security and useability. The previous login form can be seen in Picture 2.



Picture 2. What the authentication form to be replaced looked like

4.3 Choice of authorization flow

An initial attempt was made to implement the OAuth2.0 authorization code flow with PKCE by following Okta's guide for signing in a user to a single-page application [11]. After a retrospective meeting, it was decided to instead implement authorization code flow due to the application having a backend that could securely handle credentials for Okta. The implemented flow is illustrated in Picture 3.



Picture 3. An example of the authorization code flow

Okta's OIDC and OAuth2.0 API were used instead of the software development kit (SDK) that Okta provided for Spring Boot. This was due to the web application not being compatible with Spring Security, which was necessary in order to use the SDK.

4.4 Encountered problems and solutions

The frontend could not handle 302 redirect responses; it was instead handled by responding with a 401 unauthorized response that included the redirect location in its body. XML Http Requests (XHR) did not support it, which was used for communicating between frontend and backend in this application.

5 Results

The results presented in this chapter are intended to provide insights for those who want to implement an authorization flow in accordance with OAuth2.0 and OIDC using a more tailored approach with API requests and responses. Okta acted as the authorization server and the resource server in this project. Screenshots of the authorization code flow are presented in Appendix A.

5.1 Starting the authorization code flow

The flow begins with a user not having a valid access token. The web application (client) then redirects the user agent (web browser) to Okta's authorization endpoint while including the necessary attributes as a query. The URI is constructed with the same attributes as in the following example except that it has different values for everything other than the **scope** and **response_type**:

```
https://{OrgOktaDomain}/oauth2/v1/authorize  
?client_id=0oabucvyc38HLL1ef0h7  
&response_type=code  
&scope=openid%20offline_access  
&redirect_uri=http://localhost:8080/login/callback  
&state=state-296bc9a0-a2a2-4a57-be1a-d0e2fd9bb601
```

The Okta domain is an attribute configured for the organization; it is retrievable from the administrator pages. The various query attributes are as follows:

client_id: Public ID for Okta to identify the registered application, retrievable in Okta's administration pages.

response_type: Needs to have the value "code" so that Okta's authorization server knows that it is the authorization code that is to be returned on a successful authentication.

scope: OIDC requests must have the “openid” value of the scope attribute. The scope “offline_access” must be specified to retrieve a refresh token while also having enabled it through Okta’s administration page.

redirect_uri: This is where Okta sends the authorization code in case of a successful authentication. The created component includes a REST API with its communicational endpoint being the redirect_uri’s value.

state: This should be generated as a cryptographically secure random value, stored in the backend and sent to Okta’s endpoint to later be validated against each other after a successful authentication. The value could be stored as a cookie for the backend to use but would then need to be validated with a signature, to make sure that the backend generated the state value. This is all to prevent cross-site request forgery, a way for an attacker to forge this query themselves and tricking an authorized user into logging in via their query, resulting in the attacker gaining the access token. [12].

5.2 Acquiring tokens with the authorization code

Given that the attribute values are the same as in 5.1, and the authentication is successful, the following step uses the authorization code in a new request to Okta’s token endpoint. Within the given scope, this returns an access token, a refresh token and an ID token.

With pseudo code, acquiring the tokens is accomplished in four steps:

1. Creating a HTTP POST object as a new request. The API’s endpoint for requesting tokens is
`https://{OrgOktaDomain}/oauth2/v1/token`
2. Appending headers

- 2.1. client ID with client secret in a basic authorization header, for validating the credibility with the request.
- 2.2. JSON in an accept header, making the response from Okta in the format of JSON
- 2.3. application/x-www-form-urlencoded for the content-type header, telling the browser how the data should be encoded before sending it to Okta's endpoint.
3. Setting the entity of the request as a StringEntity object
 - 3.1. grant_type = authorization_code
 - 3.2. redirect_uri = the redirect URI as used in 5.1
 - 3.3. code = authorization code value acquired from Okta's response to a successful request as in 5.1
 - 3.4. scope = openid%20offline_access
4. Executing the post and receiving a response in JSON format

An ObjectMapper object is used to translate the JSON response into a created class containing attributes of interest such as access token, refresh token, id token and the associated username. The tokens are set as secure cookies to be validated for continuous authorized usage of the web application. At this point, the flow is performed and can be started over if a user is unauthorized again.

5.3 Handling the authorization from the client

Once a user successfully passes the authorization code flow, their validated access token can be used for requests during five minutes before having to be revalidated. The five-minute leeway logic is constructed with a LoadingCache object provided from a Google Guava library [13]. Validation of a token is done against Okta's "introspect" endpoint, which is presented in the following pseudo code:

1. Creating a HTTP POST object with the following URI:
https://{**OrgOktaDomain**}/oauth2/v1/introspect

2. Appending headers
 - 2.1. client ID with client secret as an authorization header
 - 2.2. application /JSON in an accept header
 - 2.3. application/x-www-form-urlencoded for the content-type header
3. Appending a list of name/value pairs to the created POST object's entity
 - 3.1. token = value of token (extracted from a cookie)
 - 3.2. token_type = access_token or refresh_token
4. Executing the post and receiving the response in JSON format

The response contains an attribute showing whether the token is active. This is followed by one of three possible scenarios:

1. The access token is valid. The user agent gets access to the requested resources.
2. The access token is invalid. If the refresh token is valid, issue a new access token and rotate the refresh token.
3. Both the access token and the refresh token are invalid. Return unauthorized access and redirect to restart the authorization code flow.

5.4 Logout

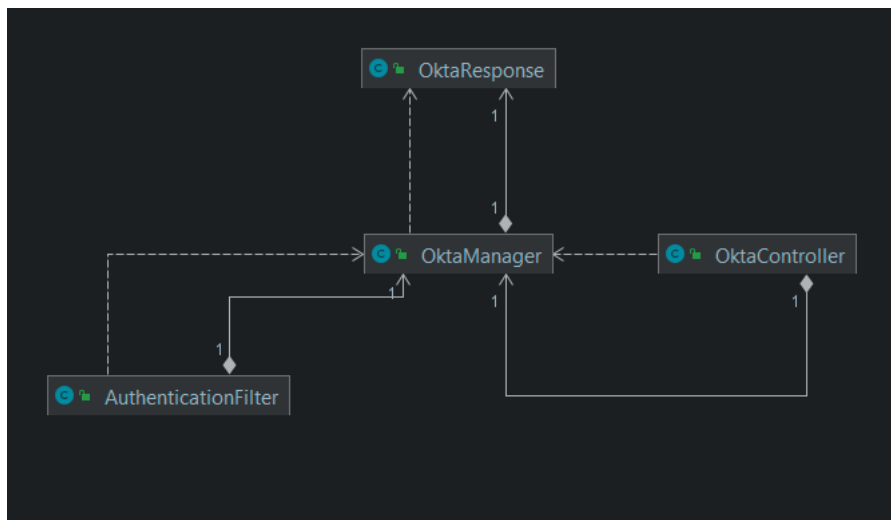
An API endpoint is implemented in the component to sign the user out. This is requested from the frontend when a logout button is clicked. Revoking tokens requires communication with Okta's revoke API endpoint. Signing the user out is carried out in seven steps:

1. Creating a HTTP POST object with the following URI:
https://{**OrgOktaDomain**}/oauth2/v1/revoke
2. Appending headers
 - 2.1. client ID with client secret as an authorization header
 - 2.2. application/JSON as an accept header

- 2.3. application/x-www-form-urlencoded for the content-type header
3. Appending the refresh token to the entity
4. Executing the POST, tokens revoked
5. Deleting all cookies
6. Redirecting to
`https://{OrgOktaDomain}/oauth2/v1/logout?id_token_hint={
ID token}`
7. Discarding all the entries in the LoadingCache

5.5 Design

The component is separated, as a package, from the authorization filter. It is included in a switch case with the different authorization methods. The switch variable is set during configuration, deciding what method to be used. An UML class diagram is presented in Picture 4 showing the relations to the classes they are coupled with according to the UML syntax documentation from Lennart Andersson [15]. Unfortunately, methods and attributes could not be shown due to terms in an agreement with Klarna.



Picture 4. The design as a class diagram. The created component

includes the classes `OktaResponse`, `OktaManager` and `OktaController`.

6 Summary

The summary includes answers to the questions presented in the problem formulation, followed by a reflection on ethical aspects as well as what possible development opportunities can be applied to the component in the future.

6.1 How does the current solution for authentication and authorization look for the web application?

Basic authorization. The web application's authorization procedure that was to be replaced is analyzed and presented in Chapter 4.2. Since the credentials are passed in an authorization header in every HTTP request, it is less secure than any other method that does not have that limitation. However, it is less vulnerable since the website of use is protected with HTTPS, meaning that the requests and responses are secured by transport layer security (TLS).

Having basic authorization has also resulted in negative user experiences since the users need to remember credentials for several different applications. (What happens when an employee leaves the company?) Then, the stored credentials need to be removed manually in every application. With Okta's SSO solution, employees can easily be centrally deactivated. Managing the access through Okta also provides better options for monitoring all the authorization requests and issued tokens.

6.2 What are the requirements in terms of security related to an Okta single sign-on solution?

1. It is important to choose the correct authorization flow. In this case, the web application stores a secret key used to communicate to Okta's endpoints. Communication should be

encrypted over TLS, which is included in the HTTPS protocol used by most of today's websites.

2. A value should be sent with the initial request in the authorization code flow, encrypted, signed and validated. This is to avoid cross-site forgery requests; it basically ensures that the request has come from a valid source.
3. Tokens need to be stored securely, in this case as cookies with all the security flags enabled. Rotation of refresh tokens is an effective feature to mitigate guessing the token's value.

6.3 How is a user-friendly authentication solution implemented?

The user-friendly authentication solution in this work achieves the following:

- Presents users with a familiar login-form.
- Allows users to authenticate themselves with their central set of credentials.
- Keeps users logged in if they are active without having to reauthenticate by performing the validation in the background.

6.4 What is one way to implement single sign-on via Okta for a web application?

There are several ways to implement SSO via Okta. The solution used in this work was through Okta's OIDC and OAuth2.0 API. It worked well and should be flexible enough to be applied in applications that uses different frameworks or programming language due to the communicational logic being HTTP requests.

6.5 Reflection on ethical aspects

As most of today's applications are available over various networks, it is highly important to ensure security, not only at the network level but also within applications. Sharing knowledge that can help implement a secure way of authorization is therefore important. In a blog article authored by Rob Sobers [16], he states that cloud breaches will increase due to the large increase of remote work, and according to a report published by IBM [17], the average cost for a data breach in 2020 was \$3.86 million USD, with an average of 280 days required to identify and contain the breach. Therefore, necessary security measures would be of great social benefit. This thesis also has the intention of sharing important knowledge to implement secure solutions.

6.6 Future development opportunities

The results from this project can help create more tools for secure authorization implementations, including more thorough research into how to test a component designed to authorize users. If someone were to implement their own identity platform with SSO functionality like Okta's, the results presented in this thesis could be used as a helping tool. Development opportunities for the implemented component could include well-constructed Java tests.

7 Terminology

Authentication – a process of proving that someone is who they say they are.

Authorization – giving permission to someone who is authenticated to do something. For example, allowing a certain authenticated user to make a server request.

API – abbreviation for Application Interface Protocol, acts as an intermediary layer between an application and the web server. An API processes data transfer between systems.

Central set of credentials – a username/password pair that is to be used for several different authentication processes.

Component – A component in this context is a package of the resulting Java classes that the web application uses.

Endpoint – one end of a communication channel. For example, a URI: `https://domain.com/endpoint`.

8 References

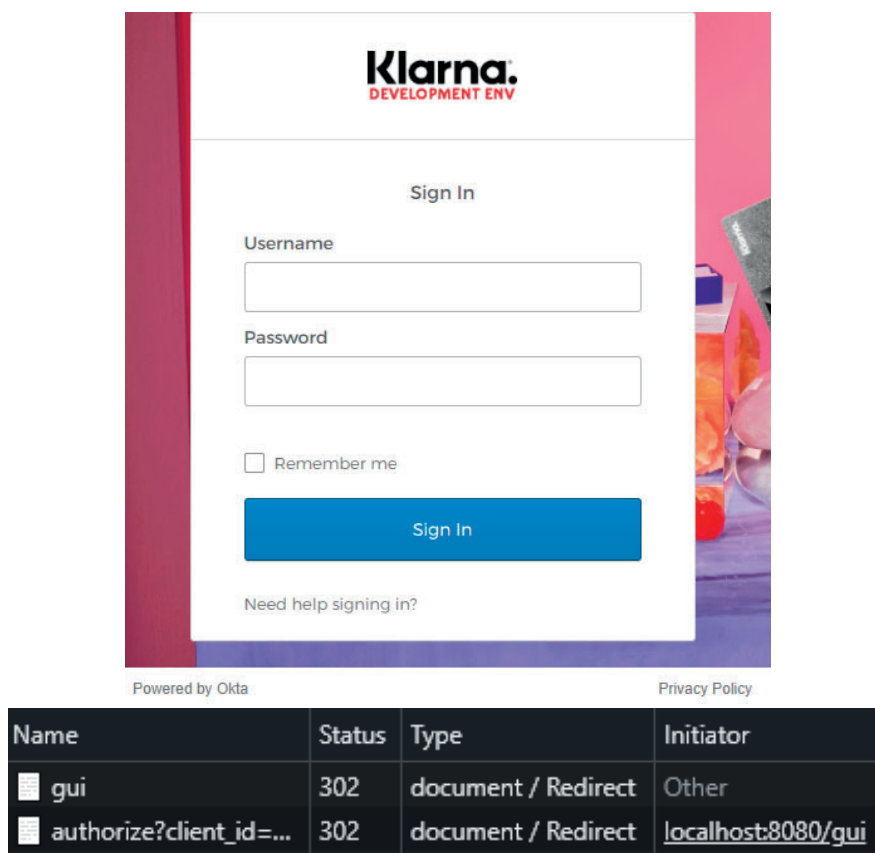
- [1] Spring, *Spring Framework*, [Online]. Retrieved: Aug 2021. Available at: <https://spring.io/projects/spring-framework>
- [2] Spring, *Spring Framework documentation*, Jul 2021. [Online]. Retrieved: Aug 2021. Available at: <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html>
- [3] Spring, *Spring Boot*, [Online]. Retrieved: Aug 2021. Available at: <https://spring.io/projects/spring-boot>
- [4] Jones, M., Bradley, J., and N. Sakimura, *JSON Web Token (JWT)*, May 2015, [Online]. Retrieved: Aug 2021. Available at: <https://datatracker.ietf.org/doc/html/rfc7519>
- [5] Hardt, D., Ed., *The OAuth 2.0 Authorization Framework*, Oct 2012. [Online]. Retrieved: Aug 2021. Available at: <https://www.rfc-editor.org/rfc/rfc6749>
- [6] Hardt, D., Ed., *The OAuth 2.0 Authorization Framework*, Oct 2012. [Online]. Retrieved: Aug 2021. Available at: <https://www.rfc-editor.org/rfc/rfc6749>
- [7] N. Sakimura, et al. *OpenID Connect Core 1.0 specification*, Nov 2014. [Online]. Retrieved: Aug 2021. Available at: https://openid.net/specs/openid-connect-core-1_0.html
- [8] S. Lauesen, *Software Requirements: Styles and Techniques*, Addison Wesley, 2002, p. 342-343.
- [9] Eastlake, D., Hansen, T, *US Secure Hash Algorithms*, May 2011. [Online]. Retrieved: Aug 2021. Available at: <https://datatracker.ietf.org/doc/html/rfc6234>

- [10] Okta Developer, *Sign users into your single-page application*, [Online]. Retrieved: Jul 2021. Available at: <https://developer.okta.com/docs/guides/sign-into-spa/react/before-you-begin/>
- [11] The OWASP Foundation, *Cross Site Request Forgery (CSRF)*, [Online]. Retrieved: Aug 2021. Available at: <https://owasp.org/www-community/attacks/csrf>
- [12] Google Guava, *Guava: Google Core Libraries for Java*, [Online]. Retrieved: Jul 2021. Available at: <https://github.com/google/guava>
- [13] OpenID, *OpenID Certifications*, [Online]. Retrieved: Aug 2021. Available at: <https://openid.net/certification/>
- [14] Lennart Andersson, *UML-syntax*, Jan 2013. [Online]. Retrieved: Aug 2021. Available at: <https://fileadmin.cs.lth.se/cs/Education/EDAA20/pdf/umlsyntax.pdf>
- [15] Rob Sobers, *134 Cybersecurity Statistics and trends for 2021*, Mar 2021. [Online]. Retrieved: Aug 2021. Available at: <https://www.varonis.com/blog/cybersecurity-statistics/>
- [16] IBM, *Cost of a Data Breach: Report*, July 2020. [Online]. Retrieved: Aug 2021. Available at: <https://www.capita.com/sites/g/files/nginej291/files/2020-08/Ponemon-Global-Cost-of-Data-Breach-Study-2020.pdf>
- [17] IETF, *A mission statement for the IETF*, Okt 2004. [Online]. Retrieved: Aug 2021. Available at: <https://datatracker.ietf.org/doc/html/rfc3935>
- [18] Atlassian, *Jira Software*. [Online]. Retrieved Nov 2021. Available at: <https://www.atlassian.com/software/jira>
- [19] Okta, *Beginner's Guide to SAML*, Sep 2021. [Online]. Retrieved Nov 2021. Available at: https://support.okta.com/help/s/article/Beginner-s-Guide-to-SAML?language=en_US

9 Appendices

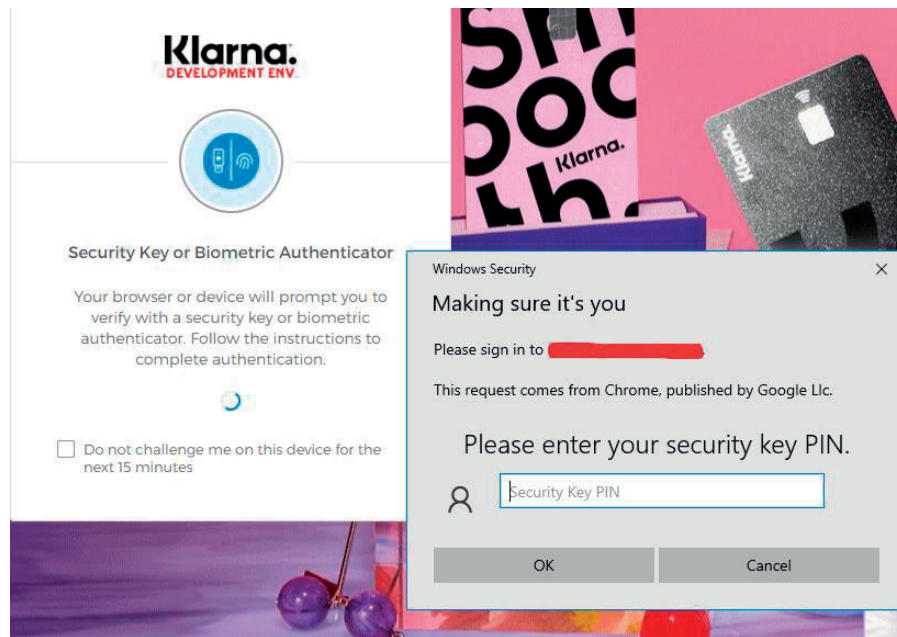
Appendix A. Screenshots of the login flow

The web application starts the authorization code flow by redirecting the user agent. A login form hosted by Okta is then presented. This is shown in Picture 6.





Picture 6. Screenshot of a login form together with the network requests viewed through Chrome's developer console.

When the user enters their correct credentials, MFA is triggered as shown in Picture 7.



Picture 7. MFA required to proceed with the authentication

After providing the MFA, the user is allowed access to the web application's resources. The HTTP requests and the set cookies are shown in Picture 8.

 callback?code=IEkSH4...	302	document / Redirect
 gui	302	document / Redirect

Name	Value	Domain	P...	Expires / ...	Size	HttpOnly	Secure	SameSite
idToken	eyJra...	localhost	/	Session	903	✓	✓	Strict
refreshToken	pm-...	localhost	/	2021-09-...	55	✓	✓	Lax
accessToken	eyJra...	localhost	/	2021-09-...	876	✓	✓	Lax
state	ehElj...	localhost	/	2021-09-...	32	✓	✓	Lax

Picture 8. The callback query response from Okta's authorization server includes the code with which to initially request tokens with.

After a call to the token endpoint, following cookies were set.

Appendix B. JWT

The anatomy of a JWT can be seen in Picture 9.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5cS

HEADER: ALGORITHM & TOKEN TYPE
<pre>{ "alg": "HS256", "typ": "JWT"}</pre>
PAYLOAD: DATA
<pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022}</pre>
VERIFY SIGNATURE
<pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret)</pre> <p><input type="checkbox"/> secret base64 encoded</p>

Picture 9. The anatomy of an encoded and decoded JWT, signed using the secure hash algorithm HMAC-SHA256 [10].

Source: Screenshot taken from <https://jwt.io/>



LUND
UNIVERSITY

Series of Bachelor's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-854
<http://www.eit.lth.se>