UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo para alumnos con Experiencia Profesional

TÍTULO: Integración SSO basada en componentes

AUTOR: Matías Iván Pompilio

DIRECTOR/A ACADÉMICO: Mg. Luis Mariano Bibbó

DIRECTOR/A PROFESIONAL: Adán Mauri Ungaro

CARRERA: Licenciatura en Sistemas

RESUMEN

En esta tesina se aborda la implementación de una solución de "Single Sign-On" (SSO) escalable y segura para sistemas web con arquitectura multi-tenant, basada en la necesidad de integrar un nuevo proveedor de autenticación en un entorno productivo. El trabajo detalla el proceso de investigación, diseño e implementación de una librería de componentes para facilitar esta integración, destacando los desafíos y lecciones aprendidas.

Palabras Claves

Desarrollo Web, Single Sign-On, Autenticación, Multi-tenan, Procesos de Software

Conclusiones

Tras las diferentes etapas realizadas de proceso de software, se integró exitosamente los diferentes proveedores de autenticación propuestos, mejorando el producto y generando un código legible para futuros desarrolladores. La arquitectura diseñada logró ser flexible y escalable, permitiendo adaptaciones a futuras actualizaciones y necesidades del sistema, asegurando su crecimiento y evolución conforme a los requerimientos del negocio y usuarios.

Trabajos Realizados

Análisis de requerimientos y planteamiento de un diseño de solución para la problemática de autenticación en entornos web con una arquitectura Multi-tenant. Reimplementación de código legacy y desarrollo de componentes para una aplicación web React con el objetivo de integrar diferentes proveedores de autenticación, en este caso Okta y Azure AD.

Trabajos Futuros

Integración de nuevos proveedores de autenticación.
Extensión de la librería de componentes implementada para la utilización de diferentes proveedores de identidad, así como la integración de otros flujos o protocolos de autenticación,
realizando comparativas entre las diferentes herramientas utilizadas.
Modificación de la librería para su utilización en otros tipos de arquitecturas como

Fecha de la presentación: AGOSTO 2024

Integración SSO basada en componentes

Matías Iván Pompilio

7 de agosto de 2024

Índice general

Índice de figuras

Capítulo 1

Resumen

1.1. Motivación

En el pasado, cuando los usuarios necesitaban utilizar los diferentes servicios de internet requerían crear una cuenta para cada uno y recordar múltiples credenciales [1] . Sin embargo, con el avance de la tecnología, surgieron soluciones innovadoras para simplificar este proceso, una de las cuales es el "Single Sign-On"(SSO) [2] [3]. Esta tecnología permite a los usuarios acceder a múltiples sistemas utilizando una sola instancia de identificación, lo que significa que solo necesitan autenticarse una vez, recordando un solo par de clave y contraseña. La implementación del SSO ha demostrado ser una mejora significativa en la experiencia de usuario y en la seguridad de las cuentas, al reducir la necesidad de gestionar las credenciales, se minimiza el riesgo de contraseñas débiles o reutilizadas, disminuyendo así las posibilidades de brechas de seguridad.

En el mercado actual, existen diversas soluciones que facilitan la implementación del SSO como servicio [4], ejemplos de estas podrían ser Okta, Azure Identity Manager, Keycloak, entre otros. Estos proveedores ofrecen una capa de seguridad adicional, eliminando la necesidad de desarrollar desde cero un proceso tan sensible. En mi experiencia como desarrollador, en los diferentes proyectos de los que participé pude detectar que esta opción es atractiva para las empresas, ya que pueden agilizar la implementación de estos tipos de mecanismo y asegurarse una aplicación mucho más segura, ya que este tipo de soluciones se mantiene continuamente actualizada y en mantenimiento.

Sin embargo, en el contexto de arquitecturas multi-tenant, aparecen nuevos desafíos. En este tipo de arquitecturas los recurso de un sistema, tales como software y hardware son compartidos entre todos los usuarios que acceden a la aplicación, sin compartir necesariamente datos. En estos entornos, una sola instancia de la aplicación sirve a múltiples clientes, y cada cliente tiene su propio grupo de usuarios que comparten el acceso a la misma instancia de software [5] [6]. Durante el ciclo de vida de una aplicación, la cantidad de usuarios que este

posee puede variar y, además, cada cliente puede utilizar su propio proveedor de autenticación. Esto crea la necesidad de que el producto sea escalable, permitiendo una integración de nuevos posibles proveedores de autenticación, como también que se mantenga actualizado para asegurar que el sistema sea seguro.

Esta tesina surge de una experiencia propia ante un problema real en la que se engloban los tópicos mencionados anteriormente. En este problema, la empresa en donde me desempeñaba como desarrollador se encontró ante la necesidad de integrar un nuevo proveedor de autenticación a su sistema productivo con una arquitectura multi-tenant, el cual ya contaba con una implementación SSO fuertemente acoplada y difícil de mantener utilizando Okta. A través de este caso, la tesina busca demostrar los conocimientos que obtuve a lo largo de mi carrera laboral y mis estudios universitarios en la Facultad de Informática de la Universidad Nacional de La Plata para la investigación, diseño y desarrollo de la solución.

1.2. Objetivo

Esta tesina explica el proceso que se llevó a cabo para la implementación de una librería de componentes con el objetivo de proveer una solución SSO escalable, eficiente y segura para un sistema web con una arquitectura multi-tenant. Se presentarán las diferentes etapas por la que pasó dicho proceso, desde la investigación de las diferentes tecnologías, hasta el diseño y la implementación de la solución, para luego analizar las lecciones aprendidas durante el mismo.

1.3. Organización de la tesina

La organización de este documento se encuentra dividida de la siguiente manera:

En el capítulo 2 se identificará la arquitectura del sistema y los requerimientos propuestos.

En el capítulo 3 se detallará el análisis realizado en la implementación inicial y la investigación en el marco teórico de la autenticación SSO, así como los distintos estándares y protocolos involucrados. En este punto se buscó identificar los requisitos principales para la implementación SSO, centrándose en aspectos de seguridad y en los flujos de proceso para realizar la autenticación.

En el capítulo 4 se hablará del proceso que se tomó para la refactorización del sistema de autenticación. En este mismo se identificaron los problemas de la implementación previa y se desarrolló un diseño para la solución.

En el capítulo 5 se detalla los trabajos realizados para la integración de un nuevo proveedor de autenticación para el sistema.

En el capítulo 6 se realizará una conclusión de los resultados obtenidos. Esto incluye una reflexión de como esta solución busca facilitar futuras implementaciones y, además, se hará un análisis de la escalabilidad y seguridad final de la solución.

Capítulo 2

Análisis de la Arquitectura y Seguridad del Sistema

2.1. Sistema y arquitectura

La empresa, de la que presté servicio como desarrollador, implementó un producto que ofrece la monitorización de los diversos dispositivos que componen la red de sus clientes. Este sistema consta de un hardware que se instala físicamente en la localización del cliente, el cual se conecta a sus red y monitoriza el tráfico en tiempo real. Cada vez que la empresa establece un acuerdo con un nuevo cliente, se debe instalar este dispositivo en el entorno físico de dicho cliente. Una vez instalado este hardware, los usuarios pueden acceder a una aplicación web la cual comparte vistas con la información de los de la red, informando características de los dispositivos, los posibles riesgos de seguridad y, además, realizar acciones en la red ante estos posibles riesgos. Acciones tales como establecer alertas, bloquear dispositivos, entre otras. Algunos ejemplos de las clases de establecimientos en donde se ha implementado este sistema son hospitales, universidades, oficinas y otros entornos similares. Es importante remarcar que se supervisa una amplia gama de dispositivos, que van desde computadoras personales y teléfonos smartphone hasta televisores, consolas de videojuegos e incluso dispositivos médicos.

En cuanto a la arquitectura se puede explicar, en un alto nivel, analizando el camino por donde fluye la información. Es decir, desde que el dispositivo recolecta los datos de la red, hasta que estos son expuestos como información para un usuario en el sistema. Comenzando desde la recolección de información, los datos de una red son recolectados por el dispositivo físico instalado en el establecimiento del cliente. Una vez recolectados, estos son procesados por diferentes servicios, los cuales se encargan de enriquecer la información. Esta información previamente procesada, es almacenada en diferentes bases de datos dependiendo del tipo de

información. La información almacenada es accedida y manipulada mediante microservicios, los cuales exponen una API para que finalmente una aplicación frontend SPA (Single Page Application) implementada en React, genere visuales sobre los datos obtenidos de los microservicios, con el objetivo de que el usuario logre consumir y manipular esta información.

Es necesario destacar que este sistema cuenta con una arquitectura multi-tentant, esto significa que existe una única instancia del sistema a la que múltiples clientes o tenants tienen acceso [7]. En este sistema se comparten una única instancia de la aplicación web, microservicios y una única base de datos de las cuales todos los usuarios acceden. Cada tenant es una organización que posee un grupo de usuarios los cuales comparten estos recursos. Por lo tanto, en este sistema, cuando se incorpora un nuevo cliente o tenant es necesario instalar el dispositivo en su entorno físico y luego proporcionar a los usuarios las credenciales necesarias para acceder al sistema y así acceder a la información recolectada en la red.

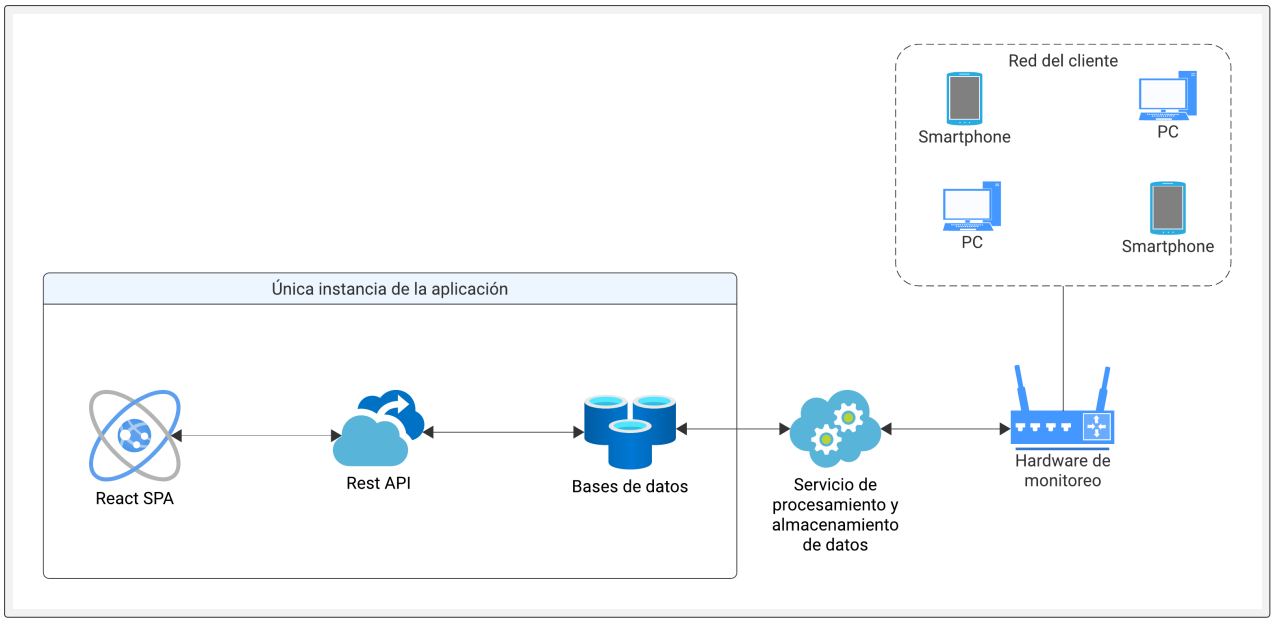

Figura 2.1: Arquitectura del sistema a alto nivel

2.2. Seguridad del sistema

Debido a que una arquitectura Multi-tenant implica que varios clientes compartan recursos, almacenamiento y servicios dentro del mismo sistema, la seguridad se convierte en una parte esencial del proceso de desarrollo. En la ingeniería de seguridad de estas arquitecturas, se destacan dos conceptos clave [8]. El primero se refiere al aislamiento de la información de cada cliente, lo que significa que cada uno solo puede acceder a sus propios datos, garantizando así la privacidad y confidencialidad de la información. El segundo concepto se relaciona

con la capacidad de permitir a cada cliente aplicar sus propios requisitos de seguridad. Esto implica que cada cliente pueda ajustar y modificar la configuración de seguridad según sus necesidades y preferencias específicas, lo que contribuye a una experiencia de usuario más segura y personalizada para cada grupo de usuarios.

Estos conceptos serán analizados en el sistema en cuestión, poniendo en perspectiva el frontend del mismo, área de la que se enfoca el desarrollo de esta tesina. Por lo tanto, para poder explicar como se satisfacen estos conceptos es necesario introducir en los mecanismos de autenticación [9] que el sistema provee para el acceso a la información por parte de los usuarios.

2.2.1. Mecanismos de autenticación

Para proveer el acceso al sistema a los múltiples grupos de usuarios que existen, el sistema necesita de un mecanismo de autenticación que permita identificar a cada usuario asignándolo a un tenant en especifico de manera que, cuando se inicie sesión en la aplicación web, la información que visualice el usuario sea la del tenant del que forma parte.

En el caso de este sistema, este provee dos mecanismos diferentes de autenticación los cuales, al momento de integrar al tenant en el producto, el cliente elige el tipo de mecanismo por el cual quiere autenticar a todos sus usuarios. Los mecanismos implementados y provistos para los clientes: basado en sesión o SSO mediante un proveedor externo, en este caso Okta [10].

En el caso de que el cliente escoja la autenticación por sesión, se le crean las cuentas asociadas para el ingreso al sistema, asociando estas cuentas al tenant del que forman parte. Por lo tanto, para que un usuario se autentique requiere ingresar utilizando sus credenciales de usuario y contraseña únicas (no se repiten entre tenants) en un formulario simple de inicio de sesión. Este mecanismo de autenticación que el sistema provee no permite una personalización en los requerimientos de seguridad para cada cliente, debido a que es un mecanismo de autenticación ya incorporado en el sistema e implementado por nuestro equipo de desarrollo, de manera que el cliente deberá adaptarse a las reglas de seguridad que el mismo sistema posee.

En cambio, si el cliente escoge el mecanismo de autenticación SSO mediante el proveedor externo, se requiere crear una nueva aplicación Okta (en caso de que el tenant no posea una), y se le provee un único acceso a un administrador, de manera que sea el encargado de crear las cuentas para su grupo de usuario. De esta forma el cliente con su cuenta de administrador tiene total acceso para administrar y configurar los requerimientos de seguridad, cumpliendo el segundo concepto sobre permitir a cada tenant configurar sus propios requerimientos de seguridad.

Finalmente, para permitir que el grupo de usuarios del tenant logre acceder a su meca-

nismo de autenticación elegido, cada tenant posee un subdominio de la aplicación por el cual acceder y la única instancia de la aplicación sabe identificar a que mecanismo pertenece dicho subdominio. A modo de ejemplo, si se accede al sistema mediante la url *tenant-1.app.com*, el sistema muestra una pantalla con el formulario típico de inicio de sesión, el cual pertenece al mecanismo de autenticación por sesión. En cambio, cuando la aplicación es accedida mediante *tenant-2.app.com*, el sistema redirige al usuario al portal de acceso del proveedor de autenticación, en este caso Okta. Cabe destacar que por más de que se pueda acceder a los subdominios de cualquier tenant, se requieren las credenciales para acceder a cada uno, y los usuarios son únicos entre los diferentes tenants, aislando completamente la información de cada uno.

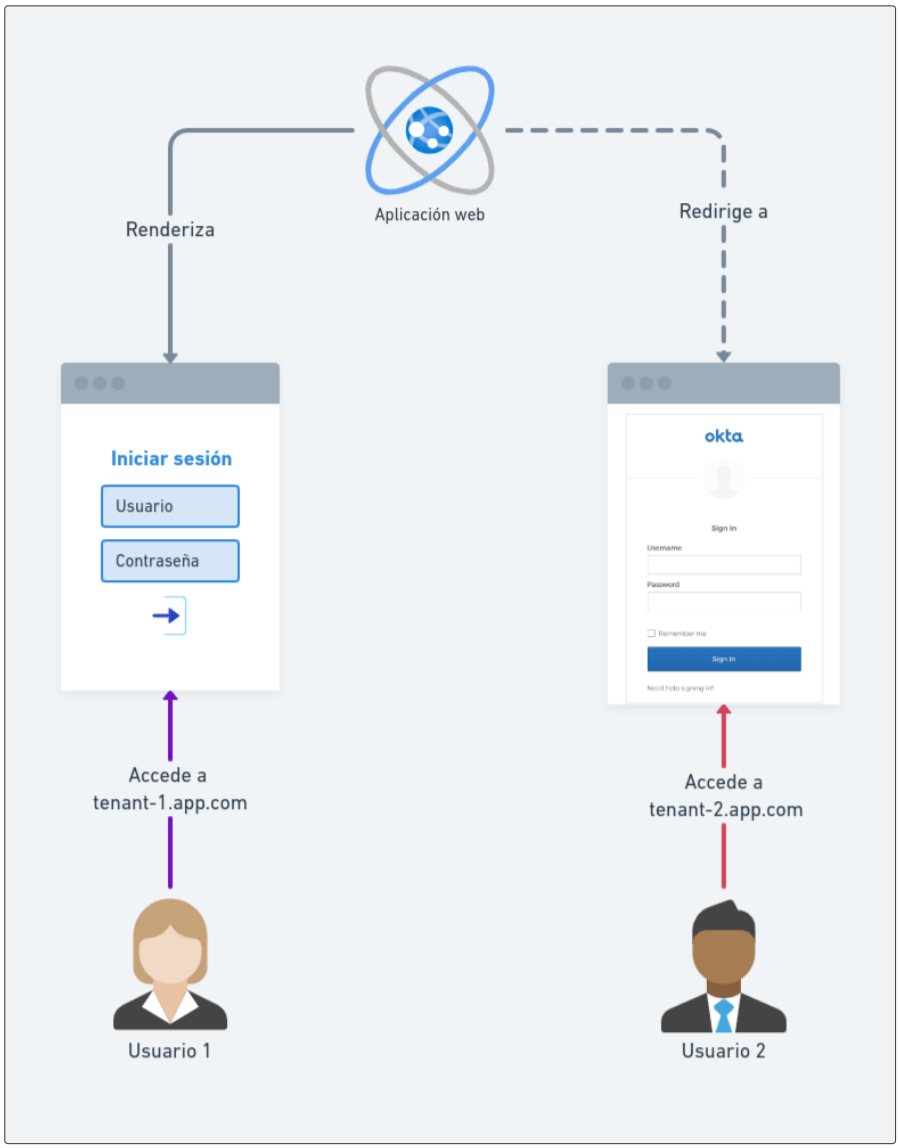

Figura 2.2: Selección de mecanismo de autenticación

2.2.2. Autenticación por sesión

Como se mencionó anteriormente, este es uno de los mecanismos de autenticación que es optativo para el tenant. En la autenticación basada en sesión la aplicación web provee una pantalla de login por la cual los usuarios se identifican con sus credenciales (usuario/-contraseña). Estas credenciales son enviadas por la aplicación web, a un servicio interno del sistema para su validación. En caso de ser aceptadas, el servicio genera una sesión o cookie la cual es almacenada para luego ser enviada al cliente que inició la petición de sesión. El cliente utiliza esta cookie para autenticar las peticiones siguientes que lo requieran y de esta manera persistir la sesión del usuario. Es importante destacar que este tipo de autenticación tiene la limitación de que es necesario almacenar cada sesión de los usuarios en el servidor, haciéndolo poco escalable.

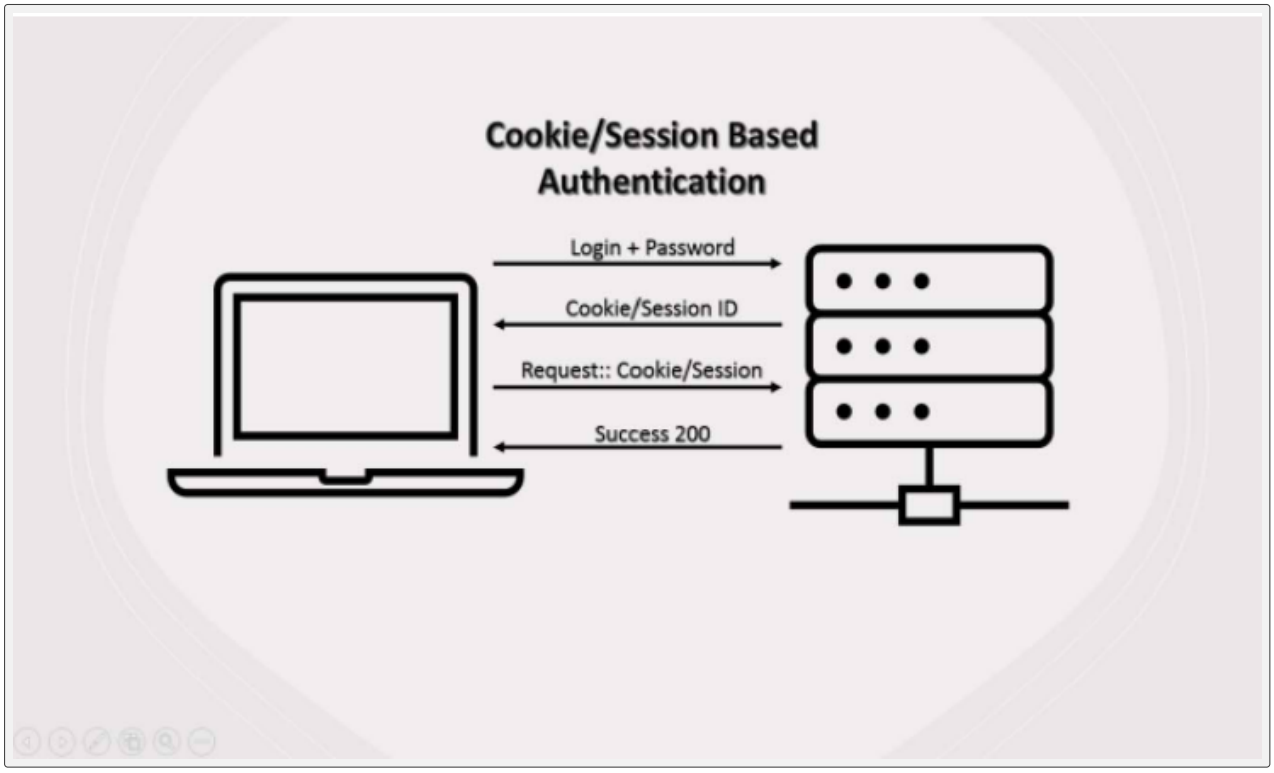

Figura 2.3: Diagrama de autenticación por sesión presentada en [9]

En este tipo de mecanismo, las credenciales de los usuarios son válidas únicamente para este sistema y los servicios que lo conforman. En caso de que la organización tenga otros sistemas por los cuales los usuarios también requieran autenticarse, éstos necesitarán recordar las credenciales específicas para cada sistema. Por lo que este tipo de autenticación no es recomendable para organizaciones con una infraestructura grande y que administren múltiples servicios privados.

2.2.3. Autenticación SSO

Por otro lado, otro de los mecanismos que son ofrecidos a los clientes al momento de adoptar el sistema es la autenticación mediante Single Sign On. Este método de autenticación se define como un mecanismo que utiliza una sesión centralizada y permite a un usuario acceder y autenticarse a diferentes servicios con un único set de credenciales [11]. Este proceso permite a los usuarios autenticarse una sola vez a los diversos sistemas que puede tener la organización [3]. En este tipo de mecanismo existe un proveedor de identidad central (IdP) que permite identificar al usuario y controlar su sesión a través de toda la plataforma. Este proveedor es totalmente configurable, tanto en la visualización de pantallas de inicio de sesión como también requerimientos de seguridad y métodos de autenticación.

Teniendo en cuenta estas características mencionadas, este mecanismo de autenticación puede ser recomendable para organizaciones las cuales mantienen múltiples sistemas que requieran la autenticación de los usuarios.

Una característica de este tipo de mecanismo de autenticación, que lo diferencia del método de autenticación por sesión, es el uso de un mecanismo conocido como Access token [9]. A diferencia de la autenticación basada en sesiones, donde la información de sesión se almacena en una cookie en el servidor, en la autenticación basada en tokens, la sesión del usuario no depende de una cookie. En su lugar, se generan tokens de acceso con un tiempo de expiración específico. Estos tokens poseen la estructura denominada JWT (JSON Web Token), la cual será explicada en profundidad en la sección A. Estos tokens de acceso son emitidos por el proveedor de identidad tras la autenticación exitosa del usuario y son almacenados en el lado del cliente, generalmente en el almacenamiento local del navegador de cada usuario. Cada vez que el cliente realiza una solicitud al servidor, el token de acceso correspondiente se incluye en la solicitud. El servidor entonces valida el token con el proveedor de identidad para garantizar su autenticidad.

Al no depender de sesiones mantenidas en el servidor, se reduce la carga en la infraestructura y se simplifica la gestión de sesiones. Además, al tener un tiempo de expiración definido, los tokens de acceso ayudan a mitigar los riesgos de seguridad al limitar la ventana de exposición en caso de que un token sea comprometido.

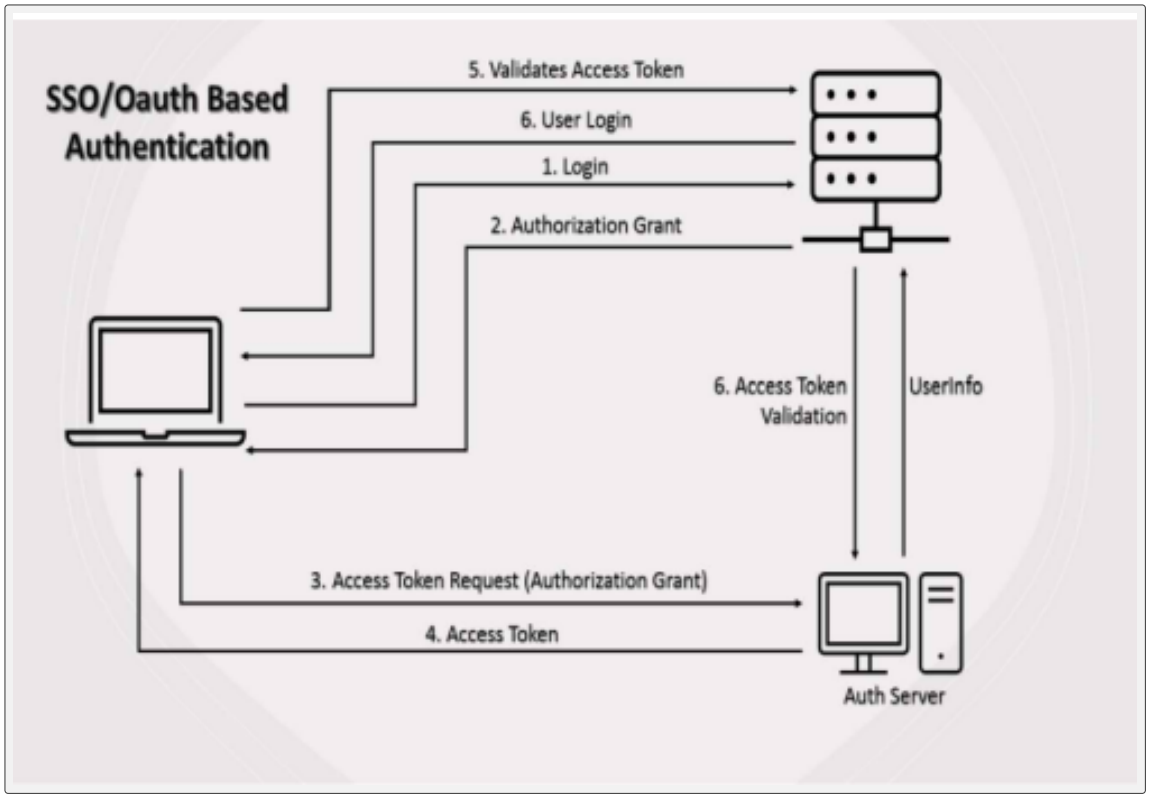

Figura 2.4: Diagrama de autenticación SSO presentada en [9]

Por último es necesario destacar que para la implementación del IdP, el sistema utiliza un software de terceros conocido como Okta [10]. Okta es una plataforma de software como servicio (SaaS) que ofrece diferentes opciones para la implementación de Single Sign-On (SSO) [12], todas ellas configurables y adaptables, lo que permitía que los diferentes tenants puedan personalizar sus criterios de seguridad.

2.3. Requerimientos

La aparición de un potencial cliente generó nuevas necesidades para la empresa en la que ejercía como desarrollador. Este cliente, interesado en el producto que manteníamos, estableció un requisito para su adopción: la integración del sistema debía permitir la autenticación mediante un proveedor SSO diferente, en este caso, Azure Active Directory. Este requisito nos planteó un conjunto de desafíos significativos que necesitaron ser abordados previo al desarrollo:

Debido a que el sistema de autenticación llevaba un gran tiempo productivo, con un código poco legible y desactualizado, y que los autores de la implementación ya no formaban parte del equipo, se reconoció la necesidad de una refactorizar la integración de Okta SSO en

el frontend. Por lo que se planteó generar una librería de componentes capaz de solucionar el sistema de autenticación de una manera robusta, segura y mantenible, buscando una solución que sea sencilla de entender y seguir, de manera que permita su futuro mantenimiento y ampliación.

Y finalmente en la implementación de la autenticación a través de Azure Active Directory, se buscaría ampliar el patrón de implementación creado en la refactorización de Okta, de manera que se integre satisfactoriamente al sistema. Esto implica no solo configurar correctamente el nuevo flujo de autenticación provisto por Azure AD, sino también, garantizar la seguridad y la compatibilidad con el sistema existente.

Capítulo 3

Análisis de la implementación Single Sign-On

El primer paso que se tomó para cumplir con los requerimientos fue analizar la implementación de autenticación que en ese momento generaba un gran esfuerzo mantener. Esta decisión se tomó ya que el código llevaba un gran tiempo productivo y que los autores de la implementación ya no formaban parte del equipo. Por lo tanto se necesitó de ganar conocimiento y generar confianza en el código para, finalmente, cumplir con el objetivo final de tener una implementación de autenticación robusta y segura.

Para este análisis se buscó identificar los elementos principales con los que el código implementaba el flujo de autenticación SSO.

3.1. Protocolo de autenticación SSO

Dada la diversidad de protocolos disponibles para la implementación del Single Sign On [3], y considerando que estos protocolos suelen ser integrados utilizando las herramientas proporcionadas por los proveedores externos, surge la necesidad de tener una visión clara del flujo de autenticación [13]. Por lo tanto, se realizó un análisis detallado del protocolo utilizado por el proveedor de SSO en el sistema actual. Este análisis se justificó por la complejidad inherente a la implementación de SSO, que involucraba la interacción de múltiples componentes. Al comprender a fondo el protocolo específico utilizado por el proveedor de identidad, se podía obtener una comprensión más clara de cómo se lleva a cabo el proceso de autenticación en el sistema actual. Esto nos permitió identificar posibles puntos de mejora, así como garantizar una integración fluida y segura con el nuevo proveedor SSO.

Para iniciar con este análisis comenzamos por verificar la configuración que tenía el proveedor para la aplicación, permitiéndonos saber cual eran los parámetros y características

del sistema actual. Para realizar esto fue necesario contar con el acceso al panel de administración de Okta en un ambiente de pruebas, es decir, un ambiente que sea una réplica del sistema productivo en cuanto a los recursos, pero sin acceso a los datos de los usuarios reales. En este caso, se utilizó el denominado ambiente de Staging, ambiente del cual el equipo utilizaba para testear la aplicación y tenía una replica de la configuración utilizada para los diferentes tenants que conformaban al sistema. Al acceder a este panel, se identificó que el protocolo utilizado por el sistema y por el cual se realizaba el flujo de autenticación era mediante el protocolo OpenID Connect.

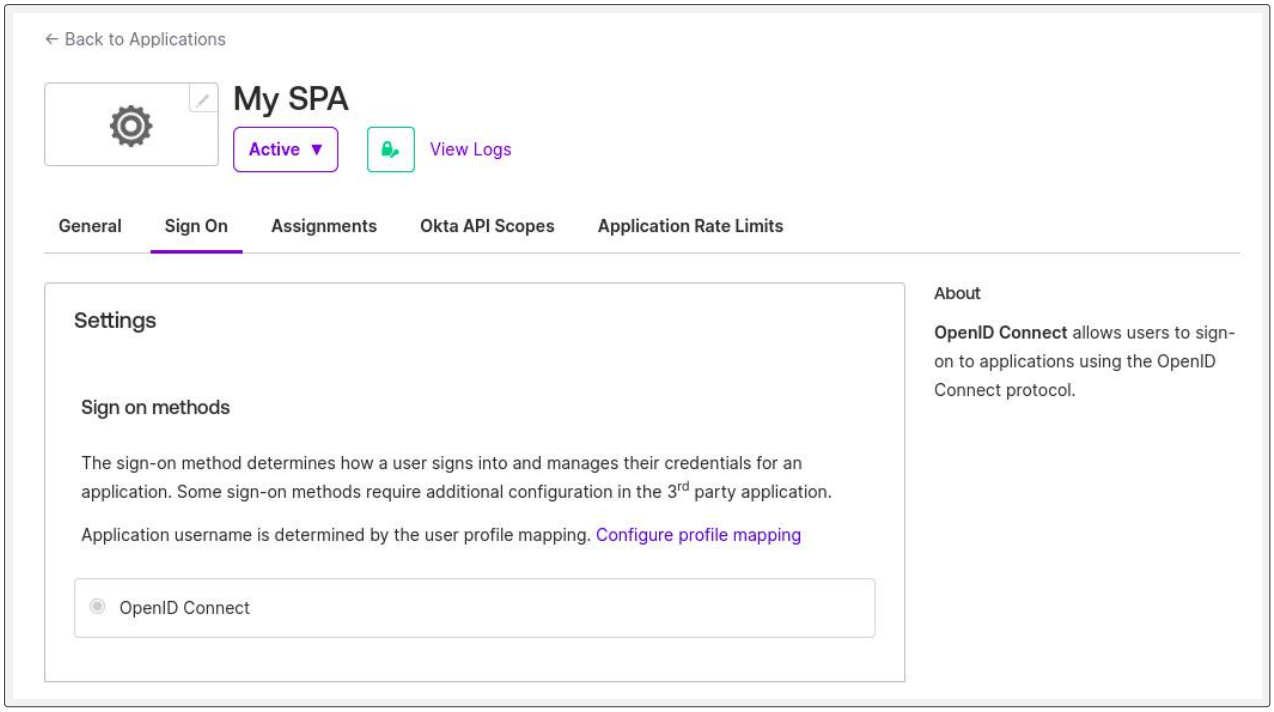

Figura 3.1: Ejemplo del panel de configuración de Okta para una aplicación OIDC

De esta manera el equipo ya tenía un enfoque por el cual empezar la investigación. La idea entonces fue, analizar el flujo por el cual este protocolo llevaba a cabo la autenticación de manera que nos permitiera tener una visión del proceso que se realizaba en el frontend y así lograr un mayor entendimiento del código implementado para llevar a cabo los cambios necesarios por los requerimientos del cliente. Por lo que a continuación se detallarán los hallazgos obtenidos en cuanto a la definición del protocolo utilizado por el sistema, en este caso OpenID Connect y el framework directamente relacionado, OAuth 2.0.

3.1.1. OpenID Connect y OAuth 2.0

OpenID Connect es una capa de autenticación construida sobre el framework de autorización de OAuth 2.0 [14] [15] . Debido a que OAuth 2.0 es un framework de autorización [16], es decir, solo se encarga de la autorización de aplicaciones frente a servicios externos, OIDC extiende este framework y permite a los servidores de autorización la autenticación de usuarios de una manera estándar.

Los autores Yvonne Wilson y Abhishek Hingnikar definen el proceso de autenticación del protocolo OIDC [15] de la siguiente manera:

Cuando un usuario accede a una aplicación, esta redirige el navegador del usuario a un servidor de autorización el cual implementa OIDC, el cual es llamado por este protocolo como proveedor OpenID. El usuario interactúa con el proveedor para autenticarse y, una vez autenticado, el navegador es redirigido de vuelta a la aplicación, la cual puede solicitar que se devuelva la información del usuario autenticado en un token de seguridad, llamado token de identificación o ID Token. Alternativamente, puede solicitar por un Access Token, el cual es definido en el protocolo OAuth 2.0 y puede ser usado posteriormente para llamar al endpoint provisto por el proveedor OpenID para obtener la información del usuario cuando se lo requiera.

Terminología

Existen tres roles involucrados en una solución OpenID connect [15], algunos de estos son definidos por el protocolo OAuth 2.0 [16], pero que la idea es reutilizada por este protocolo:

- **Usuario final**. El usuario que es autenticado.
- **Proveedor OpenID (OP)**. Este proveedor es un servidor de autorización que implementa OAuth 2.0 y es el encargado de autenticar a los usuarios y retornar información sobre los mismos.
- **Aplicación Confiable o Relying Party (RP)**. Es un cliente OAuth 2.0, que se encarga de delegar la autenticación al proveedor OpenID, y solicita la información del usuario autenticado. En el caso de este documento, es la aplicación mantenida por el equipo.

Otras terminologías que estos protocolos definen [14] [15] son :

- **ID Token**. Un token generado por el proveedor OpenID que contiene información acerca del usuario final en forma de claims.

- **Authorization Code o Código de autorización**. Un código retornado por parte del proveedor a la aplicación utilizado para obtener un Access Token y opcionalmente un Refresh Token. Cada código de autorización es utilizado una única vez.
- **Access Token**. Un token utilizado por la aplicación para acceder a una API privada. Es la forma que tiene la aplicación para identificar al usuario sin la necesidad de enviar las credenciales.
- **Refresh Token**. Un token opcional que puede ser usado por la aplicación para solicitar un nuevo Access Token, cuando este expira, sin tener que autenticar nuevamente al usuario.
- **Claims**. Conjunto de datos acerca del usuario final.

3.1.2. Flujo de autenticación

Una vez identificada la definición y terminología del protocolo, debido a que existen diversas implementaciones por los cuales OIDC puede realizar la autenticación [14], el siguiente paso que se tomó fue el de analizar el flujo utilizado por la implementación inicial de OIDC. No es el objetivo de esta tesina indagar en cada tipo de mecanismo pero sí se ve necesario explicar el utilizado por el sistema, ya que en el momento de la investigación fue importante su aprendizaje para comprender el comportamiento del sistema en la autenticación.

Para lograr identificar el mecanismo que se estaba utilizando en ese momento en el sistema, se inició observando la configuración nuevamente en el panel de administración de Okta. En este panel se identificó que estaba establecido el flujo llamado Authorization Code flow with PKCE.

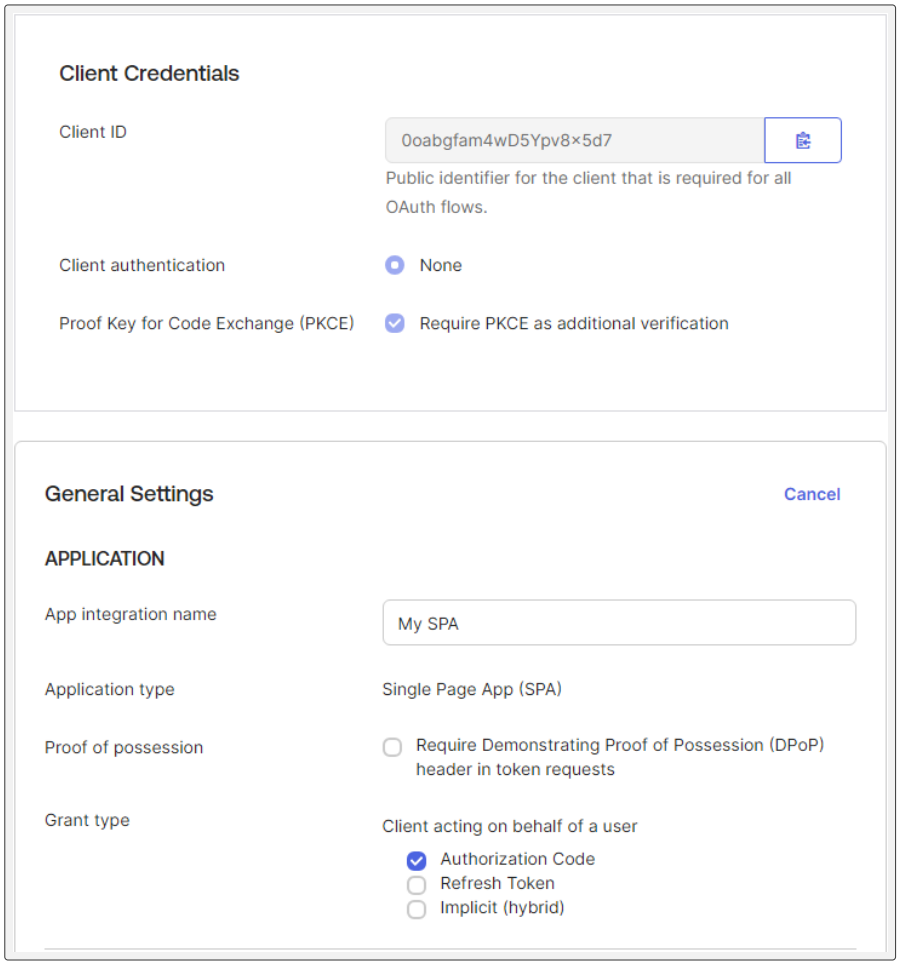

Figura 3.2: Ejemplo de configuración de Okta para una aplicación OIDC utilizando Authorization Code flow con PKCE

3.1.3. Authorization Code flow con PKCE

El flujo de autenticación Authorization Code with PKCE [17] es una extensión del mecanismo llamado Authorization Code, en el cual se reemplaza en el proceso de autenticación, el uso de una clave secreta del cliente (client secret) con un par de valores dinámicos, de los cuales se derivan el nombre PKCE (Proof Key for Code Exchange o Clave de prueba para el intercambio de códigos). Estos valores dinámicos están conformados por un código de verificación (code verifier) y un código de desafío (code challenge). Esto permite que las aplicaciones públicas autentiquen a los usuarios de manera segura sin la necesidad de incluir secretos que podrían ser manipulados.

Este flujo es recomendado en la documentación de Okta para utilizar en aplicaciones públicas [14], debido a que en tipo de aplicación los usuarios tienen acceso al código fuente de la aplicación y pueden modificarlo libremente. Si tomamos el sistema en cuestión, la

decisión de utilizar este flujo de autenticación fue acertada, ya que la aplicación que requiere autenticación es pública, la cual es accedida por múltiples usuarios a través de una dirección pública a través del internet.

Según la documentación oficial de Okta [18], este flujo se comporta de la siguiente manera: Inicia generando, del lado del cliente, una cadena de caracteres aleatorios llamado código de verificación (code verifier). Este código luego se codifica creando el llamado código de desafío (code challenge). Cuando la aplicación cliente inicia la primera etapa del flujo, este dispara la petición del código de autorización (Authorization Code) enviando el código de desafío en la misma. El IdP responde presentando una pantalla de autenticación para el usuario y, una vez que este se autentica ingresando sus credenciales, el proveedor responde con el código de autorización y lo asocia con el código de desafío. Para la segunda etapa, el cliente genera una petición del token enviando el código de autorización y el código de verificación. Luego, el proveedor procede a verificar esta información volviendo a computar el desafío utilizando el código de verificación (asociado previamente al código de autorización). Finalmente, si esta información es válida, el proveedor responde con los tokens de autenticación, el ID token, el access token y, opcionalmente el refresh token, para ser almacenados por el cliente, finalizando así el proceso de autenticación.

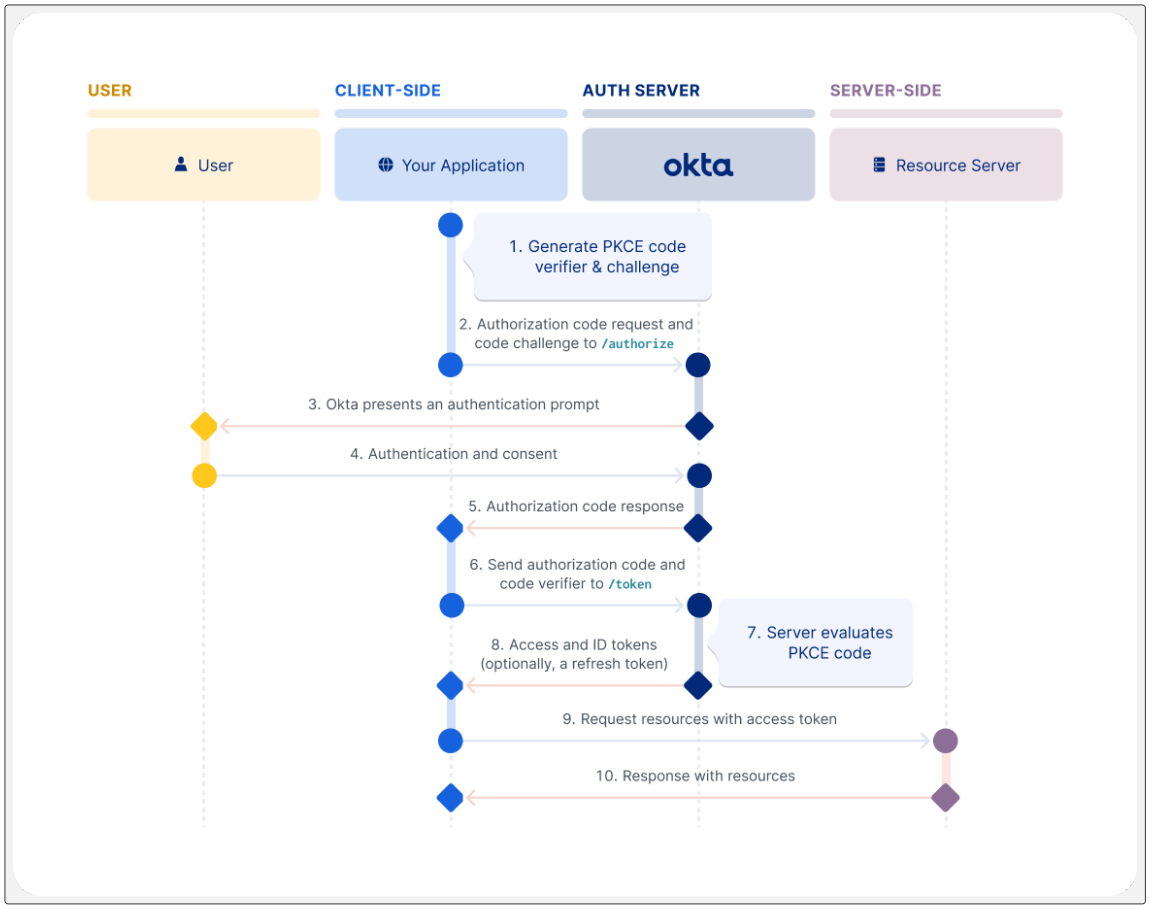

Figura 3.3: Protocolo OIDC - Authorization Code Flow con PKCE presentado por Okta [19]

3.2. Single Sign-on en el código

Después de identificar el protocolo y comprender su flujo, la siguiente etapa en este análisis implicó examinar el código del cliente. Esto se debió a que ya teníamos una comprensión más clara de cómo el sistema gestionaba la autenticación de los usuarios en la aplicación web.

3.2.1. Librería SSO

En este análisis se observó que la mayor parte de la funcionalidad de la aplicación se basaba en la utilización de la librería **okta-react** [20]. Según la documentación del repositorio oficial [21], esta librería es una SDK (Software Development Kit) que ofrece un conjunto de herramientas diseñadas para simplificar la integración con Okta en una aplicación frontend React. La documentación lista las facilidades que esta librería provee:

- Login y Logout de Okta usando la API de OAuth 2.0

- Recuperar información del usuario autenticado
- Determinar el estado de la autenticación
- Validar la sesión actual del usuario
- Protección de recursos web utilizando componentes de rutas seguras
- Protección de recursos web utilizando componentes de rutas seguras
- Posibilidad de personalizar el comportamiento en el proceso de autenticación
- Compartir el estado de Okta a los componentes de la aplicación, mediante un Hook de React [22] o utilizando componentes de orden superior [23]

Otra información importante que comparte la librería, es que soporta los siguientes flujos de autenticación.

- Implicit Flow
- Authorization Code Flow con PKCE

Debido a que la librería era mantenida por el equipo oficial del proveedor de autenticación y, debido a las facilidades antes mencionadas que provee la librería, el equipo decidió mantener la utilización de la misma pero llevando a cabo un proceso de refactorización el cual será detallado en el capítulo siguiente.

Capítulo 4

Identificación de problemas y refactorización

4.1. Actualización de okta-react

El primer desafío que se identificó, es que la aplicación estaba utilizando una versión muy antigua de la librería okta-react, en este caso la versión 2.0, cuando las versiones consideradas como estables en para la fecha eran 5.0 y 6.0 [21]. Antes de realizar una refactorización, se evaluó la actualización de la versión por las siguientes razones:

- **Seguridad**. La librería había tenido múltiples cambios y la versión utilizada no tenía más actualizaciones.
- **Documentación actualizada y facilidad para su uso**. La dificultad para encontrar documentación relevante para la versión utilizada motivó a avanzar de versión, para contar con información más actualizada y facilitar la utilización de la librería.
- **Soporte ante problemas o errores**. Dado que la versión utilizada estaba deprecada, actualizarla permitiría tener soporte ante posibles problemas o errores.

El siguiente paso fue determinar a que versión llevar la librería. Se consideraron dos opciones: una actualización gradual, siguiendo la documentación de migración entre versiones, o una actualización directa a la última versión, lo que requería reimplementar funcionalidades basándose en la documentación más reciente. La decisión final fue la actualización directa a la última versión (6.0), motivada por los siguientes factores:

- **Eficiencia y tiempo de desarrollo**. La actualización gradual hubiera implicado un gran esfuerzo y mayor tiempo de desarrollo debido al número alto de versiones intermedias.

- **Evitar problemas entre versiones**. La posibilidad de actualizar a la última versión, sin enfrentarse a posibles problemas entre versiones que peligrarían llegar a las estables.
- **Confianza en la implementación**. El haber tenido una investigación previa del flujo de autenticación generó confianza de no perder la funcionalidad previa y posibilitaba borrar el código necesario para empezar desde una base más sólida de trabajo.
- **Mejor soporte**. Más allá de que la librería tenía su propia guía de migración, los posibles problemas que se podían encontrar en la última versión se podrían resolver gracias a la documentación y a la comunidad más activa.

4.2. Sistema de ruteo

Una vez que la versión de la librería fue actualizada se comenzó identificando las secciones del código en donde se hacía uso de la misma. De esta manera, se identificó que el componente Router hacía uso de la librería y por lo que fue el primer componente por el cual se comenzó la reimplementación.

Antes de adentrarse en el código y su estructura, es importante destacar por qué es crucial modificar el router o enrutador del sistema, ya que es una parte fundamental de la aplicación debido a que gestiona la navegación entre las diferentes vistas o páginas. Al modificar la implementación de Okta, es necesario ajustar el Router para que pueda manejar correctamente las redirecciones y el acceso a las distintas partes de la aplicación según el estado de autenticación del usuario. De esta manera, el Router jugará un papel clave en garantizar una experiencia de usuario fluida y segura en el sistema.

Al revisar esta implementación, se encontró con un código similar al que se presenta a continuación. Aunque existen más rutas en el original, se han resumido para facilitar su presentación:

```
1  function Router() {
2    if (OKTA_ENABLED) {
3      const rootBoot = cookies.get('rootBoot');
4
5      if ((window.location.pathname === '' || window.location.pathname === '/')
         && rootBoot === undefined) {
6
7        window.location.href = ROUTES.DASHBOARD;
8        cookies.set('rootBoot', 'true', { path: '/', maxAge: 5 });
9      }
10
11     return (
12       <Security
13         oktaAuth={oktaAuth}
14         restoreOriginalUri={restoreOriginalUri}
15       >
16         <Switch>
17           <SecureRoute path={ROUTES.DASHBOARD} component={Dashboard} />
18           <Route path={ROUTES.LOGIN_ERROR} component={LoginError} />
19           <Route path={ROUTES.CALLBACK} component={ExplicitCallback} />
20         <Switch>
21       </Security>
22     );
23 }
24   return (
25     <Switch>
26       <PrivateRoute path={ROUTES.DASHBOARD} component={Dashboard} />
27       <Route path={ROUTES.LOGIN_ERROR} component={LoginError} />
28       <Route path={ROUTES.LOGIN} component={LoginPage} />
29       <Route path="/" component={LoginPage}>
30         <Redirect to={ROUTES.LOGIN} />
31       </Route>
32     </Switch>
33   );
34 }
```

Bloque de código 4.1: Implementación inicial del componente Router

Como se puede ver en el código, el componente Router define las rutas de la aplicación dependiendo del tipo de mecanismo de autenticación utilizado por el subdominio (Ver apartado 2.2.1). Es decir, si Okta está habilitado para el dominio por el cual el usuario ingresó, el componente renderiza el primer set de rutas (Lineas 2 a 22). En cambio, si el usuario no ingresó a un subdominio que utilice Okta, se renderizan otro sets de componentes por los cuales se declaraban las rutas, para la autenticación mediante cookies (Lineas 24 a 33).

De esta implementación se identificaron dos grandes problemas: Al momento de declarar las rutas del sistema, una gran parte del código se encuentra duplicado (Lineas 12 a 32) y, por otro lado, se define una redirección cuando se accede mediante Okta (Lineas 3 a 10). Ambos problemas derivan de la forma en que el código fue diseñado. Esta solución se pensó de una forma imperativa, es decir, el componente fue implementado como una secuencia de pasos a realizar, lo que conlleva a la perdida de una de las características fundamentales de la programación en React, en donde se busca un enfoque declarativo buscando que los componentes cumplan las diferentes funciones en base a eventos o cambios de estado. Por lo que el siguiente objetivo que se tomó fue diseñar un componente Router simple con un enfoque declarativo y que tenga como única responsabilidad retornar un solo conjunto de rutas, delegando el resto de las responsabilidades a otros componentes especializados.

4.2.1. Código duplicado y solución

Como se mencionó anteriormente, el componente Router se comporta de forma diferente dependiendo del tipo de autenticación elegido para el subdominio, generando así rutas repetidas. Esto se hace difícil mantener, ya que requiere duplicar las rutas de la aplicación para cada tipo de autenticación, característica que lleva a posibles errores por parte de los desarrolladores y a código poco escalable.

La razón por la que podría haberse tomado esta decisión de implementación es que los componentes de ruteo que provee la librería okta-react, como Security o SecureRoute, solo funcionan para un subdominio que utilize Okta, ya que requieren de los datos de configuración del proveedor de Okta. Por lo que, para los subdominios en los cuales se utiliza la autenticación mediante el servicio interno, esta configuración no existe y la aplicación podría fallar. Es por eso que se utilizan otros componentes diferentes de ruteo obtenidos de la librería react-router, los cuales son, Switch, Route y PrivateRoute, siendo este último una implementación personalizada basada en el componente Route.

La solución que se tomó para abordar el problema del código duplicado fue dividir la lógica de los distintos tipos de autenticación, siguiendo los principios SOLID [24]. Estos principios nacen de la programación orientada a objetos pero se pueden aplicar también a la programación en React [25] y se aplicó a lo largo de toda la solución, principalmente implementando el primer principio llamado Single Responsibility Principle (Principio de responsabilidad única), en el cual una clase, o componente debe tener únicamente una única responsabilidad. Este enfoque nos permitió una estructura de código más modular y mantenible, facilitando la gestión de los diferentes flujos de autenticación y promoviendo una mayor reutilización de código, al mejorar la legibilidad del mismo.

Volviendo al ejemplo de la implementación inicial del componente Router, como se mencionó anteriormente, este mantiene múltiples responsabilidades ya que no solo actúa como

enrutador del sistema, sino que también tiene diferentes comportamientos según el mecanismo de autenticación a utilizar, rompiendo así el primer principio de SOLID. Para solucionar este problema, se buscó generar nuevos conjuntos de componentes que cumplan responsabilidades simples y únicas para el tipo de autenticación al que corresponden, dividiendo el código en componentes más pequeños y con responsabilidades específicas, los cuales serán vistas a continuación.

4.2.2. Componentes contenedores de rutas

Con la objetivo de separar la lógica de los diferentes tipos de autenticación que el sistema proveía, se creó un nuevo componente llamado **SecurityWrapper**. Este componente actúa como único contenedor de las diversas rutas que posee el sistema y cumple la función de almacenar en el contexto el identificador del proveedor de autenticación activo (según el subdominio por el que el usuario ingresa a la aplicación). De esta forma, se buscó compartir la información del proveedor con todas las rutas predecesoras a través del contexto de React, de manera que se mantenga en comunicación cuál es el mecanismo de autenticación activo en todo el sistema, y que exista una única fuente de información.

Se vio conveniente utilizar el contexto para esta solución debido a que este tipo de información era necesario tenerlo compartido a nivel global en el sistema. Por lo que, si se hubiera implementado mediante el pasaje de propiedades entre los componentes, esto hubiera acomplejizado la solución trayendo la posibilidad de errores, al transmitir esta información entre grandes cantidades de componentes en el árbol del DOM. Este problema es conocido como prop-drilling [26].

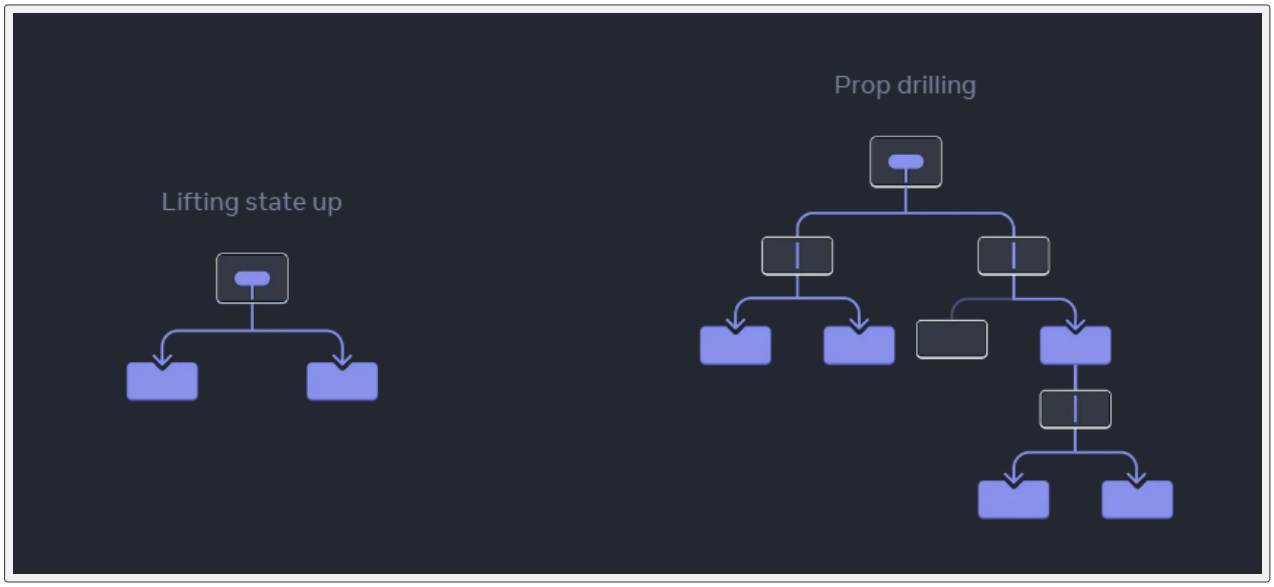

Figura 4.1: El problema de prop-drilling según la documentación oficial de React [26]

De esta forma, al utilizar la herramienta de estado global, Context de React, nos permitió transportar esta información directamente al componente que lo necesite, simplemente haciendo uso del contexto y así evitando posibles problemas de pasajes de props entre los diferentes componentes del árbol.

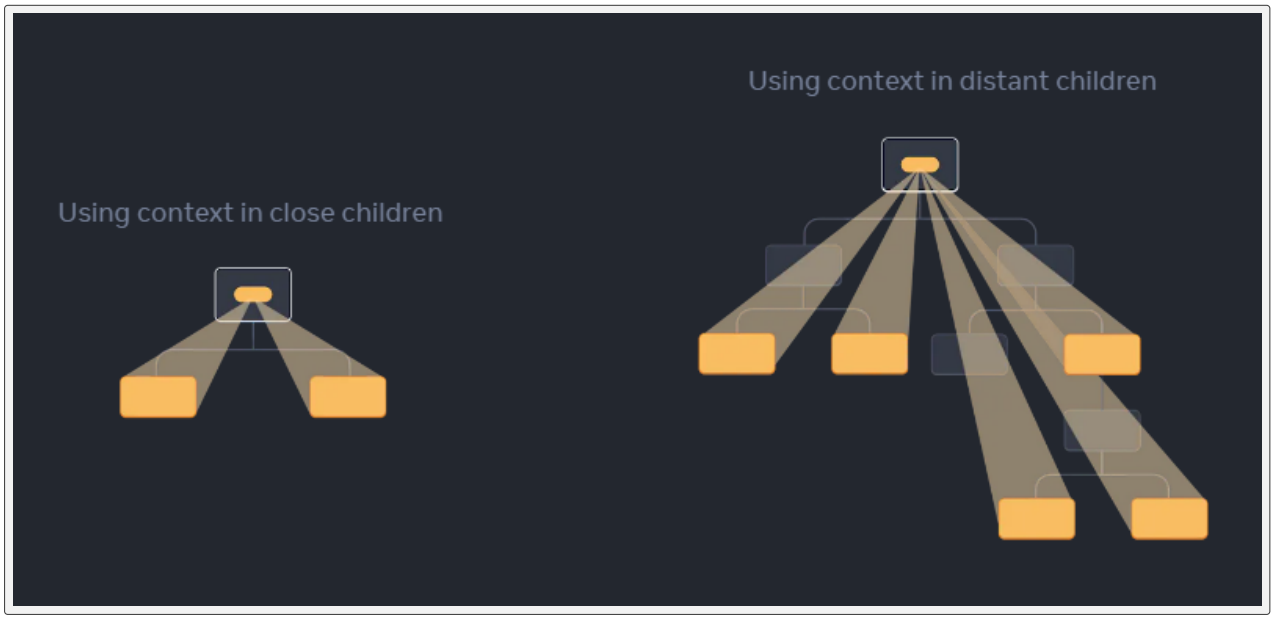

Figura 4.2: Utilizando Context para solucionar el prop-drilling según la documentación oficial de React [26]

Por último, más allá de compartir el identificador del proveedor activo, era necesario que nuestro componente implementado pueda proveer toda la información de contexto del proveedor de autenticación activo a todas las rutas del sistema. Para realizar esto, fue necesario definir nuevos componentes que inicialicen cada proveedor con su configuración correspondiente, y de esta forma, renderizarlos condicionalmente según el estado del proveedor activo que mantiene el componente SecurityWrapper. Siendo así, un pseudo-código de la implementación que se llevó a cabo:

```
1  import { Fragment, createContext, useState } from "react";
2  import OktaWrapper from "./Okta/OktaWrapper";
3
4  export const SecurityWrapperContext = createContext();
5
6  export const authProviders = {
7    OKTA: "okta",
8    INTERNAL: "internal",
9  };
10
11 const RouterWrappers = {
12   [authProviders.OKTA]: OktaWrapper,
13   [authProviders.INTERNAL]: Fragment,
14 };
15
16 const provider = Utility.getActiveProvider();
17 const RouterWrapper = RouterWrappers[provider];
18
19 function SecurityWrapper({ children }) {
20   return (
21     <SecurityWrapperContext.Provider value={{ provider }}>
22       <RouterWrapper>
23         {children}
24       </RouterWrapper>
25     </SecurityWrapperContext.Provider>
26   );
27 };
28
29 export default SecurityWrapper;
```

Bloque de código 4.2: Implementación del componente SecurityWrapper

Como se puede ver desde las lineas 6 a la 9, se definen los diferentes tipos de proveedores que el sistema soporta, asignándole a cada uno, un identificador único en forma de cadena de caracteres. Por lo que se definió un identificador para el proveedor de Okta y otro para la autenticación mediante el servicio interno de la aplicación.

En la línea 16, se utiliza una utilidad que implementa un algoritmo sencillo para verificar el subdominio desde el cual se accede a la aplicación web. Este algoritmo mapea el subdominio con el identificador del proveedor correspondiente, permitiendo así obtener el identificador del proveedor activo en el sistema. Cabe destacar que no se consideró necesario incluir la demostración del código de este algoritmo, dado que hace uso de funcionalidades básicas proporcionadas por JavaScript en el navegador, las cuales exceden el alcance de esta tesina.

Una vez que se obtuvo el identificador del proveedor de autenticación activo, el sistema utiliza esta información para elegir el componente contenedor de rutas a utilizar (Linea 17 del código). De esta forma, si Okta está habilitado para el sistema, se utilizará un componente OktaWrapper que contiene la configuración de este proveedor, y que provee todo el contexto necesario para utilizar Okta. Caso contrario, cuando el mecanismo de autenticación es mediante el servicio interno, es decir, mediante autenticación por cookies, se utilizará Fragment de React, el cual no necesita configuración y no tiene comportamiento alguno más que contener las rutas.

Un ejemplo de implementación para el componente OktaWrapper sería:

```
1  const oktaAuth = new OktaAuth({
2    issuer: okta_config.issuer,
3    clientId: okta_config.clientId,
4    redirectUri: `${window.location.origin}${ROUTES.CALLBACK}`,
5  });
6
7  function OktaWrapper({ children }) {
8    const history = useHistory();
9
10   const restoreOriginalUri = (_oktaAuth, originalUri) =>
11     history.replace(toRelativeUrl(originalUri || '/', window.location.origin))
12
13   return (
14     <Security
15       oktaAuth={oktaAuth}
16       restoreOriginalUri={restoreOriginalUri}
17     >
18       {children}
19     </Security>
20   );
21 }
```

Bloque de código 4.3: Implementación del componente OktaWrapper

4.2.3. Componentes de rutas

Como se vió en la implementación del componente Router, la aplicación tiene dos tipos diferentes de rutas. Las rutas públicas y las rutas privadas. La idea de estas es que, cuando un usuario intenta acceder a una ruta privada de la aplicación, este requiere haber sido autenticado previamente para lograr a acceder al componente el cual la ruta renderiza. En cambio, para las rutas publicas el usuario puede acceder libremente sin necesidad de haber iniciado sesión en el sistema.

Para implementar las rutas públicas, la librería okta-react tiene compatibilidad con la librería react-router [21], por lo que se pueden usar libremente los componentes Route importados de react-router para implementar este tipo de rutas.

En el caso de las rutas privadas, react-router no provee solución a esto ya que esta librería no está pensada para manejar información de autenticación y protección de recursos. Por lo que se ve necesario implementar componentes de rutas personalizadas que verifiquen si el usuario está logueado para mostrar el componente. En este caso, el sistema ya tiene implementado la llamada PrivateRoute, la cual hace uso del componente Route y verifica si un usuario está autenticado en el sistema mediante el mecanismo de sesión por cookies.

Por otro lado, la librería okta-react facilita un componente llamado SecureRote, el cual permite proteger las rutas verificando si existe un usuario autenticado mediante Okta. Es importante señalar que este componente requiere ser un componente hijo del contenedor de rutas antes mencionado Security, también importado de la librería okta-react, ya que requiere utilizar el contexto del proveedor de Okta para poder verificar la información de sesión del usuario autenticado.

La idea entonces fue reimplementar el componente llamado **PrivateRoute**, de manera que sea el único componente necesario para cumplir la funcionalidad de proteger las rutas de la aplicación. Este nueva implementación utiliza SecureRoute, en caso de que Okta se encuentre habilitado, o verificando la sesión mediante la cookie, si el sistema está autenticado mediante el servicio interno. Un pseudo-codigo de la implementación realizada sería:

```
1  function PrivateRoute({ component: Component, ...rest }) {
2    const { provider } = useContext(SecurityWrapperContext);
3
4    if (provider === authProviders.OKTA) return <SecureRoute component={Component
       } {...rest} />;
5
6    const isUserInternalAuthenticated = getInternalUserSession();
7
8    return (
9      <Route
10       {...rest}
11       render={
12         props => isUserInternalAuthenticated ? (
13           <Component {...props} />
14         ) : (
15           <Redirect to={{ pathname: ROUTES.LOGIN }} />
16         )
17       }
18     />
19   );
20 }
```

Bloque de código 4.4: Implementación del componente PrivateRoute

Como se puede ver en la linea 2 de la implementación de PrivateRoute, se hace uso del estado SecurityWrapperContext, el cual es inicializado en el componente SecurityWrapper mencionado anteriormente. La idea de esto es que los componentes se mantengan sincronizados con el mismo estado, de manera de que si la aplicación cambia el proveedor de autenticación, todos los componentes que utilicen este estado también sean comunicados y reaccionen por estos cambios.

4.2.4. Redirección en el Router y solución

Otro de los problemas que se identificaron del componente Router es que, únicamente cuando se utilizaba la autenticación por Okta, se hacía una redirección a una ruta específica llamada Dashboard. Este problema es debido a que la ruta por defecto por la que se accede a la aplicación cuando no está configurado Okta en el sistema, es la pantalla de Login. Por el contrario, cuando Okta se encuentra configurado, no existe una pantalla de Login propia del sistema, por lo que la aplicación redirige al usuario a la ruta privada de Dashboard, mostrando el formulario de Login del proveedor, en caso de que el usuario no haya sido autenticado previamente, gracias al comportamiento de SecureRoute de okta-react, el cual redirige si no encuentra una sesión activa en el sistema. Este tipo de implementación lleva a

los siguientes problemas:

- El componente Router no debería modificar el flujo de navegación del usuario, ya que la idea del mismo es declarar las rutas por las que el usuario puede navegar. Esto hace que el componente sea más complejo e impacta en la legibilidad del código.
- Mantener un flujo de navegación diferente entre mecanismos de autenticación, por parte del usuario en la aplicación, complejiza el entendimiento y mantenimiento del código.
- Esta implementación utiliza una redirección mediante la variable global window [27], quedando fuera del ecosistema de React y React Router. Esto puede generar posibles errores en la ejecución, siendo difíciles de detectar y solucionar.

La solución que se optó para este problema es remover la redirección implicada del Router y, teniendo noción de que, en caso de necesitar una redirección en la navegación, se utilizarán las herramientas que provee la librería react-router. La solución completa que se desarrolló para el flujo de autenticación y navegación del usuario será detallada en la sección siguiente.

4.2.5. Implementación final del sistema de ruteo

Una vez que se identificaron los problemas y se plantearon sus soluciones, el último paso fue volver a implementar el componente Router, quedando de la siguiente forma:

```
1  function Router() {
2    return (
3      <SecurityWrapper>
4        <Switch>
5          <PrivateRoute path={ROUTES.DASHBOARD} component={Dashboard} />
6          <Route path="/" component={Home} />
7        </Switch>
8      </SecurityWrapper>
9    );
10 }
```

Bloque de código 4.5: Implementación final del componente Router

Se puede comprobar que la cantidad de lineas de código se redujo drásticamente ganando además legibilidad, ya que el componente pasó a tener una única responsabilidad, declarar las rutas que el sistema posee y por las cuales el usuario puede navegar. A simple vista, también, se le otorgó un significado único a las rutas privadas y facilitando al desarrollador al momento de identificar que función cumplen las mismas, sin la necesidad de analizar su implementación.

4.3. Flujo de autenticación y navegación del usuario

Después de actualizar la librería y reestructurar el sistema de rutas de la aplicación, se consideró esencial diseñar un flujo coherente para la autenticación y navegación del usuario. Esto fue necesario ya que se identificó que el código tenía flujo difícil de seguir, donde las redirecciones estaban dispersas entre varios componentes que asumían múltiples responsabilidades, resultando en una estructura poco escalable. Esta complejidad tenía un impacto negativo en la legibilidad de la solución, volviéndola propensa a la generación de errores difíciles de identificar y corregir. La implementación de un nuevo flujo no solo busca mejorar la legibilidad sino también hacer que la solución sea más robusta y fácil de mantener.

4.3.1. Flujo de autenticación inicial

Para realizar el diseño de la solución, primero se analizaron los flujos por los cuales la aplicación frontend realizaba la autenticación en los diferentes métodos, a través del servicio interno o utilizando Okta.

Flujo de autenticación por sesión

Para la autenticación mediante el servicio interno, el sistema poseía un flujo bastante sencillo de seguir: Cuando el usuario ingresa al sistema por primera vez, es redirigido al componente Login, el cual posee el formulario para iniciar sesión mediante usuario/contraseña. Cuando el usuario ingresa los datos, el componente se encarga de validar los datos a través de una petición de tipo POST hacia el servicio interno de autenticación. Si los datos son correctos, el servicio responde con la cookie de la sesión y el componente de Login redirige al usuario al componente Dashboard, el cual es la pantalla principal de usuarios autenticados del sistema.

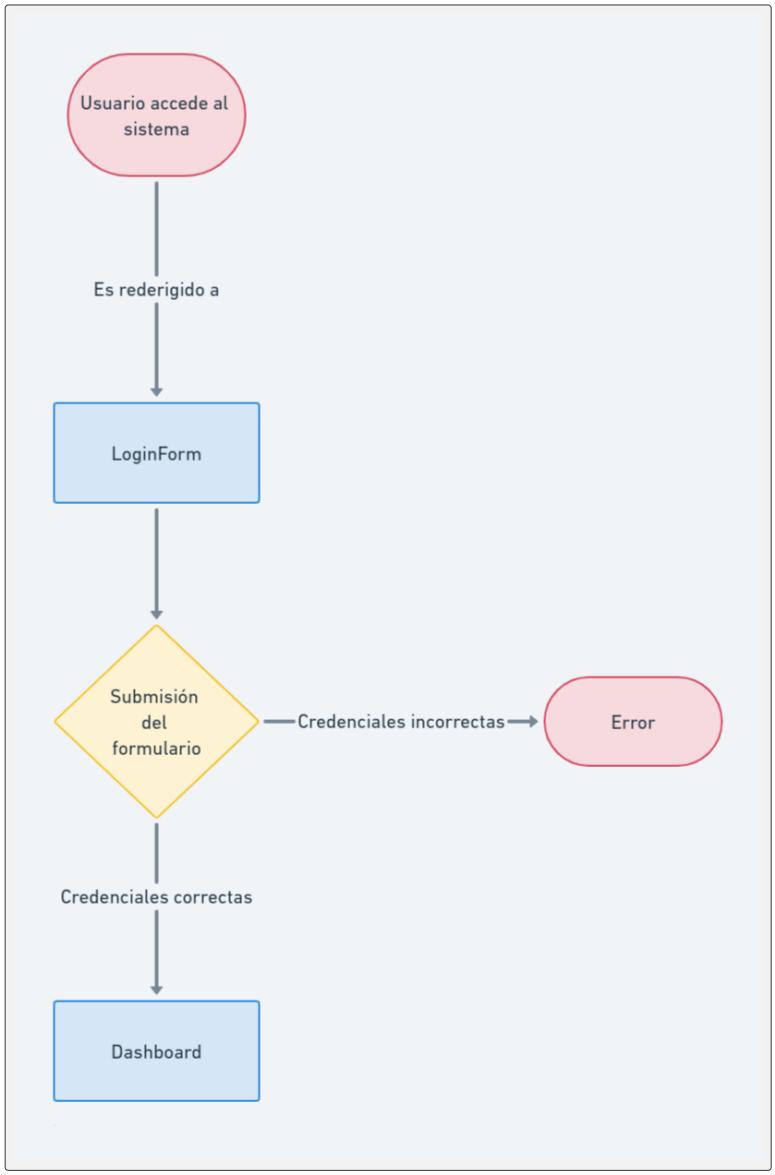

Figura 4.3: Flujo de autenticación mediante el servicio interno

Flujo de autenticación mediante Okta

Al analizar la autenticación mediante Okta (SSO), se encontró con que el sistema tenía un flujo muy diferente con respecto al anterior. Cuando el usuario ingresa al sistema, la aplicación redirige automáticamente a la ruta principal del mismo (Como se menciona en la sección 4.2.4). Como esta ruta es privada, gracias al comportamiento de este tipo de rutas, el sistema redirige al usuario a la pantalla de inicio de sesión del proveedor. Una vez autenticado, el proveedor redirige a la ruta de regreso configurada para el sistema. Esta ruta lleva al usuario a un componente llamado ExplicitCallback, el cual estaba encargado de

completar el proceso de login, redirigiendo al usuario a la pantalla de inicio del sistema, en caso de que este se autentique correctamente.

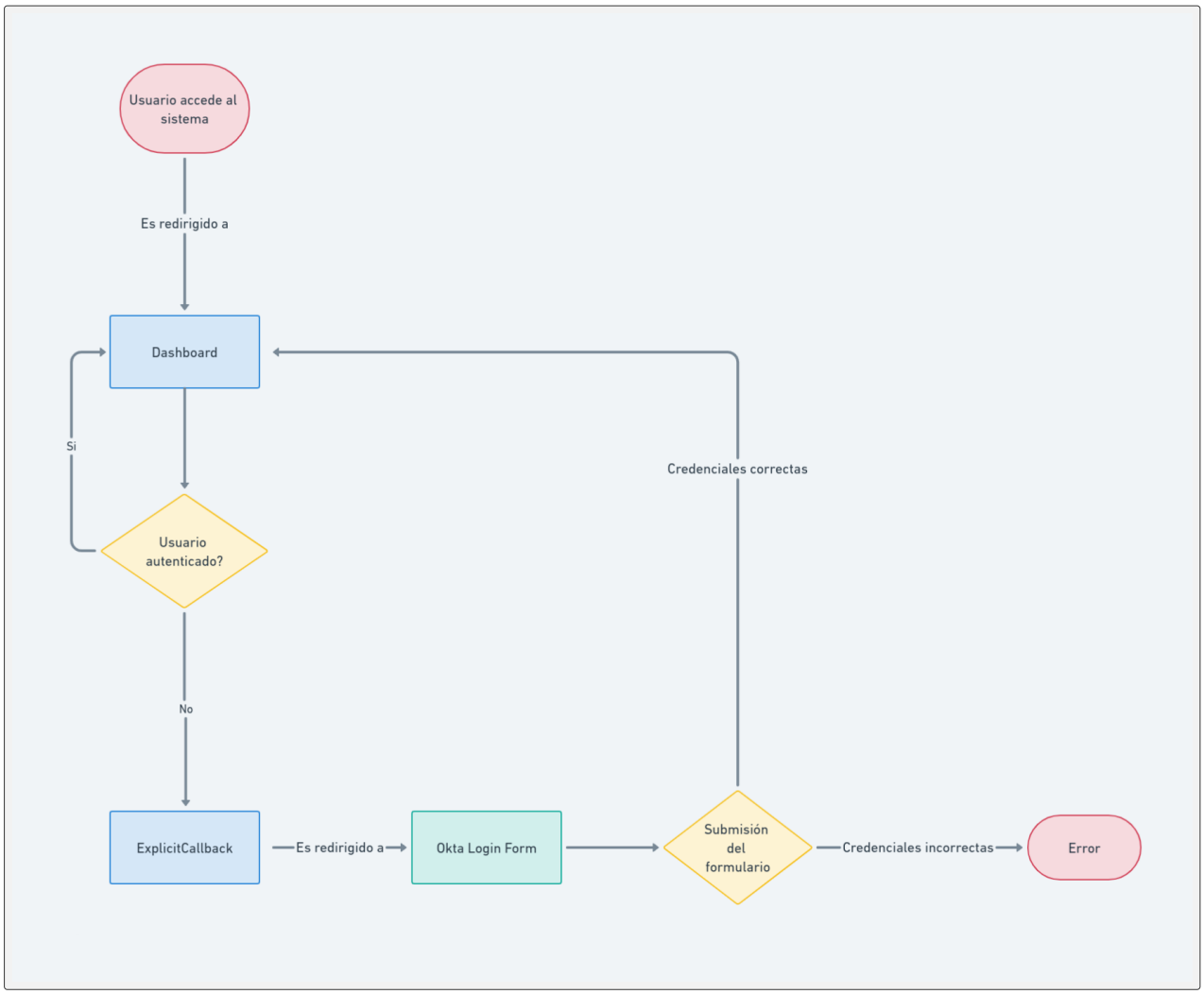

Figura 4.4: Flujo de autenticación mediante Okta

4.3.2. Reimplementación del inicio de la autenticación

Un problema que se identificó sobre estas implementaciones fueron que ambos tenían un flujo muy distinto para realizar la autenticación, desde que el usuario ingresa en la aplicación hasta que llega a la pantalla principal del sistema. A pesar de que sea normal que mecanismos diferentes tengan otros caminos para lograr la autenticación, la lógica estaba muy esparcida entre los diferentes componentes, generando una implementación poco escalable y dificultando la posibilidad de hacer modificaciones o mejoras. Ejemplo de este problema es que ambos mecanismos poseen un inicio diferente. En el caso de la autenticación por sesión,

el componente inicial por el que el usuario inicia en la aplicación es el formulario de Login y en el caso de la autenticación por Okta, es el componente de Dashboard el encargado de redirigir al usuario al formulario de inicio de sesión. Este tipo de problemas que se encontraban en implementación inicial surgían desde el planteamiento de soluciones imperativas, donde se buscó que la aplicación se comporte diferente ante ciertas condiciones, en este caso según el tipo de mecanismo de autenticación. De esta forma, el punto de arranque para la solución fue generar un componente Home el cual se encargue de definir cual sería el siguiente componente por el cual iniciar la autenticación. Este componente sería la ruta raíz de la aplicación y sería común para cualquier mecanismo de autenticación. Un ejemplo de la solución para este componente sería:

```
1  const authenticateRoutes = {
2    [authProviders.OKTA]: ROUTES.OKTA_AUTHENTICATE,
3    [authProviders.INTERNAL]: ROUTES.LOGIN,
4  }
5
6  function Home() {
7    const { provider } = useContext(SecurityWrapperContext);
8
9    const redirectPath = authenticateRoutes[provider];
10
11   return <Redirect to={redirectPath} />;
12 }
```

Bloque de código 4.6: Implementación del componente Home

La idea entonces fue que este componente elija la siguiente ruta por la cual el usuario lograría la autenticación, de acuerdo a la información proveniente del contexto. Cualquier componente siguiente sería totalmente independiente e implementaría el método de autenticación requerido por el usuario.

4.3.3. Reimplementación de auutenticación por Okta

Una vez planteado el inicio de ambos mecanismos, lo siguiente fue implementar un flujo para la autenticación mediante Okta ya que se identificaron diversos problemas en este flujo.

En primer parte, la iniciación de la autenticación estaba delegada en el comportamiento de la ruta privada al intentar acceder sin tener una sesión iniciada. Esto derivaba en código difícil de entender y de mantener ya que la redirección no estaba declarada explícitamente en el código. Por otra parte, el componente ExplicitCallback estaba implementado en base a una versión antigua de la librería okta-react, la cual había tenido cambios en su estructura que impactaban en el funcionamiento de los componentes generando errores.

Todos estos problemas fueron tenidos en cuenta a la hora de reimplementar el nuevo flujo. Algo que facilitó este proceso es que, al utilizar la última versión de la librería, la documentación estaba actualizada y resultó útil para plantear un flujo que vaya acorde a lo que la librería promueve como solución.

Comenzando por el inicio del flujo de autenticación, en lugar de utilizar el comportamiento de las rutas privadas, se utilizó una de las utilidades que la librería proveía. En este caso se utilizó el hook **useOktaAuth**, el cual retorna un objeto con el estado de Okta, mas ciertas funciones que permiten ejecutar cambios de estado en la misma, entre ellas, realizar el login del usuario. Para utilizar esta librería se necesito crear un nuevo componente, el cual haga uso de este hook, e inicie el proceso de login. Lo siguiente fue pensar el momento en el que se ejecute este comportamiento. Como la aplicación iniciaba el flujo de autenticación automáticamente cuando el usuario ingresaba a la aplicación, la idea fue pensar en un efecto por parte de este componente para que, cuando éste se monte en la aplicación, se inicie el proceso de autenticación. Finalmente, cuando la librería iniciaba el proceso de autenticación y el usuario complete este proceso, el estado de la librería cambiaría, almacenando el estado de la sesión en su estado, además del token JWT resultante de este proceso. Por lo que se utilizó esta información para redirigir al usuario hacia el Dashboard principal del sistema, una vez que completado el proceso de autenticación con Okta. Una primera idea de implementación para este componente sería:

```
1  function OktaAuthenticate() {
2    const { oktaAuth, authState } = useOktaAuth();
3
4    useEffect(() => {
5      if (authState && !authState.isAuthenticated) {
6        oktaAuth.signInWithRedirect();
7      }
8    }, [oktaAuth, authState]);
9
10   if (!authState) {
11     return <Loading />;
12   }
13
14   if (authState.error) {
15     return (
16       <Redirect
17           to={{
18           pathname: ROUTES.LOGIN_ERROR,
19           state: {
20             msg: 'Could not authenticate with OKTA. Please retry.',
21           },
22         }}
23       />
24     );
25   }
26
27   return authState.isAuthenticated && <Redirect to={ROUTES.DASHBOARD} />;
28 }
```

Bloque de código 4.7: Implementación del componente OktaAuthenticate

Como se puede ver en el código, el efecto que se definió en la linea 4, es el encargado de iniciar el proceso de autenticación del usuario mediante la función llamada **signInWithRedirect**. Tal y como se mencionó anteriormente, la idea era ejecutar el efecto al momento del montado del componente, pero en este caso esto no fue suficiente debido a que el usuario es redirigido hacia el formulario de inicio de sesión del proveedor para completar el formulario con sus credenciales, por lo que el componente es montado una vez más al ser devuelto de la página del proveedor. Debido a esto, se vio necesario también agregar la condición verificando si el usuario no fue autenticado al momento de montar el componente, para ejecutar el inicio del proceso de autenticación en caso de que no se realizó previamente.

Por último, al componente se le agregó comportamiento para manejar los cambios de estados, retornando un componente visual cuando la librería está cargando y redirigiendo a

una pantalla de error en caso de alguna falla en la autenticación.

4.3.4. Ruta de regreso o Callback

Siguiendo la documentación oficial de Okta [28] [29], la misma especificaba que se requería de una ruta que se encargara de continuar con el proceso de login en la aplicación frontend, una vez que el usuario es redirigido tras haber ingresado las credenciales en la pantalla de acceso del proveedor. Este proceso consiste en analizar los tokens que viajan en la uri, el componente los almacena, y luego redirige a la ruta por la que se inició la autenticación. Para cumplir con este proceso, la librería facilitaba un componente llamado **LoginCallback**, el cual al ser configurado en el sistema, permitía cumplir con este proceso.

Para configurar el componente LoginCallback en la aplicación, se creó la ruta por la cual será accedida a través de la uri /**callback**, agregandola en el componente Router. Luego fue necesario configurarla en el componente OktaWrapper con la propiedad **redirectUri**, especificado en la sección 4.2.2, la cual necesita coincidir con la ruta que se le asigna en el Router y por la que se accede al componente. Por último, fue necesario habilitar la url por la que el proveedor accederá, en este caso /callback, en la configuración de la aplicación del proveedor.

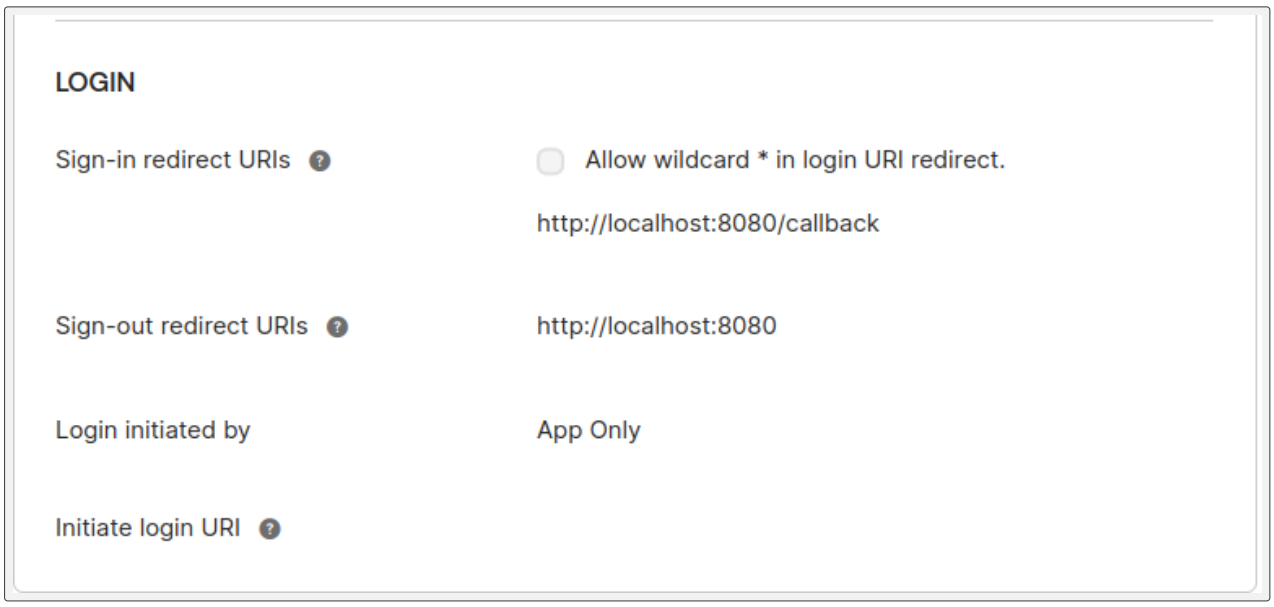

Figura 4.5: Panel de configuración de Okta para gestión de URIs

En este último ejemplo se muestra una configuración de Okta para el desarrollo local. Es importante remarcar que para el sistema productivo, al declarar la URI de redirección, se utilizó la dirección de dominio real del sistema. Esto es debido a que utilizar una dirección

de tipo localhost en un ambiente productivo puede ser inseguro, un atacante podría intentar explotar las configuraciones de desarrollo en un entorno de producción.

4.3.5. Flujos de autenticación finales

Después de implementar los flujos de autenticación para ambos mecanismos, se llevó a cabo una comparativa con el objetivo de establecer un diseño que sirva de guía para futuras implementaciones de otros mecanismos.

En el caso de la autenticación con servicio interno, no se necesitaron hacer muchos cambios en el proceso debido a que solo se necesitó agregar el componente Home al flujo. Este componente es el encargado de redirigir al usuario al inicio del proceso de autenticación correspondiente según el mecanismo definido para el subdominio (Mencionado en la sección 4.3.2), en este caso, hacia el formulario de login. Al completar la autenticación, el usuario es redirigido a la pantalla de Dashboard del sistema.

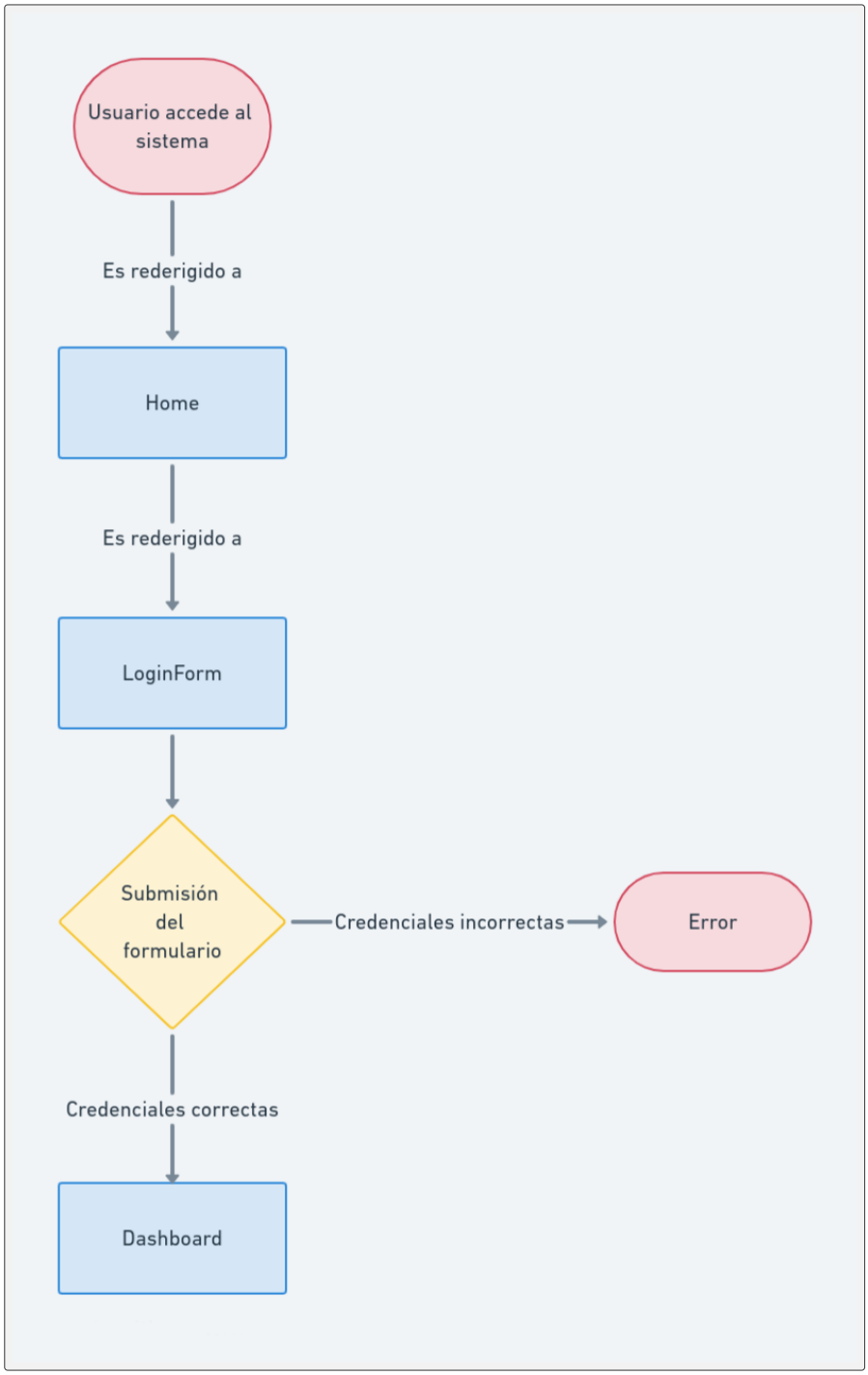

Figura 4.6: Diseño final del flujo de autenticación mediante el servicio interno

Por otro lado, en la autenticación utilizando Okta como proveedor, el inicio de este flujo es idéntico al anterior ya que el usuario comienza desde el componente Home, quien se encarga de redirigir al usuario componente encargado de iniciar la autenticación con el proveedor, el componente OktaAuthenticate. Este componente identifica si el usuario posee

una sesión iniciada en el sistema. Si el usuario no se encuentra autenticado, inicia el proceso de autenticación con el proveedor. En cambio, si el usuario ya posee una sesión previa al momento de montar el componente, este lo redirige hacia el componente Dashboard.

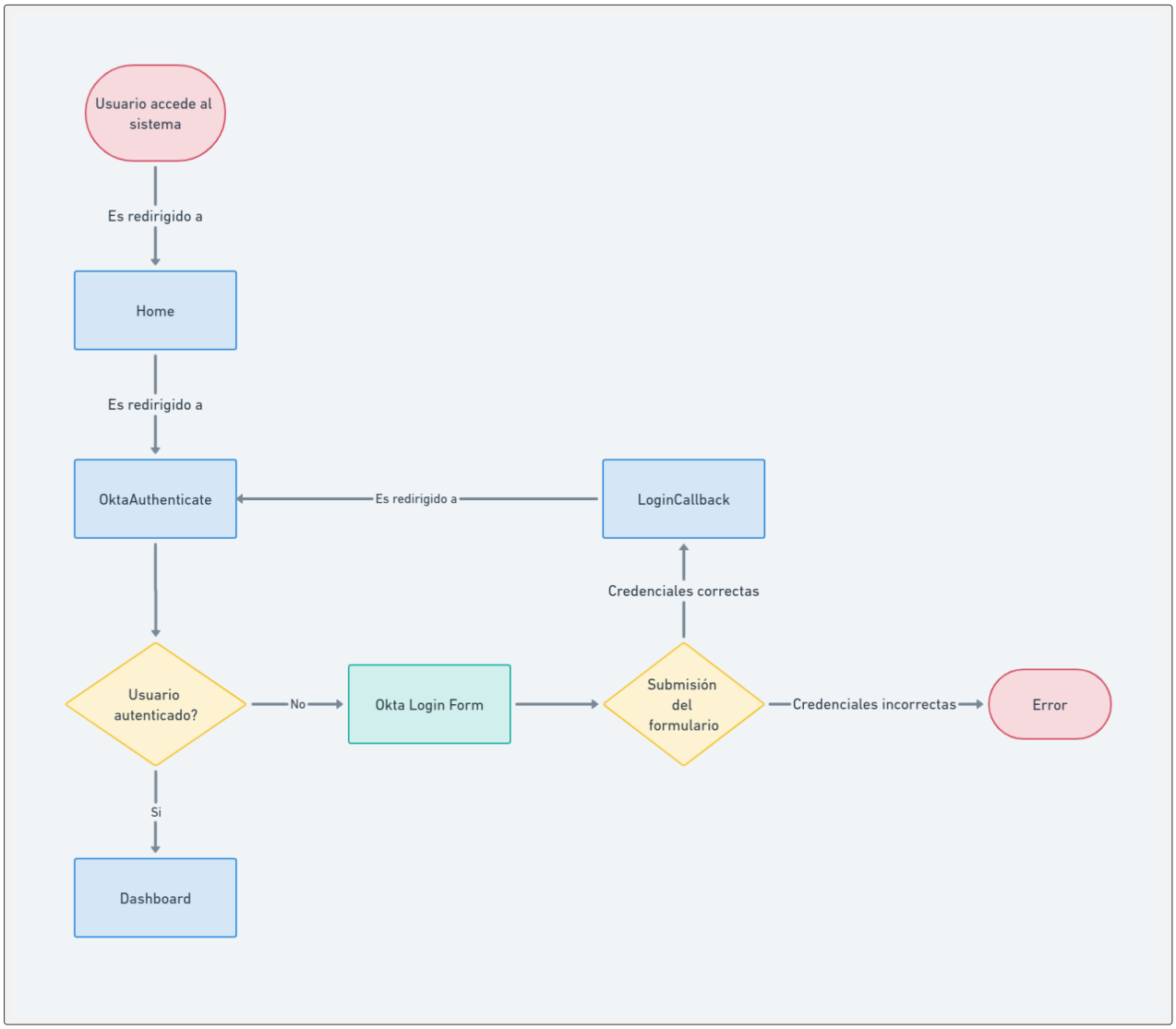

Figura 4.7: Diseño final del flujo de autenticación mediante Okta

Tomando ambos flujos se puede identificar que el inicio y el fin son idénticos. De esta forma, si agrupáramos todos los pasos intermedios entre los componentes de Home y Dashboard, y se los pensara como una caja negra la cual se encarga de autenticar al usuario, se puede tomar esto como un único módulo encargado de la autenticación de la aplicación, evitando repartir esta lógica en componentes fuera de este grupo. Siguiendo esta idea, se logró definir un flujo consistente para los diferentes mecanismos de autenticación y se dejó el camino marcado para futuras implementaciones de otros mecanismos o proveedores de

autenticación.

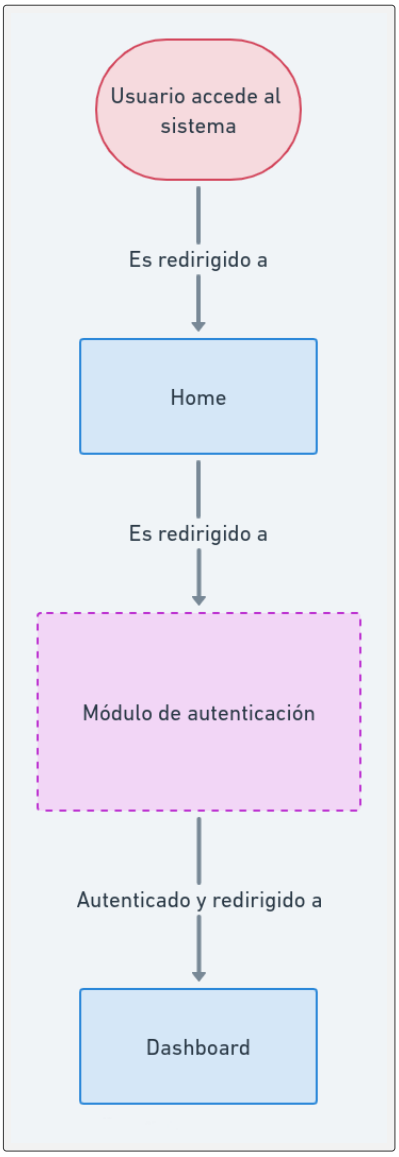

Figura 4.8: Diseño general del flujo de autenticación

4.4. Cierre de sesión

La siguiente tarea que se tomó fue la de analizar el proceso de cierre de sesión. En este proceso se encontró en el código un módulo javascript el cual era llamado por un evento de click en un elemento del DOM y su función era limpiar, en el cliente, el token de acceso de Okta, las cookies de sesión correspondientes y realizar una petición de finalización de sesión al servicio interno de autenticación para invalidar las cookies del lado del servidor.

Una vez realizada la limpieza de sesión, se redireccionaba al usuario de la pantalla de

acceso según el tipo de mecanismo de autenticación utilizado, es decir, al formulario de inicio de sesión de la aplicación en el caso de la autenticación mediante cookies, o a la pantalla de acceso del proveedor en caso de Okta.

4.4.1. Reimplementación del proceso

Durante la refactorización se realizaron varios cambios en este proceso. Estas modificaciones se centraron en el módulo encargado del cierre de sesión, y además, se introdujo un nuevo componente en este proceso que haría uso de dicho módulo.

El primer cambio en el módulo fue de borrar la lógica de redirección condicional según el tipo de mecanismo de autenticación, y, en su lugar, redirigir al componente Home mencionado en la sección 4.3.2. La lógica de redirección según el tipo de mecanismo de autenticación pasaría a estar a cargo únicamente de este nuevo componente Home, de manera que se simplifique el código del módulo y se facilite su futuro mantenimiento.

El segundo cambio que se realizó en este módulo fue el de remover la lógica para eliminar el token JWT en el cierre de sesión, de manera que el módulo se utilice únicamente para limpiar la sesión por cookies. La razón de este cambio fue debido a que se pensó en mantener este módulo únicamente para la finalización de sesión del servicio interno, sin manejar la lógica de Okta, ni de ningun proveedor externo de autenticación.

Algo importante mencionar en este punto es que este módulo no realizaba ninguna petición con el proveedor de Okta. En cambio, solo realizaba la limpieza de sesión del lado del cliente, de manera que nunca invalidaba la sesión del lado del servidor, dejando una carga innecesaria del lado del proveedor y un problema de seguridad al dejar token válidos, los cuales podían ser robados por más de que el usuario haya cerrado sesión. Este problema mencionado fue tenido en cuenta en la solución final mediante la implementación del componente que será explicado a continuación.

Finalmente, se creó un nuevo componente llamado Logout, del cual se agregó una ruta de acceso en el componente Router de manera que, cuando el usuario hace click en el elemento del DOM encargado del logout, la aplicación ejecuta una redirección a este nuevo componente.

La idea de la creación de este componente fue de separar los llamados a los módulos de cierre de sesión para los diferentes mecanismos y que sean ejecutados mediante la renderización de nuevos componentes granulares. La ventaja de esto es que se separa la lógica de cierre de sesión entre componentes especializados, los cuales se encargan únicamente del cierre de sesión del mecanismo al que pertenezcan, separando la lógica de cada proveedor y facilitando su futuro mantenimiento.

Una implementación para este componente sería:

```
1  import { useContext } from "react";
2  import { SecurityWrapperContext, authProviders } from "./SecurityWrapper";
3  import OktaLogout from "./Okta/OktaLogout";
4  import InternalLogout from "./Internal/InternalLogout";
5
6  const LogoutComponents = {
7    [authProviders.OKTA]: OktaLogout,
8    [authProviders.INTERNAL]: InternalLogout,
9  };
10
11 function Logout() {
12   const { provider } = useContext(SecurityWrapperContext);
13
14   const LogoutComponent = LogoutComponents[provider];
15
16   return <LogoutComponent />;
17 }
18
19 export default Logout;
```

Bloque de código 4.8: Implementación del componente Logout

Como se puede ver en el código anterior, este nuevo componente de Logout elige de acuerdo al estado del proveedor activo de autenticación, el componente hijo que posee la lógica especifica para realizar el cierre de sesión.

Por lo que fueron necesario definir otros dos componentes de los cuales Logout hace uso. Estos serían **OktaLogout** y **InternalLogout**.

```
1  import { useEffect } from "react";
2  import { useOktaAuth } from '@okta/okta-react';
3
4  function OktaLogout() {
5    const oktaAuthUtils = useOktaAuth();
6
7    useEffect(() => {
8      if (oktaAuthUtils) {
9        oktaAuthUtils.oktaAuth.signOut(ROUTES.HOME);
10     }
11
12   }, [oktaAuthUtils, provider]);
13
14   return <Loading />;
15 }
```

Bloque de código 4.9: Implementación del componente OktaLogout

El componente OktaLogout, hace uso de la librería okta-react para utilizar el hook useOktaAuth. Este hook posee la instancia de oktaAuth con sus métodos. En este caso, se invoca a la función signOut para realizar el proceso de cierre de sesión del proveedor, limpiando la sesión tanto en el cliente como en el servidor.

Para poder realizar el cierre de sesión del lado del servidor, esta función redirige al cliente hacia la dirección de la instancia del proveedor y, una vez ahí, ejecutar dicha petición. Cuando esta petición de cierre de sesión se completa, el proveedor redirige nuevamente a la aplicación.

```
1  import { useEffect } from "react";
2  import finishUserSession from '../utils/logout';
3
4  function InternalLogout() {
5    useEffect(async () => {
6      await finishUserSession();
7    }, []);
8
9    return <Loading />;
10 }
```

Bloque de código 4.10: Implementación del componente InternalLogout

Para el mecanismo de sesión por cookies, en la linea 6 del código anterior, se hace llamado al módulo con la lógica de la limpieza de cookies y el llamado a la api para finalizar la sesión. Este módulo también se encarga de redireccionar la aplicación al formulario de inicio de sesión, tras haber finalizado la petición.

Por último, es necesario destacar que ambos componentes de cierre de sesión renderizan una visual para la pantalla de carga, mientras se están ejecutando dichas la peticiones para la finalización de la sesión. De esta manera, se visualiza al usuario que la aplicación está terminando el proceso de cierre de sesión para ambos mecanismos.

Capítulo 5

Integración Azure Active Directory

En este punto del desarrollo, gracias a la refactorizacíon y actualización de la implementación preexistente, como equipo teníamos el conocimiento necesario para llevar a cabo una integración de Azure AD de una manera ágil debido a que ya teníamos identificados, tanto el protocolo de autenticación a seguir, como los componentes a implementar que el flujo requería. Por lo tanto, los pasos que se tomaron para la integración de este nuevo proveedor fueron buscando una similitud al diseño de la solución utilizada para integrar Okta, de manera que se mantenga una concordancia y que se acople fácilmente a la solución completa de autenticación.

5.1. Análisis Azure Active Directory

Previo a realizar la integración de este nuevo proveedor de autenticación, fue necesario realizar un análisis de la tecnología con el fin de facilitar su implementación.

Según la documentación oficial [30], Microsoft Azure Active Directory es un servicio que permite la administración de identidades y acceso basado en la nube, el cual permite a usuarios acceder a recursos externos. La documentación de este servicio especifica que permite a los desarrolladores utilizar esta tecnología como proveedor de autenticación basado en estándares, de manera que permita implementar Single Sign-On.

Otra información que identificamos de la documentación fue que la plataforma del servicio estaba compuesta por diversos componentes [31], de los cuales se nos hicieron útiles los siguientes:

- Servicio de autenticación compatible con los estándares OAuth 2.0 y OpenID Connect.
- Bibliotecas de código abierto: bibliotecas de autenticación de Microsoft (MSAL) y compatibilidad con cualquier otra biblioteca que cumpla con los estándares.

- Contenido para desarrolladores: documentación técnica que incluye inicios rápidos, tutoriales, guías paso a paso, referencia de API y ejemplos de código.

De esta forma podíamos analizar cada componente de manera que nos facilitara la implementación de este nuevo proveedor de autenticación.

5.2. Librería utilizada

Para implementar Azure AD como proveedor de autenticación SSO se pensó en buscar una librería que permitiera simplificar la integración. Esta librería debía cumplir con requisitos importantes, como respaldar el protocolo de autenticación Authorization Code flow with PKCE y, al mismo tiempo, gestionar la sesión del usuario y garantizar la protección de los recursos en la aplicación.

En esta búsqueda se identificó al paquete **MSAL.js** (Microsoft Authentication Library for JavaScript) [32] el cual forma parte de los componentes que conforman el servicio Azure AD [31]. Esta librería nos proveía una colección de paquetes pensados para la integración de las diferentes soluciones que otorga Microsoft en la autenticación, tanto para aplicaciones cliente como del lado del servidor.

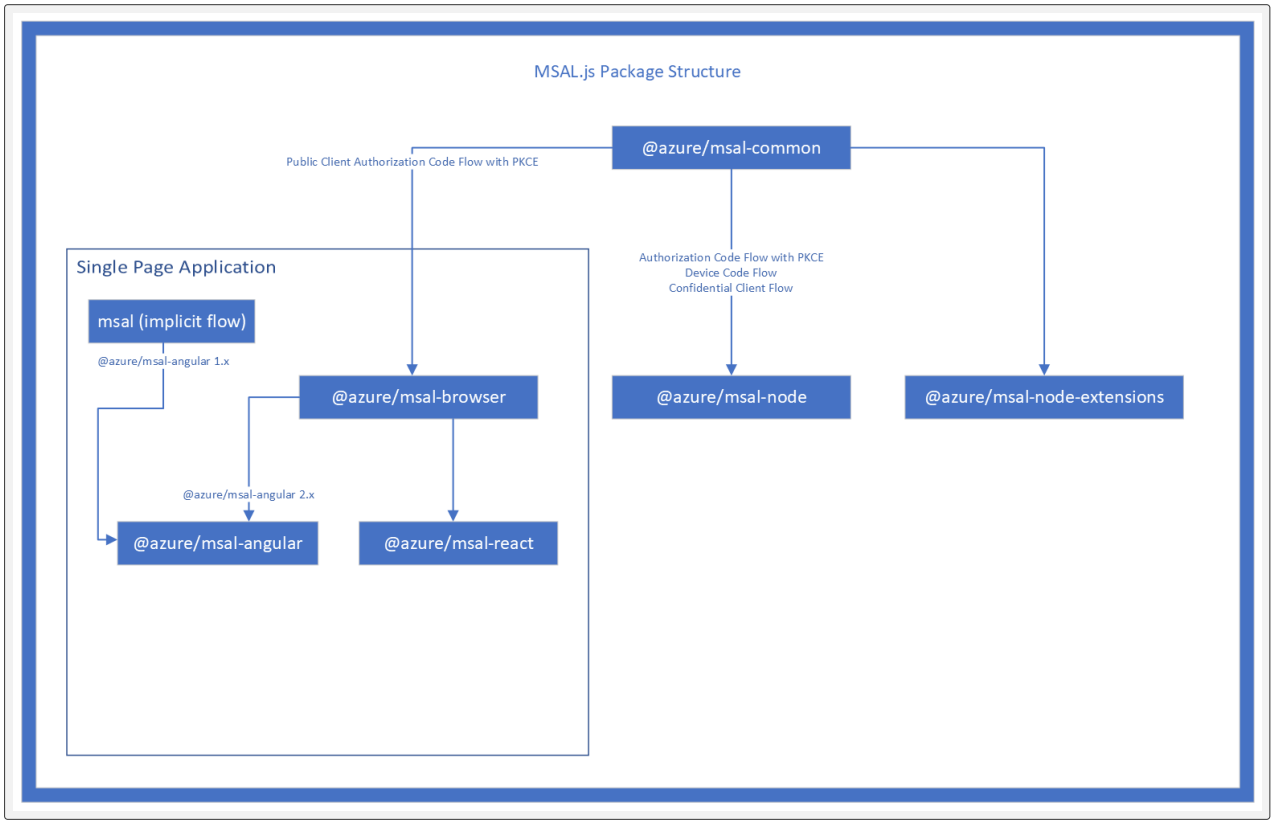

Figura 5.1: Estructura del paquete MSAL publicado por la documentación oficial [32]

El equipo optó por utilizar este conjunto de herramientas motivado por el hecho de que se encontraba publicado y mantenido por el equipo oficial de Microsoft, lo que generó confianza en términos de soporte frente a problemas.

Dentro de este conjunto de herramientas, decidimos incorporar dos librerías específicas en nuestro sistema: **msal-browser** y **msal-react**. La primera de ellas, msal-browser, establece las bases para implementar la autenticación en aplicaciones de página única (SPA), proporcionando funcionalidades esenciales para la comunicación con Azure AD. Por otro lado, msal-react ofrece componentes predefinidos para React, lo que simplifica considerablemente la integración de la autenticación en aplicaciones desarrolladas con esta tecnología. Ambas librerías eran de mucha ayuda ya que facilitaban la implementación de la autenticación de Azure AD en nuestro sistema, lo que nos permitía garantizar la seguridad y la confiabilidad del proceso de autenticación de los usuarios con el frontend.

5.3. Inicialización del proveedor

Siguiendo la documentación oficial [33], el primer paso que se tomó para proveer autenticación a la aplicación cliente fue de inicializar la librería con los parámetros de configuración del proveedor [34], los cuales permiten establecer la relación entre el proveedor Azure AD y la aplicación cliente. Estos parámetros son:

- **client-id** o Application ID. Este parámetro es el único requerido por la aplicación, el cual identifica a la aplicación dentro de la plataforma de Azure AD.
- **Authority**. Un parámetro opcional que posee la URL de la instancia del proveedor de autenticación, junto con el identificador de la aplicación del sistema, ambos contactenados.
- **Directory ID**. Parámetro opcional, el cual es utilizado para sistemas Single-tenant.
- **Redirect URI**. Parámetro opcional, utilizado para aplicaciones web, el cual especifica a donde el proovedor de identidad debe retornar los tokens de seguridad.

En esta implementación el cliente se encargó de registrar la aplicación en su instancia de Azure AD, por lo que se nos compartió los valores de los parámetros client-id y authority. También, fue necesario configurar un tercer parámetro opcional, el llamado llamado Redirect URI del cual se hizo un consenso para definir su valor, ya que este hace referencia a la dirección por la que se accede a la aplicación web tras haberse realizado la autenticación del usuario con el proveedor externo, para recibir los tokens generados. Por lo que este valor fue necesario tener en cuenta más adelante, al momento de implementar el flujo de autenticación, ya que este valor debía ser una ruta valida para la aplicación web.

5.3.1. Contexto del proveedor

Para garantizar que la aplicación permanezca sincronizada con los datos del proveedor de autenticación, es esencial que todos los componentes que lo requieran tengan acceso a esta información. Para lograr esto, se necesitó de un componente contenedor que sea responsable de mantener el contexto del proveedor y permita a los componentes predecesores acceder al mismo. La librería msal-react exporta un componente llamado **MsalProvider** la cual cumple este objetivo. De acuerdo a la documentación oficial[35], este componente recibe la instancia de msal configurada con los parámetros del proveedor previamente mencionados para inicializar la librería y compartir la instancia, mediante contexto, a los componentes predecesores.

Por lo tanto, la solución que se llevó a cabo fue la de crear un nuevo componente llamado **AzureWrapper**, el cual se encarga de inicializar la instancia de la librería y de renderizar el componente provisto llamado MsalProvider.

```
1  import React, { useEffect, useState } from "react";
2  import { MsalProvider } from "@azure/msal-react";
3  import { PublicClientApplication } from "@azure/msal-browser";
4
5  const msalConfiguration = {
6    auth: {
7        clientId: "client-id",
8        authority: "https://login.microsoftonline.com/common",
9        redirectUri: "/azureAuth"
10     }
11 };
12
13 const pca = new PublicClientApplication(msalConfiguration);
14
15 function AzureWrapper({ children }) {
16   const [msalInitialized, setMsalInitialized] = useState(false);
17
18   useEffect(() => {
19     const initializeMsal = async () => {
20       await pca.initialize();
21       setMsalInitialized(true)
22     };
23
24     initializeMsal();
25   }, []);
26
27   return (
28       <MsalProvider instance={pca}>
29         {msalInitialized && children}
30       </MsalProvider>
31   );
32 }
```

Bloque de código 5.1: Implementación del componente AzureWrapper

La finalidad de la creación de este componente fue la de integrarlo con el ecosistema implementado para los diferentes mecanismos de autenticación ya existentes en la aplicación, por lo que se modificó el componente SecurityWrapper tal y como se realizó con Okta, detallado en la sección 4.2.2. De esta manera, cuando el sistema identifica que el dominio utiliza Azure como proveedor de autenticación mediante la lógica implementada, el sistema utiliza este componente para proveer el estado y las utilidades de autenticación de Azure a los componentes predecesores.

```
import { Fragment, createContext, useState } from "react";
import OktaWrapper from "./Okta/OktaWrapper";
import AzureWrapper from "./Azure/AzureWrapper";

export const SecurityWrapperContext = createContext();

export const authProviders = {
  OKTA: "okta",
  AZURE: "azure",
  INTERNAL: "internal",
};

const RouterWrappers = {
  [authProviders.OKTA]: OktaWrapper,
  [authProviders.AZURE]: AzureWrapper,
  [authProviders.INTERNAL]: Fragment,
};

const provider = Utility.getActiveProvider();
const RouterWrapper = RouterWrappers[provider];

function SecurityWrapper({ children }) {
  return (
    <SecurityWrapperContext.Provider value={{ provider }}>
      <RouterWrapper>
        {children}
      </RouterWrapper>
    </SecurityWrapperContext.Provider>
  );
};
```

Bloque de código 5.2: Implementación del componente SecurityWrapper utilizando Azure

5.4. Flujo de autenticación

Al definir un flujo de autenticación, en primera instancia, se identificó el proceso que llevaba a cabo la librería y se definió un conjunto de componentes que pueda cumplir este proceso y que se acople fácilmente a lo ya implementado.

Para iniciar el proceso de autenticación [36], se requirió de un punto de entrada para el usuario, es decir, un componente que sea capaz de disparar el inicio de sesión y una ruta que permita al usuario acceder a este componente. Por lo que se definió un nuevo componente de Login para Azure, el cual utilizando las funcionalidades de la librería, permite iniciar con

el flujo de autenticación.

Para iniciar el proceso de autenticación la librería cuenta con diferentes métodos de login de los que se diferencian por la interacción del usuario [37]. De estos métodos se utilizó el inicio de sesión con redirección, en donde el usuario es redirigido al portal del proveedor para ingresar sus credenciales y luego, tras ser autenticado correctamente, redirigido nuevamente hacia la aplicación con una sesión iniciada.

Una implementación para el componente de AzureLogin podría ser:

```
1  import { InteractionStatus } from "@azure/msal-browser";
2  import { useIsAuthenticated, useMsal } from "@azure/msal-react";
3  import { useEffect } from "react";
4  import { Redirect } from "react-router-dom";
5
6  function AzureLogin() {
7    const { inProgress, instance } = useMsal();
8    const isAuthenticated = useIsAuthenticated();
9
10   useEffect(() => {
11     if (inProgress === InteractionStatus.None && !isAuthenticated) {
12       instance.loginRedirect();
13     }
14   }, [instance, isAuthenticated, inProgress]);
15
16   if (inProgress === InteractionStatus.Login) {
17     return <Loading />;
18   }
19
20   return isAuthenticated && <Redirect to="/dashboard" />;
21 };
```

Bloque de código 5.3: Implementación del componente AzureLogin

Como se puede ver en el código, la implementación requirió de dos hooks de la librería msal-react. Uno para acceder a la instancia del proveedor y al estado de la misma, y otro para saber si el usuario se encuentra autenticado o no.

La lógica de la implementación entonces fue: Cuando el componente es montado se identifica el estado de la sesión del usuario. Si la instancia se encuentra ociosa (inProgress === InteractionStatus.None) y el usuario no se encuentra autenticado, se hace el llamado al método de loginRedirect, el cual inicia con el proceso de autenticación, redirigiendo al usuario hacia el sitio del proveedor, como se comentó en el apartado anterior. Una vez que el usuario se haya autenticado satisfactoriamente, el proveedor redirige hacia la ruta de callback que se configuró al inicializar la librería, mencionado en la sección 5.3.1.

Por otro lado, si se encuentra en progreso el proceso de login, el componente retornará una pantalla de carga hasta que finalice. Y por último, si el usuario ya se encuentra autenticado con el sistema, el componente redirige hacia la pantalla de dashboard principal del sistema.

Una vez implementado el componente que da inicio al flujo de autenticación, solo restaba crear una nueva ruta de manera que el usuario logre acceder a este componente. Esta ruta fue configurada con el mismo valor utilizado para el parámetro callback al momento de inicializar el proveedor de Azure, visto en la sección 5.3.1. De esta manera, cuando el usuario realiza satisfactoriamente la autenticación, el proveedor redirige a la aplicación renderizando el mismo componente que inició el proceso de login, el cual posee la lógica para continuar a la pantalla de inicio del sistema, cuando el usuario está correctamente autenticado.

De esta forma, se logró generar un flujo de autenticación que se integra perfectamente con los métodos de autenticación preexistentes debido a que se respetó el flujo diseñado anteriormente.

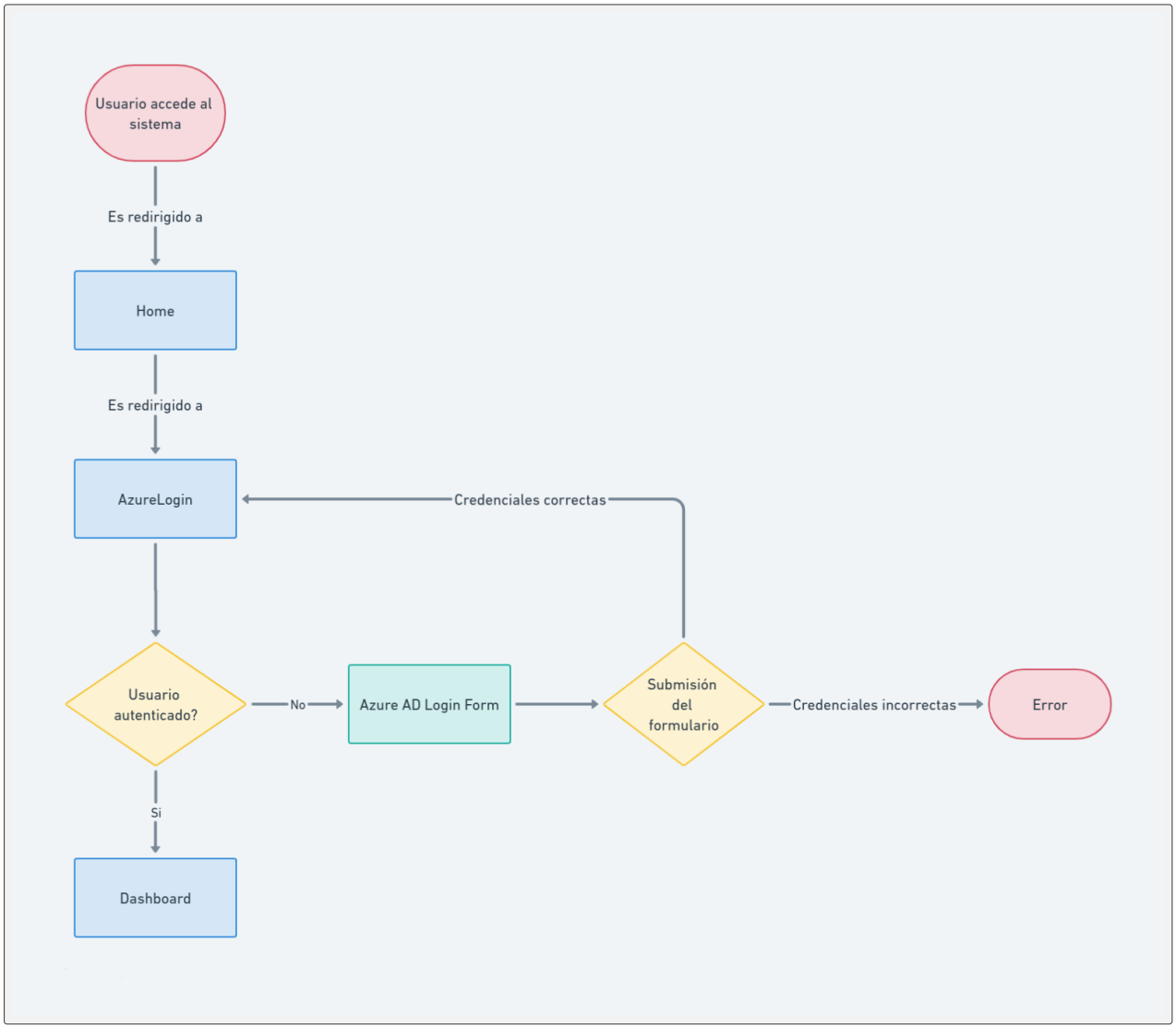

Figura 5.2: Diseño final del flujo de autenticación mediante Azure AD

5.5. Rutas privadas y protección de recursos

Una vez implementado el flujo de autenticación, lo siguiente que se llevó a cabo fue la creación del componente que permita la protección de los recursos mediante rutas privadas en la aplicación. Tal y como se implementó para Okta, se requirió de un nuevo set de componentes que utilicen el estado, en este caso de Azure, en el momento que un usuario requiere acceder a una ruta especifica de la aplicación para detectar si el usuario puede acceder a la misma o no.

A diferencia de Okta, Azure no facilitaba un componente de ruta privada que implemente esta funcionalidad para ser usada en el sistema, por lo que se requirió de crear este

componente utilizando otras utilidades que la librería proveía.

En este caso, se hizo uso del hook **useIsAuthenticated** provisto por msal-react, el cual devuelve en forma de booleano si el usuario se encuentra o no autenticado con el sistema.

De esta forma, se creó el componente **AzurePrivateRoute**, el cual actúa como componente de ruta y recibe como propiedad el componente a ser renderizado. Si el usuario se encuentra autenticado, renderiza el componente provisto o, caso contrario, redirige al usuario a una ruta especifica, en este caso a la ruta raíz del sistema.

```
1  import { useIsAuthenticated } from "@azure/msal-react";
2  import { Route, Redirect } from "react-router-dom";
3
4  function AzurePrivateRoute({ component: Component, ...rest }) {
5    const isAuthenticated = useIsAuthenticated();
6
7    return (
8      <Route
9        {...rest}
10       render={
11         props => isAuthenticated ? (
12           <Component {...props} />
13         ) : (
14           <Redirect to={{ pathname: "/" }} />
15         )
16       }
17     />
18   );
19 };
```

Bloque de código 5.4: Implementación del componente AzurePrivateRoute

5.6. Cierre de sesión

Para implementar el cierre de sesión del sistema, solo fue necesario crear un nuevo componente AzureLogout. Este componente utilizó la funcionalidad que provee la librería msal-browser para realizar el cierre de sesión con el proveedor, limpiando tanto la sesión en el cliente como en el servidor externo.

Al igual que con el inicio de sesión (Ver sección 5.4), la librería MSAL proporciona dos métodos distintos para llevar a cabo el cierre de sesión, diferenciándose por la interacción del usuario durante este procedimiento [38]. Estos métodos son conocidos como logoutRedirect y logoutPopup. Una vez más, se optó por emplear el uso del redirect (redirección), igual al elegido para el proceso de inicio de sesión. Este método redirige al usuario a la pantalla del

proveedor para realizar la solicitud de cierre de sesión, y una vez completada, retorna a la aplicación.

```
1  import { InteractionStatus } from "@azure/msal-browser";
2  import { useMsal } from "@azure/msal-react";
3  import { useEffect } from "react";
4
5  function AzureLogout() {
6    const { instance, inProgress } = useMsal();
7
8    useEffect(() => {
9      if (inProgress === InteractionStatus.None && instance) {
10       instance.logoutRedirect();
11     }
12   }, [instance]);
13
14   return <Loading />;
15 };
16
17 export default AzureLogout;
```

Bloque de código 5.5: Implementación del componente AzureLogout

Para realizar esta implementación, se utilizó el hook provisto por msal-react, **useMsal**, el cual provee la instancia del proveedor MSAL y el estado de la misma. El objeto de la instancia provee el método logoutRedirect, del cual se hizo uso para iniciar el proceso de logout. Esto sólo cuando la instancia se encuentra sin realizar algún otro proceso (inProgress === InteractionStatus.None).

Para la parte visual, sólo se requirió de renderizar otro componente de carga para mostrar al usuario que el sistema está realizando una acción, ya que el método logoutRedirect se encarga de redireccionar al usuario tras haber realizado con éxito el cierre de sesión.

Una vez que este componente fue implementado, se requirió de su integración con la funcionalidad actual de Logout (Especificado en la sección 4.4.1). Por lo que se agregó una condición adicional al componente de Logout previamente creado, de manera que, al intentar realizar el cierre de sesión utilizando Azure AD como proveedor, redirigiera al componente AzureLogout para iniciar el proceso y cerrar satisfactoriamente la sesión del usuario.

Capítulo 6

Conclusión

Mis conocimientos adquiridos durante mi etapa como estudiante en la Facultad de Informática de la Universidad de la Plata, me dio las herramientas necesarias para llevar a cabo una solución acorde para este problema. A nivel proceso, materias como Ingeniería de Software, me permitieron identificar requerimientos importantes para este desarrollo y, a su vez, agilizar cada etapa del mismo. A nivel diseño y arquitectura, materias como Patrones de Arquitecta de Software me aportaron conocimientos para saber identificar cada elemento que formaba parte del software y sus soluciones. A nivel desarrollo, Programación Orientada a Objetos me enseñó la importancia del código legible y la búsqueda de patrones que permitan llegar a una solución para problemas conocidos.

Gracias a esta experiencia tuve la oportunidad aplicar mis conocimientos aprendidos en el ámbito académico, y así participar completamente en el proceso de software. Trabajando tanto en el diseño arquitectónico de la solución, dividiendo la carga de trabajo que este requería en diferentes tareas, para finalmente trabajar en el código de la aplicación.

Luego de las diferentes etapas llevadas a cabo para el cumplimiento de los requerimientos, no solo se logró integrar con éxito un nuevo proveedor de autenticación en el sistema, sino que se mejoró íntegramente el producto. Se logró generar confianza en el código ya implementado y un código entendible para futuros desarrolladores que requieran mantenerlo sin tener conocimiento previo. Además, se logró la flexibilidad y escalabilidad del sistema. La arquitectura diseñada para la autenticación logró ser flexible y escalable, lo que permitirá adaptarse fácilmente a futuras actualizaciones y cambios en los requisitos del sistema. Esto garantizará que el sistema pueda crecer y evolucionar con las necesidades del negocio y los usuarios.

Finalmente, durante el desarrollo de la tesina, me encontré con varias lecciones aprendidas y recomendaciones para futuros proyectos similares. Lecciones tales como la comprensión de nuevas prácticas de desarrollo y la detección de futuros errores en las etapas iniciales del proceso de software.

6.1. Trabajos futuros

Basándome en las experiencias y aprendizajes obtenidos durante el desarrollo de la tesina, así como en las áreas de mejora identificadas, algunos trabajos futuros podrían incluir:

- Integración de nuevos proveedores de autenticación. Se podría seguir expandiendo la librería de componentes implementada para la utilización de diferentes proveedores de identidad, así como también la integración de otros flujos o protocolos de autenticación, realizando comparativas entre las diferentes herramientas utilizadas.
- Modificación de la librería para su utilización en otros tipos de arquitecturas como Single-Tenant.

Apéndice A

JSON Web Token

A.1. Introducción

JWT (JSON Web Token) es un estándar especificado en el documento RFC 7519 [39], el cual define un mecanismo para poder propagar entre dos partes, y de forma segura, la identidad de un determinado usuario, así como también privilegios o Claims del mismo. Estos privilegios están codificados en objetos de tipo JSON, que se forman parte del payload o cuerpo de un mensaje que va firmado digitalmente.

A.1.1. Usos

Existen diferentes escenarios en donde se pueden usar los JSON Web Tokens [40]:

- Autorización: Este es el caso más común de uso. Cuando el usuario inicia sesión en una aplicación, cada llamado posterior que se haga al servidor, incluirá el token JWT, permitiendo al usuario acceder a rutas, servicios y recursos que el token permita. De esta forma se evita el intercambio de las credenciales del usuario. Single Sign On es uno de los protocolos que utilizan JWT, debido a que permiten ser utilizados entre los diferentes dominios.
- Intercambio de información: Los token JWT son un buen mecanismo para transmitir de forma segura el intercambio de información entre partes. Debido a que los token JWT son firmados digitalmente, permitiendo confiar en que quien envió la información es quien dice ser.

A.2. Estructura

Un token JWT es representado como una secuencia de caracteres separadas por el carácter punto ('.'). Cada parte contiene un valor codificado utilizando en base64.

Figura A.1: Ejemplo de token JWT obtenido de jwt.io [40]

Al token JWT tiene tres partes:

- **Header**: encabezado dónde se especifica cierta información acerca del token.
- **Payload**: se especifican los datos de usuario y privilegios, así como toda la información que se necesite añadir.
- **Signature**: una firma que nos permite verificar si el token es válido.

A.2.1. Header

El header o encabezado es la primera cadena de caracteres que se obtiene de un token JWT y al decodificarlo de base64, se obtiene una estructura JSON conformada por:

- **typ** - Especifica el tipo de token, el cual en este caso es JWT.
- **alg** - Especifica el algoritmo utilizado para firmar el token, los cuales pueden ser HMAC SHA256 o RSA.

Por lo que, al decodificar de base64 el encabezado de un token se podría obtener una estructura JSON como la siguiente:

```
1 {
2   "alg": "HS256",
3   "typ": "JWT"
4 }
```

Bloque de código A.1: Ejemplo de header JWT decodificado

A.2.2. Payload

La segunda parte de un token es el payload, el cual contiene los llamados claims. Los claims son datos sobre una entidad (típicamente, el usuario) e información adicional que se requiera transportar en el token. Los nombres de los claims deben ser únicos.

```
{
  "iss": "joe",
  "exp": 1300819380,
  "is_root": true
}
```

Bloque de código A.2: Ejemplo de payload JWT decodificado

Existen tres tipos de claims: Claims registrados, públicos y privados.

Claims registrados

La razón del nombre de este tipo de claims es debido a que los valores que existen se encuentran registrados en el registro IANA "JSON Web Token Claims"[41].

Algunos ejemplos de estos tipos de claims que son listados pueden ser:

- **iss**. (issuer o emisor) Identifica a quien es el emisor del JWT. El uso de este claim es opcional.
- **exp**. (tiempo de expiración) Identifica el tiempo de expiración tras el cual el JWT debe dejar de ser aceptado para su procesamiento.
- **aud** (audiencia) Identifica a los destinatarios a los que se dirige el JWT. Si quien procesa el claim no se identifica con el valor de audiencia, entonces el JWT debe ser rechazado.

Claims Públicos

Estos pueden ser definidos a demanda, al momento de crear el JWTs. Pero se debe evitar colisiones con los claims registrados por el registro de IANA o deben ser un nombre público, es decir, un valor que posea resistencia a colisiones.

Claims Privados

Estos claims personalizados son creados para compartir información entre las partes que aceptan usarlos y no son claims registrados ni públicos. A diferencia de los públicos, estos tipos de claims pueden colisionar y deben ser usados con precaución.

A.2.3. Signature

El signature o firma de un JSON Web Token permite verificar que el remitente del token es quien dice ser, y que el mensaje no se ha modificado por el camino.

Para crear la firma, se toma el encabezado codificado en base64, el cuerpo codificado en base64, y una clave secreta para firmar digitalmente esos datos utilizando el algoritmo especificado en el encabezado del token.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Bloque de código A.3: Firmado digital de token utilizando el algoritmo SHA256

De esta forma, si alguien modifica el token por el camino, por ejemplo, inyectando alguna credencial o algún dato falso, entonces podríamos verificar que la comprobación de la firma no es correcta, por lo que no podemos confiar en el token recibido y se descarta.

A.3. Funcionamiento del token en la autenticación

Cuando un usuario se autentica en un sistema usando sus credenciales, los servidores de autorización retornan un JSON Web Token. Como estos tokens reemplazan a las credenciales, se debe tener cuidado para prevenir problemas de seguridad, por lo que no se deben mantener los tokens almacenados más de lo necesario.

En el momento en el que un usuario requiere de acceder a una ruta protegida o un recurso, el cliente del usuario debe enviar el JWT en cada petición utilizando típicamente el encabezado Authorization, con el esquema Bearer. Por lo que el encabezado debería quedar de la siguiente manera:

```
Authorization: Bearer <token>
```

Bloque de código A.4: Encabezado de una petición utilizando Authorization

Esto puede ser utilizado en ciertos casos como un mecanismo de autorización sin estado. Lo que quiere decir que no existe una sesión del lado del servidor, ya que al comprobar la veracidad del token JWT (Los son correctos tras verificar la firma y el token ha expirado), se puede verificar que el usuario tiene una sesión válida.

Bibliografía

[1] G. A. Gellert, J. F. Crouch, L. A. Gibson, G. S. Conklin, S. L. Webster y J. A. Gillean, «Clinical impact and value of workstation single sign-on,» *International Journal of Medical Informatics*, vol. 101, págs. 131-136, 2017, ISSN: 1386-5056. DOI: https://doi.org/10.1016/j.ijmedinf.2017.02.008. dirección: https://www.sciencedirect.com/science/article/pii/S1386505617300394.

[2] L. S. Ramamoorthi y D. Sarkar, «Single sign-on: A solution approach to address inefficiencies during sign-out process,» *IEEE Access*, vol. 8, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3033570.

[3] T. Bazaz y A. Khalique, «A Review on Single Sign on Enabling Technologies and Protocols,» *International Journal of Computer Applications*, vol. 151, 11 2016. DOI: 10.5120/ijca2016911938.

[4] C. Mainka, V. Mladenov, F. Feldmann, J. Krautwald y J. Schwenk, «Your software at my service : Security analysis of SaaS single sign-on solutions in the cloud,» vol. 2014-November, 2014. DOI: 10.1145/2664168.2664172.

[5] J. Kabbedijk, C. P. Bezemer, S. Jansen y A. Zaidman, «Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective,» *Journal of Systems and Software*, vol. 100, 2015, ISSN: 01641212. DOI: 10.1016/j.jss.2014.10.034.

[6] K. Munir y S. Palaniappan, «Simple authentication process for cloud user,» IEEE Computer Society, septiembre de 2014, págs. 1175-1181, ISBN: 9781479938247. DOI: 10.1109/ICMCS.2014.6911264.

[7] C. P. Bezemer y A. Zaidman, «Multi-tenant SaaS applications: Maintenance dream or nightmare?,» 2010. DOI: 10.1145/1862372.1862393.

[8] M. Almorsy, J. Grundy y A. S. Ibrahim, «TOSSMA: A tenant-oriented SaaS security management architecture,» 2012. DOI: 10.1109/CLOUD.2012.146.

[9] M. Papathanasaki, L. Maglaras y N. Ayres, «Modern Authentication Methods: A Comprehensive Survey,» *AI, Computer Science and Robotics Technology*, vol. 2022, 2022. DOI: 10.5772/acrt.08.

[10] *Okta SSO*. dirección: https://www.okta.com/products/single-sign-on/.

[11] P. Pandey y T. N. Nisha, «Challenges in Single Sign-On,» vol. 1964, 2021. DOI: 10.1088/1742-6596/1964/4/042016.

[12] *Overview of Single Sign-On in the OIN*. dirección: https://developer.okta.com/docs/guides/oin-sso-overview/.

[13] V. Beltran, «Characterization of web single sign-on protocols,» *IEEE Communications Magazine*, vol. 54, págs. 24-30, 7 julio de 2016, ISSN: 01636804. DOI: 10.1109/MCOM.2016.7514160.

[14] *OAuth 2.0 and OpenID Connect overview*. dirección: https://developer.okta.com/docs/concepts/oauth-openid/.

[15] Y. Wilson y A. Hingnikar, *Solving Identity Management in Modern Applications*. 2019. DOI: 10.1007/978-1-4842-5095-2.

[16] D. Hardt, *The OAuth 2.0 Authorization Framework*, RFC 6749, octubre de 2012. DOI: 10.17487/RFC6749. dirección: https://www.rfc-editor.org/info/rfc6749.

[17] N. Sakimura, J. Bradley y N. Agarwal, *Proof Key for Code Exchange by OAuth Public Clients*, RFC 7636, septiembre de 2015. DOI: 10.17487/RFC7636. dirección: https://www.rfc-editor.org/info/rfc7636.

[18] *Implement authorization by grant type*. dirección: https://developer.okta.com/docs/guides/implement-grant-type/authcodepkce/main/.

[19] *OAuth 2.0 and OpenID Connect overview*. dirección: https://developer.okta.com/docs/concepts/oauth-openid/.

[20] *Okta React SDK - NPM Registry*. dirección: https://www.npmjs.com/package/@okta/okta-react.

[21] *Okta React SDK*. dirección: https://github.com/okta/okta-react.

[22] *React Hooks*. dirección: https://es.react.dev/reference/react/hooks.

[23] *Componentes de orden superior*. dirección: https://es.legacy.reactjs.org/docs/higher-order-components.html.

[24] V. K. Madasu, T. V. S. N. Venna y T. Eltaeib, «SOLID Principles in Software Architecture and Introduction to RESM Concept in OOP,» *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, vol. 2, págs. 3159-3199, 2 2015. dirección: www.jmest.org.

[25] E. Moniz, *Applying SOLID To React*. dirección: https://medium.com/docler-engineering/applying-solid-to-react-ca6d1ff926a4.

[26] *Passing Data Deeply with Context*. dirección: https://react.dev/learn/passing-data-deeply-with-context.

[27] *Objeto Window en Javascript*. dirección: https://developer.mozilla.org/es/docs/Web/API/Window.

[28] Okta, *Redirect Authentication*. dirección: https://developer.okta.com/docs/concepts/redirect-vs-embedded/#redirect-authentication.

[29] *Sign users in to your SPA using the redirect model*. dirección: https://developer.okta.com/docs/guides/sign-into-spa-redirect/react/main/.

[30] Microsoft, *¿Qué es Microsoft Entra ID?* Dirección: https://learn.microsoft.com/es-es/entra/fundamentals/whatis.

[31] Microsoft, *¿Qué es la plataforma de identidad de Microsoft?* Dirección: https://learn.microsoft.com/es-mx/entra/identity-platform/v2-overview.

[32] *Microsoft Authentication Library for JavaScript (MSAL.js)*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js.

[33] Microsoft, *MSAL Initialization*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-browser/docs/initialization.md#initialization-of-msal.

[34] Microsoft, *MSAL Initialization Parametters*. dirección: https://learn.microsoft.com/en-us/entra/identity-platform/msal-js-initializing-client-applications#prerequisites.

[35] Microsoft, *MSAL React Initialization*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-react/docs/getting-started.md#initialization.

[36] Microsoft, *Azure AD - Login*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-browser/docs/login-user.md#login-user.

[37] Microsoft, *Azure AD - Login Interaction Type*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-browser/docs/initialization.md#choosing-an-interaction-type.

[38] Microsoft, *Logging Out of MSAL*. dirección: https://github.com/AzureAD/microsoft-authentication-library-for-js/blob/dev/lib/msal-browser/docs/logout.md.

[39] M. B. Jones, J. Bradley y N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, mayo de 2015. DOI: 10.17487/RFC7519. dirección: https://www.rfc-editor.org/info/rfc7519.

[40] *Introduction to JSON Web Tokens*. dirección: https://jwt.io/introduction.

[41] *JSON Web Token Claims*. dirección: https://www.iana.org/assignments/jwt/jwt.xhtml.