Documentación: API Users

***Resumen*—Este trabajo presenta el diseño e implementación de una API de gestión de usuarios basada en la arquitectura Traccar, una plataforma de código abierto para el rastreo de dispositivos GPS. La API fue desarrollada con un enfoque en la escalabilidad, modularidad y seguridad, implementando técnicas avanzadas como la autenticación basada en JSON Web Tokens (JWT) y la verificación multifactor mediante Google Authenticator. La API permite realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre usuarios y gestionar roles y permisos a través de un sistema de control de acceso basado en roles (RBAC). Se utilizó Swagger para documentar y facilitar la interacción con los endpoints de la API, mientras que el despliegue se realizó utilizando contenedores Docker para asegurar la replicabilidad y escalabilidad. Los resultados obtenidos muestran una gestión eficiente de usuarios y un alto nivel de seguridad, mitigando riesgos como ataques de intermediario y accesos no autorizados. En este contexto, se discuten los desafíos encontrados durante la implementación, incluyendo la complejidad en la gestión de permisos y autenticación, así como posibles mejoras futuras para optimizar la seguridad y el rendimiento de la API.**

I. Introducción

En los últimos años, la creciente adopción de tecnologías de microservicios, la computación en la nube y sistemas distribuidos ha elevado la importancia de los mecanismos robustos de gestión de usuarios y autenticación. A medida que las aplicaciones evolucionan hacia arquitecturas más modulares, como los microservicios, surge el desafío de implementar controles de acceso y autenticación escalables y seguros. La autenticación basada en roles (*RBAC*) ha emergido como una de las soluciones más eficientes para gestionar los permisos de los usuarios, reduciendo la carga administrativa y mejorando la seguridad mediante la asignación precisa de permisos según roles organizacionales [1], [2].

El uso de tecnologías emergentes, como *blockchain*, ha permitido mejoras sustanciales en la autenticación y la gestión de accesos, ofreciendo soluciones descentralizadas que eliminan los puntos únicos de fallo. Esto ha resultado en modelos de autenticación más resilientes frente a amenazas comunes, como ataques *man-in-the-middle*, y ha mejorado la verificación y validación de roles sin depender de una autoridad centralizada [3].

Asimismo, las arquitecturas basadas en microservicios han requerido la implementación de mecanismos de autenticación más flexibles y seguros, como *OAuth 2.0*, que permiten la verificación y autorización de usuarios en entornos de API altamente distribuidos. Estos sistemas no solo mejoran la velocidad de respuesta ante incidentes de seguridad, sino que también proporcionan una mayor flexibilidad en la gestión de permisos en tiempo real [4].

El uso de contratos inteligentes en plataformas como *Ethereum* ha permitido automatizar la gestión de permisos de usuario mediante *blockchain*, asegurando la integridad y trazabilidad de las transacciones relacionadas con roles y recursos en una organización [3]. Además, la autenticación mediante servicios en la nube, como *Firebase*, ha ganado popularidad en la implementación de sistemas *RESTful*, eliminando la manipulación indebida de datos y proporcionando una verificación estricta para evitar suplantaciones de identidad [5].

II. Materiales y Métodos

Proyecto y Arquitectura Base

El proyecto se desarrolló sobre la arquitectura *Traccar*, una plataforma de código abierto para rastreo de dispositivos GPS que incluye una API para la gestión de usuarios. La arquitectura de *Traccar* está basada en un modelo de microservicios que permite una escalabilidad y seguridad en la gestión de grandes volúmenes de datos y dispositivos conectados. Dentro de este contexto, la API de usuarios se diseñó para manejar la creación, actualización, autenticación y eliminación de usuarios, incorporando controles de acceso basados en roles.

Tecnologías Utilizadas

- **Lenguaje de programación:** Java, utilizado para la implementación de la API.
- **Frameworks:** El proyecto se desarrolló utilizando el framework *Jakarta EE* para el manejo de recursos RESTful.
- **Autenticación y Autorización:** Se implementó un sistema de autenticación basado en *JSON Web Tokens (JWT)*, que proporciona seguridad adicional para la comunicación entre clientes y servidores.
- **Base de Datos:** Para la persistencia de datos de usuarios, se utilizó una base de datos SQL en la que se almacenan las credenciales y datos de configuración del sistema de usuarios.
- **Control de Versiones:** Se utilizó *Git* para gestionar el código fuente y mantener un registro de los cambios realizados a lo largo del desarrollo.

Metodología de Desarrollo

a) Diseño de la API:: La API de usuarios fue diseñada con un enfoque en la modularidad y escalabilidad. Cada recurso relacionado con los usuarios se implementó como un servicio REST independiente, que incluye los métodos *GET*, *POST*, *PUT* y *DELETE* para realizar operaciones CRUD sobre los usuarios.

b) Documentación y Pruebas:: Se utilizó *Swagger* para documentar la API de manera detallada. Esto permitió generar automáticamente la documentación interactiva de los *endpoints* y facilitar el entendimiento de los parámetros y respuestas de cada método.

Se implementaron pruebas unitarias y de integración utilizando *JUnit*, simulando solicitudes a la API y verificando el correcto funcionamiento de las operaciones de autenticación, manejo de roles y gestión de usuarios.

c) Manejo de Seguridad:: Se integró el uso de *Google Authenticator* para generar contraseñas de un solo uso (*TOTP*), como una capa adicional de seguridad para la autenticación de usuarios. Se implementaron verificaciones de permisos, garantizando que solo los usuarios con los roles adecuados (por ejemplo, administradores) pudieran acceder o modificar información sensible.

d) Despliegue y Gestión de API:: El sistema fue desplegado en un entorno de prueba utilizando contenedores *Docker* para asegurar la replicabilidad del entorno de desarrollo y producción. La comunicación entre el *frontend* y *backend* se realizó utilizando una API RESTful, permitiendo una interacción segura y eficiente entre los servicios.

Herramientas

- **Postman:** Herramienta utilizada para probar manualmente los *endpoints* de la API, verificando respuestas esperadas y la funcionalidad de los métodos de la API de usuarios.
- **GitHub:** Utilizado para el control de versiones y la colaboración entre los diferentes integrantes del equipo de desarrollo.

III. Resultados

Este apartado describe las principales clases relacionadas con la gestión de usuarios en el sistema Traccar, incluyendo las funcionalidades más importantes como la autenticación, autorización y manejo de roles.

A. UserResource.java

La clase `UserResource` extiende de `BaseObjectResource<User>`, lo que significa que hereda funcionalidades comunes para manejar objetos de tipo `User`. Esta clase está anotada con `@Path("users")`, lo que la convierte en el punto de entrada para todas las operaciones relacionadas con usuarios en la API. Aquí está un resumen de sus principales responsabilidades:

- **Manejo de solicitudes GET:** Permite recuperar una lista de usuarios. Dependiendo de los parámetros (`userId` o `deviceId`), puede filtrar la lista por usuarios específicos o por dispositivos asignados a los usuarios. Si no se proporcionan parámetros, devuelve la lista completa de usuarios.
- **Manejo de solicitudes POST:** Permite agregar un nuevo usuario al sistema. Antes de la creación, se realizan varias validaciones, como la verificación de permisos, límites de usuario para administradores, y la configuración de valores predeterminados para el nuevo usuario. También maneja la configuración de contraseñas y el registro de acciones en el sistema de logs.
- **Manejo de solicitudes DELETE:** Permite eliminar un usuario por su `id`. Si el usuario que se está eliminando es el mismo que realiza la solicitud, también se elimina su sesión.
- **Generación de claves TOTP:** Permite generar una clave para un sistema de contraseñas de un solo uso (`TOTP`), habilitado solo si el servidor tiene la funcionalidad `TOTP` activada.

Además, esta clase implementa una serie de verificaciones de permisos, asegurando que solo los usuarios autorizados puedan realizar acciones específicas como la creación y eliminación de usuarios, y configura la seguridad, incluyendo la validación del límite de usuarios para administradores y la generación de contraseñas de un solo uso (`TOTP`). También gestiona las sesiones de usuario y utiliza `LogAction` para registrar la creación, enlace y eliminación de usuarios en los logs del sistema.

Listing 1. UserResource.java

```
/*
* Copyright 2015 - 2024 Anton Tananaev (
    anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (
    the "License");
* you may not use this file except in compliance
    with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in
    writing, software
* distributed under the License is distributed on an
    "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
    either express or
* implied.
* See the License for the specific language
    governing permissions and
* limitations under the License.
*/
package org.traccar.api.resource;

import com.warrenstrange.googleauth.
    GoogleAuthenticator;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import org.traccar.api.BaseObjectResource;
import org.traccar.config.Config;
import org.traccar.config.Keys;
import org.traccar.helper.LogAction;
```

```
import org.traccar.helper.model.UserUtil;
import org.traccar.model.*;
import org.traccar.storage.StorageException;
import org.traccar.storage.query.Columns;
import org.traccar.storage.query.Condition;
import org.traccar.storage.query.Request;

import java.util.Collection;

@Path("users")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class UserResource extends BaseObjectResource
    <User> {

    @Inject
    private Config config;

    @Context
    private HttpServletRequest request;

    public UserResource() {
        super(User.class);
    }

    @GET
    public Collection<User> get(
            @QueryParam("userId") long userId,
            @QueryParam("deviceId") long deviceId)
                throws StorageException {

        if (userId > 0) {
            permissionsService.checkUser(getUserId()
                , userId);
            return storage.getObjects(baseClass, new
                 Request(
                    new Columns.All(),
                    new Condition.Permission(User.
                        class, userId, ManagedUser.
                        class).excludeGroups()));
        } else if (deviceId > 0) {
            permissionsService.checkAdmin(getUserId
                ());
            return storage.getObjects(baseClass, new
                 Request(
                    new Columns.All(),
                    new Condition.Permission(User.
                        class, Device.class,
                        deviceId).excludeGroups()));
        } else if (permissionsService.notAdmin(
            getUserId())) {
            return storage.getObjects(baseClass, new
                 Request(
                    new Columns.All(),
                    new Condition.Permission(User.
                        class, getUserId(),
                        ManagedUser.class).
                        excludeGroups()));
        } else {
            return storage.getObjects(baseClass, new
                 Request(new Columns.All()));
        }
    }

    @Override
    @PermitAll
    @POST
    public Response add(User entity) throws
        StorageException {
        User currentUser = getUserId() > 0 ?
            permissionsService.getUser(getUserId())
            : null;
        if (currentUser == null || !currentUser.
            getAdministrator()) {
            permissionsService.checkUserUpdate(
                getUserId(), new User(), entity);
            if (currentUser != null && currentUser.
                getUserLimit() != 0) {
                int userLimit = currentUser.
                    getUserLimit();
                if (userLimit > 0) {
                    int userCount = storage.
                        getObjects(baseClass, new
                        Request(
                            new Columns.All(),
                            new Condition.Permission
                                (User.class,
                                getUserId(),
                                ManagedUser.class).
                                excludeGroups()))
                        .size();
                    if (userCount >= userLimit) {
                        throw new SecurityException(
                            "Manager user limit
                            reached");
                    }
                }
            } else {
                if (UserUtil.isEmpty(storage)) {
                    entity.setAdministrator(true);
                } else if (!permissionsService.
                    getServer().getRegistration()) {
                    throw new SecurityException("
                        Registration disabled");
                }
                if (permissionsService.getServer().
                    getBoolean(Keys.WEB_TOTP_FORCE.
                    getKey())
                        && entity.getTotpKey() ==
                            null) {
                    throw new SecurityException("One
                        -time password key is
                        required");
                }
                UserUtil.setUserDefaults(entity,
                    config);
            }
        }

        entity.setId(storage.addObject(entity, new
            Request(new Columns.Exclude("id"))));
        storage.updateObject(entity, new Request(
                new Columns.Include("hashedPassword"
                    , "salt"),
                new Condition.Equals("id", entity.
                    getId())));

        LogAction.create(getUserId(), entity);

        if (currentUser != null && currentUser.
            getUserLimit() != 0) {
            storage.addPermission(new Permission(
                User.class, getUserId(), ManagedUser
                .class, entity.getId()));
            LogAction.link(getUserId(), User.class,
                getUserId(), ManagedUser.class,
                entity.getId());
        }
        return Response.ok(entity).build();
    }

    @Path("{id}")
    @DELETE
    public Response remove(@PathParam("id") long id)
         throws Exception {
        Response response = super.remove(id);
        if (getUserId() == id) {
            request.getSession().removeAttribute(
```

```
                SessionResource.USER_ID_KEY);
        }
        return response;
    }

    @Path("totp")
    @PermitAll
    @POST
    public String generateTotpKey() throws
        StorageException {
        if (!permissionsService.getServer().
            getBoolean(Keys.WEB_TOTP_ENABLE.getKey()
            )) {
            throw new SecurityException("One-time
                password is disabled");
        }
        return new GoogleAuthenticator().
            createCredentials().getKey();
    }
}
```

B. *UserPrincipal.java*

La clase `UserPrincipal` se utiliza para encapsular la identidad del usuario dentro del sistema de seguridad de Traccar. Esta clase mantiene información básica sobre el usuario autenticado, como su `userId` y la fecha de expiración de su sesión o token de autenticación.

1) Funcionalidades Principales:

- **Almacenamiento del ID del Usuario:** La clase contiene un campo `userId` que almacena el identificador único del usuario en el sistema. Este ID se utiliza para identificar al usuario en las operaciones y verificaciones de seguridad.
- **Gestión de Expiración de Sesión/Token:** La clase también almacena una `Date expiration`, que indica el momento en que expira la sesión del usuario o su token de autenticación. Esto es crucial para gestionar el tiempo de vida de la autenticación.
- **Interfaz Principal:** La clase implementa la interfaz `Principal`, que es un contrato en Java para representar a una entidad (en este caso, un usuario). Sin embargo, el método `getName()` está sobrescrito para devolver `null`, lo que sugiere que el nombre del usuario no se utiliza directamente en esta implementación.

2) Métodos:

- `UserPrincipal(long userId, Date expiration)`: Este constructor inicializa el `userId` y la `expiration` al crear una instancia de `UserPrincipal`.
- `getUserId()`: Devuelve el identificador único del usuario.
- `getExpiration()`: Devuelve la fecha de expiración de la sesión o token de autenticación.
- `getName()`: Implementación del método de la interfaz `Principal`, pero devuelve `null` porque el nombre del usuario no se utiliza.

3) Usos Comunes:

- **Autenticación y Autorización:** `UserPrincipal` es típicamente utilizado en el contexto de autenticación y autorización para identificar al usuario actual y verificar su identidad y permisos en el sistema.
- **Gestión de Sesiones:** Se utiliza para verificar si la sesión o el token de autenticación de un usuario ha expirado y tomar acciones correspondientes, como pedirle que vuelva a iniciar sesión.

Listing 2. UserPrincipal.java

```
/*
* Copyright 2015 - 2023 Anton Tananaev (
    anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (
    the "License");
* you may not use this file except in compliance
    with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in
    writing, software
* distributed under the License is distributed on an
     "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
    either express or implied.
* See the License for the specific language
    governing permissions and
* limitations under the License.
*/
package org.traccar.api.security;

import java.security.Principal;
import java.util.Date;

public class UserPrincipal implements Principal {
    private final long userId;
    private final Date expiration;

    public UserPrincipal(long userId, Date
        expiration) {
        this.userId = userId;
        this.expiration = expiration;
    }

    public Long getUserId() {
        return userId;
    }

    public Date getExpiration() {
        return expiration;
    }

    @Override
    public String getName() {
        return null;
    }
}
```

C. *UserSecurityContext.java*

La clase `UserSecurityContext` implementa la interfaz `SecurityContext` y está diseñada para proporcionar información sobre el contexto de seguridad de un usuario autenticado. La clase se construye a partir de un objeto `UserPrincipal`, que representa la identidad del usuario autenticado, y proporciona varios métodos clave para determinar el contexto de seguridad de la solicitud actual.

1) Funcionalidades Principales:

- **Proporcionar la Identidad del Usuario:** La clase almacena un objeto `UserPrincipal`, que representa la identidad del usuario autenticado en la aplicación. El método `getUserPrincipal()` devuelve este objeto, permitiendo que otros componentes del sistema puedan acceder a la identidad del usuario.
- **Verificar Roles de Usuario:** Aunque la implementación de `isUserInRole(String role)` siempre devuelve `true`, en un contexto más complejo, este método podría ser utilizado para verificar si un usuario pertenece a un rol específico.
- **Determinar si la Conexión es Segura:** El método `isSecure()` devuelve `false`, lo que indica que la implementación actual no está considerando si la conexión es segura (por ejemplo, si está utilizando HTTPS). En un entorno de producción, este método podría ser extendido para realizar esta verificación.
- **Esquema de Autenticación:** El método `getAuthenticationScheme()` devuelve `BASIC_AUTH`, lo que sugiere que el esquema de autenticación utilizado es Basic Authentication. Este es un método estándar para autenticar usuarios en HTTP, en el cual las credenciales se envían codificadas en base64.

2) Métodos:

- `UserSecurityContext(UserPrincipal principal)`: Inicializa la clase con un objeto `UserPrincipal`, que contiene la identidad del usuario autenticado.
- `getUserPrincipal()`: Devuelve el objeto `UserPrincipal`, que representa la identidad del usuario autenticado.
- `isUserInRole(String role)`: Devuelve `true` por defecto, pero en un sistema más avanzado podría determinar si el usuario tiene un rol específico.
- `isSecure()`: Devuelve `false`, indicando que la seguridad de la conexión no se está evaluando en esta implementación.
- `getAuthenticationScheme()`: Devuelve `BASIC_AUTH`, indicando que el esquema de autenticación utilizado es Basic Authentication.

3) Usos Comunes:

- **Contexto de Seguridad en Solicitudes HTTP:** `UserSecurityContext` es utilizado para manejar la autenticación y proporcionar información de seguridad en el contexto de una solicitud HTTP.
- **Autenticación de Usuarios:** Proporciona una forma estándar de autenticar usuarios y verificar su identidad dentro del sistema.

Listing 3. UserSecurityContext.java

```
/*
* Copyright 2015 - 2022 Anton Tananaev (
    anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (
    the "License");
* you may not use this file except in compliance
    with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in
    writing, software
* distributed under the License is distributed on an
    "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
    either express or implied.
* See the License for the specific language
    governing permissions and
* limitations under the License.
*/
package org.traccar.api.security;

import jakarta.ws.rs.core.SecurityContext;
import java.security.Principal;

public class UserSecurityContext implements
    SecurityContext {
    private final UserPrincipal principal;

    public UserSecurityContext(UserPrincipal
        principal) {
        this.principal = principal;
    }

    @Override
    public Principal getUserPrincipal() {
        return principal;
    }

    @Override
    public boolean isUserInRole(String role) {
        return true;
    }

    @Override
    public boolean isSecure() {
        return false;
    }

    @Override
    public String getAuthenticationScheme() {
        return BASIC_AUTH;
    }
}
```

D. User.java

La clase `User` extiende de `ExtendedModel` e implementa las interfaces `UserRestrictions` y `Disableable`, lo que sugiere que hereda funcionalidades adicionales y sigue ciertos contratos para gestionar restricciones y el estado de habilitación/deshabilitación del usuario. Esta clase está anotada con `@StorageName("tc_users")`, lo que indica que los objetos de esta clase se almacenan en una tabla de base de datos llamada `tc_users`.

1) Atributos Principales: La clase `User` contiene una serie de atributos que definen los datos y la configuración de un usuario en el sistema. Algunos de los atributos más importantes incluyen:

- **Datos Personales:** `name` (nombre del usuario), `login` (nombre de usuario), `email` (correo electrónico), `phone` (teléfono).

- **Roles y Permisos:** `readonly` (indica si el usuario tiene permisos de solo lectura), `administrator` (indica si es administrador), `userLimit` (límite de usuarios que un administrador puede gestionar), `deviceLimit` (límite de dispositivos que el usuario puede gestionar).
- **Configuración de Mapas:** Atributos como `map`, `latitude`, `longitude`, `zoom` y `coordinateFormat`.
- **Seguridad:** `hashedPassword` (contraseña cifrada), `salt` (sal para el hash), `totpKey` (clave para el sistema TOTP), `expirationTime` (fecha de expiración de la cuenta).
- **Estado del Usuario:** Atributos como `disabled` (indica si la cuenta está deshabilitada) y `temporary` (indica si la cuenta es temporal).

2) Métodos Principales:

- **Getters y Setters:** Métodos estándar para obtener y establecer los valores de los atributos mencionados.
- **Manejo de Contraseñas:** `setPassword(String password)` genera un hash seguro de la contraseña, `isPasswordValid(String password)` verifica si una contraseña dada coincide con la almacenada, y `getHashedPassword()` y `getSalt()` devuelven la contraseña cifrada y la sal.
- **Comparación de Usuarios:** El método `compare(User other, String... exclusions)` compara dos objetos `User`, excluyendo ciertos atributos, para determinar si son equivalentes.
- **Implementación de Interfaces:** `UserRestrictions` y `Disableable`, para aplicar restricciones a los usuarios y gestionar su estado.

3) Usos Comunes:

- **Gestión de Usuarios:** La clase `User` se utiliza para representar usuarios, gestionar roles, permisos y el estado de la cuenta.
- **Seguridad y Autenticación:** Maneja aspectos como la gestión de contraseñas y la autenticación de dos factores (TOTP).
- **Configuración de la Interfaz de Usuario:** Permite personalizar la configuración del mapa y otros aspectos visuales de la interfaz.

Listing 4. User.java

```
/*
* Copyright 2013 - 2024 Anton Tananaev (anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
@StorageName("tc_users")
public class User extends ExtendedModel implements UserRestrictions, Disableable {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    private String login;
    public String getLogin() {
        return login;
    }
    public void setLogin(String login) {
        this.login = login;
    }

    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email.trim();
    }

    private String phone;
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone != null ? phone.trim() : null;
    }

    private boolean readonly;
    @Override
    public boolean getReadonly() {
        return readonly;
    }
    public void setReadonly(boolean readonly) {
        this.readonly = readonly;
    }

    private boolean administrator;
    @QueryIgnore
    @JsonIgnore
    public boolean getManager() {
        return userLimit != 0;
    }
    public boolean getAdministrator() {
        return administrator;
    }
    public void setAdministrator(boolean administrator) {
        this.administrator = administrator;
    }

    private String map;
    public String getMap() {
        return map;
    }
    public void setMap(String map) {
        this.map = map;
    }

    private double latitude;
```

```
    public double getLatitude() {
        return latitude;
    }
    public void setLatitude(double latitude) {
        this.latitude = latitude;
    }

    private double longitude;
    public double getLongitude() {
        return longitude;
    }
    public void setLongitude(double longitude) {
        this.longitude = longitude;
    }

    private int zoom;
    public int getZoom() {
        return zoom;
    }
    public void setZoom(int zoom) {
        this.zoom = zoom;
    }

    private String coordinateFormat;
    public String getCoordinateFormat() {
        return coordinateFormat;
    }
    public void setCoordinateFormat(String
            coordinateFormat) {
        this.coordinateFormat = coordinateFormat;
    }

    private boolean disabled;
    @Override
    public boolean getDisabled() {
        return disabled;
    }
    @Override
    public void setDisabled(boolean disabled) {
        this.disabled = disabled;
    }

    private Date expirationTime;
    @Override
    public Date getExpirationTime() {
        return expirationTime;
    }
    @Override
    public void setExpirationTime(Date
            expirationTime) {
        this.expirationTime = expirationTime;
    }

    private int deviceLimit;
    public int getDeviceLimit() {
        return deviceLimit;
    }
    public void setDeviceLimit(int deviceLimit) {
        this.deviceLimit = deviceLimit;
    }

    private int userLimit;
    public int getUserLimit() {
        return userLimit;
    }
    public void setUserLimit(int userLimit) {
        this.userLimit = userLimit;
    }

    private boolean deviceReadonly;
    @Override
    public boolean getDeviceReadonly() {
        return deviceReadonly;
    }
    public void setDeviceReadonly(boolean
            deviceReadonly) {
        this.deviceReadonly = deviceReadonly;
    }

    private boolean limitCommands;
    @Override
    public boolean getLimitCommands() {
        return limitCommands;
    }
    public void setLimitCommands(boolean
            limitCommands) {
        this.limitCommands = limitCommands;
    }

    private boolean disableReports;
    @Override
    public boolean getDisableReports() {
        return disableReports;
    }
    public void setDisableReports(boolean
            disableReports) {
        this.disableReports = disableReports;
    }

    private boolean fixedEmail;
    @Override
    public boolean getFixedEmail() {
        return fixedEmail;
    }
    public void setFixedEmail(boolean fixedEmail) {
        this.fixedEmail = fixedEmail;
    }

    private String poiLayer;
    public String getPoiLayer() {
        return poiLayer;
    }
    public void setPoiLayer(String poiLayer) {
        this.poiLayer = poiLayer;
    }

    private String totpKey;
    public String getTotpKey() {
        return totpKey;
    }
    public void setTotpKey(String totpKey) {
        this.totpKey = totpKey;
    }

    private boolean temporary;
    public boolean getTemporary() {
        return temporary;
    }
    public void setTemporary(boolean temporary) {
        this.temporary = temporary;
    }

    @QueryIgnore
    public String getPassword() {
        return null;
    }

    @QueryIgnore
    public void setPassword(String password) {
        if (password != null && !password.isEmpty())
                {
            Hashing.HashingResult hashingResult =
                    Hashing.createHash(password);
            hashedPassword = hashingResult.getHash()
                    ;
            salt = hashingResult.getSalt();
        }
    }
```

```
    private String hashedPassword;
    @JsonIgnore
    @QueryIgnore
    public String getHashedPassword() {
        return hashedPassword;
    }
    @QueryIgnore
    public void setHashedPassword(String
        hashedPassword) {
        this.hashedPassword = hashedPassword;
    }

    private String salt;
    @JsonIgnore
    @QueryIgnore
    public String getSalt() {
        return salt;
    }
    @QueryIgnore
    public void setSalt(String salt) {
        this.salt = salt;
    }

    public boolean isPasswordValid(String password)
        {
        return Hashing.validatePassword(password,
            hashedPassword, salt);
    }

    public boolean compare(User other, String...
        exclusions) {
        if (!EqualsBuilder.reflectionEquals(this,
            other, "attributes", "hashedPassword", "
            salt")) {
            return false;
        }
        var thisAttributes = new HashMap<>(
            getAttributes());
        var otherAttributes = new HashMap<>(other.
            getAttributes());
        for (String exclusion : exclusions) {
            thisAttributes.remove(exclusion);
            otherAttributes.remove(exclusion);
        }
        return thisAttributes.equals(otherAttributes
            );
    }
}
```

E. Users.js

Ext JS Data Store: La clase `Traccar.store.Users` extiende de `Ext.data.Store`, lo que significa que hereda todas las capacidades de manejo de datos de Ext JS. Esta tienda específicamente maneja los datos de usuarios (`User`) en la aplicación.

1) Modelo de Datos: La tienda está configurada para utilizar el modelo `Traccar.model.User`, que define la estructura de los datos del usuario. Esto asegura que los datos manipulados por la tienda se ajusten al formato esperado del modelo `User`.

2) Proxy de Datos REST: La tienda utiliza un proxy de tipo `rest`, lo que significa que interactúa con un servicio RESTful para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos de los usuarios. El `url` especifica el `endpoint` de la API (`api/users`) con el que se comunica para realizar estas operaciones.

3) Writer Configuration: La configuración `writer` con la opción `writeAllFields: true` indica que, cuando se envían datos al servidor (por ejemplo, al crear o actualizar un usuario), todos los campos del modelo serán incluidos en la solicitud, incluso si algunos no han sido modificados.

4) Funcionalidades Principales:

- **Gestión de Datos de Usuarios:** Esta tienda se utiliza para manejar una colección de usuarios dentro de la interfaz de usuario de Traccar. Permite cargar, guardar y sincronizar la información de los usuarios con el servidor.
- **Interacción con el Backend:** A través del proxy, la tienda realiza solicitudes REST al `endpoint` `api/users` para cargar datos de usuarios desde el servidor, así como para enviar cambios (creación, edición, eliminación) de vuelta al servidor.

5) Usos Comunes:

- **Visualización de Usuarios:** La tienda `Users` se utiliza en componentes de la interfaz de usuario que requieren mostrar una lista de usuarios, como tablas o listas.
- **Edición de Usuarios:** Cuando se realizan cambios a los datos de un usuario en la interfaz (por ejemplo, a través de un formulario), estos cambios se envían al servidor utilizando esta tienda.
- **Sincronización de Datos:** Asegura que la interfaz de usuario esté sincronizada con la última información de usuarios disponible en el backend.

Listing 5. Users.js

```
/*
* Copyright 2015 Anton Tananaev (anton@traccar.org)
*
* This program is free software: you can
    redistribute it and/or modify
* it under the terms of the GNU General Public
    License as published by
* the Free Software Foundation, either version 3 of
    the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it
    will be useful,
* but WITHOUT ANY WARRANTY; without even the implied
     warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General
     Public License
* along with this program. If not, see <http://www.
    gnu.org/licenses/>.
*/
Ext.define('Traccar.store.Users', {
    extend: 'Ext.data.Store',
    model: 'Traccar.model.User',
    proxy: {
        type: 'rest',
        url: 'api/users',
        writer: {
            writeAllFields: true
        }
    }
});
```

F. UserAttributes.js

El archivo `UserAttributes.js` define una tienda de datos (*data store*) en el marco de Ext JS, específicamente diseñada para gestionar un conjunto de atributos de usuario en la interfaz de usuario de Traccar. Estos atributos están relacionados con la configuración del sistema de correo electrónico (SMTP) y se utilizan para personalizar cómo los usuarios interactúan con el servicio de correo dentro del sistema.

1) Descripción General:

- **Ext JS Data Store:** La clase `Traccar.store.UserAttributes` extiende de `Ext.data.Store`, lo que significa que hereda todas las capacidades de manejo de datos de Ext JS. Esta tienda maneja atributos conocidos (`KnownAttribute`) que están predefinidos y relacionados principalmente con la configuración del correo SMTP.
- **Modelo de Datos:** La tienda utiliza el modelo `Traccar.model.KnownAttribute`, lo que indica que cada atributo gestionado por esta tienda sigue la estructura definida en ese modelo.
- **Datos Predefinidos:** La tienda viene preconfigurada con un conjunto de datos estáticos que representan diferentes configuraciones de correo SMTP. Cada objeto en el *array* `data` representa un atributo con las siguientes propiedades:
  - `key`: La clave del atributo, que se corresponde con una propiedad de configuración específica.
  - `name`: Un nombre descriptivo para el atributo, probablemente utilizado para mostrar en la interfaz.
  - `valueType`: El tipo de valor que acepta el atributo (`string`, `number`, `boolean`).
  - `allowDecimals`: Para atributos numéricos, indica si se permiten decimales.
  - `minValue` y `maxValue`: Define los límites para los valores numéricos.

2) Funcionalidades Principales:

- **Gestión de Atributos de Usuario:** La tienda `UserAttributes` gestiona un conjunto de atributos específicos relacionados con la configuración del sistema de correo electrónico SMTP. Estos atributos pueden ser usados para personalizar cómo los usuarios configuran y utilizan los servicios de correo dentro del sistema.
- **Configuración de Correo SMTP:** Proporciona una forma estructurada de manejar las configuraciones de correo SMTP como el servidor, el puerto, las opciones de seguridad, las credenciales, etc.

3) Usos Comunes:

- **Configuración de Correo Electrónico:** Esta tienda se utiliza para gestionar las configuraciones relacionadas con el correo electrónico dentro de la interfaz de usuario, permitiendo que los usuarios ajusten opciones como el servidor SMTP, la seguridad, y las credenciales.
- **Personalización de Atributos:** Permite a los usuarios personalizar y guardar configuraciones específicas de correo que se aplicarán a sus cuentas dentro del sistema.

Listing 6. UserAttributes.js

```
/*
* Copyright 2017 Anton Tananaev (anton@traccar.org)
* Copyright 2017 Andrey Kunitsyn (andrey@traccar.org
    )
*
* This program is free software: you can
    redistribute it and/or modify
* it under the terms of the GNU General Public
    License as published by
* the Free Software Foundation, either version 3 of
    the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it
    will be useful,
* but WITHOUT ANY WARRANTY; without even the implied
     warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General
     Public License
* along with this program. If not, see <http://www.
    gnu.org/licenses/>.
*/
Ext.define('Traccar.store.UserAttributes', {
    extend: 'Ext.data.Store',
    model: 'Traccar.model.KnownAttribute',
    data: [{
        key: 'mail.smtp.host',
        name: Strings.attributeMailSmtpHost,
        valueType: 'string'
    }, {
        key: 'mail.smtp.port',
        name: Strings.attributeMailSmtpPort,
        valueType: 'number',
        allowDecimals: false,
        minValue: 1,
        maxValue: 65535
    }, {
        key: 'mail.smtp.starttls.enable',
        name: Strings.
            attributeMailSmtpStarttlsEnable,
        valueType: 'boolean'
    }, {
        key: 'mail.smtp.starttls.required',
        name: Strings.
            attributeMailSmtpStarttlsRequired,
        valueType: 'boolean'
    }, {
        key: 'mail.smtp.ssl.enable',
        name: Strings.attributeMailSmtpSslEnable,
        valueType: 'boolean'
    }, {
        key: 'mail.smtp.ssl.trust',
        name: Strings.attributeMailSmtpSslTrust,
        valueType: 'string'
    }, {
        key: 'mail.smtp.ssl.protocols',
        name: Strings.attributeMailSmtpSslProtocols,
        valueType: 'string'
    }, {
        key: 'mail.smtp.from',
        name: Strings.attributeMailSmtpFrom,
        valueType: 'string'
    }, {
        key: 'mail.smtp.auth',
```

```
        name: Strings.attributeMailSmtpAuth,
        valueType: 'boolean'
    }, {
        key: 'mail.smtp.username',
        name: Strings.attributeMailSmtpUsername,
        valueType: 'string'
    }, {
        key: 'mail.smtp.password',
        name: Strings.attributeMailSmtpPassword,
        valueType: 'string'
    }]
});
```

G. UserRestrictions.js

La interfaz `UserRestrictions` define cinco métodos que deben ser implementados por cualquier clase que la utilice. Estos métodos proporcionan una forma estándar de consultar las restricciones aplicadas a un usuario en particular.

1) Métodos Definidos:

- **boolean getReadonly():** Este método devuelve un valor booleano que indica si el usuario tiene permisos de solo lectura. Si devuelve `true`, el usuario no puede realizar modificaciones en el sistema.
- **boolean getDeviceReadonly():** Indica si los dispositivos asociados con el usuario son de solo lectura. Si devuelve `true`, el usuario no puede modificar los dispositivos (por ejemplo, agregar, eliminar o editar dispositivos).
- **boolean getLimitCommands():** Indica si el usuario tiene restricciones en cuanto al envío de comandos. Un valor `true` sugiere que hay un límite en el tipo o número de comandos que el usuario puede enviar a los dispositivos.
- **boolean getDisableReports():** Indica si los informes están deshabilitados para el usuario. Si es `true`, el usuario no puede generar o acceder a ciertos informes dentro del sistema.
- **boolean getFixedEmail():** Indica si el correo electrónico del usuario está fijado, es decir, si el usuario no puede modificar su dirección de correo electrónico. Un valor `true` significa que la dirección de correo es inmutable.

2) Usos Comunes:

- **Implementación en Clases de Usuarios:** Las clases que implementan la interfaz `UserRestrictions` están obligadas a definir estos métodos, lo que permite al sistema aplicar de manera uniforme las restricciones de usuario en diferentes partes de la aplicación.
- **Consulta de Restricciones:** Esta interfaz es útil para consultar de forma estandarizada las restricciones de un usuario, lo que permite a otras partes del sistema tomar decisiones basadas en estas restricciones.

Listing 7. UserRestrictions.js

```
/*
* Copyright 2022 Anton Tananaev (anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (
    the "License");
* you may not use this file except in compliance
    with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in
    writing, software
* distributed under the License is distributed on an
    "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
    either express or
* implied.
* See the License for the specific language
    governing permissions and
* limitations under the License.
*/
package org.traccar.model;

public interface UserRestrictions {
    boolean getReadonly();
    boolean getDeviceReadonly();
    boolean getLimitCommands();
    boolean getDisableReports();
    boolean getFixedEmail();
}
```

H. UserSecurityContext.java

La clase `UserSecurityContext` encapsula la lógica necesaria para proporcionar información sobre el usuario autenticado y su contexto de seguridad en una aplicación. Se basa en un objeto `UserPrincipal`, que representa la identidad del usuario autenticado. La implementación de la interfaz `SecurityContext` permite que otras partes de la aplicación puedan acceder y verificar la información de autenticación y seguridad del usuario.

1) Funcionalidades Principales:

- **Proporcionar la Identidad del Usuario:** La clase almacena un objeto `UserPrincipal` que contiene la identidad del usuario autenticado. El método `getUserPrincipal()` devuelve este objeto, permitiendo que otros componentes del sistema puedan acceder a la identidad del usuario.
- **Verificar Roles de Usuario:** El método `isUserInRole(String role)` devuelve siempre `true` en esta implementación, lo que sugiere que en este contexto particular no se está utilizando una verificación detallada de roles. Sin embargo, en implementaciones más complejas, este método podría ser utilizado para verificar si un usuario pertenece a un rol específico.
- **Determinar si la Conexión es Segura:** El método `isSecure()` devuelve `false`, indicando que esta implementación no evalúa si la conexión es segura (por ejemplo, si está utilizando HTTPS). En un entorno de producción, este método podría ser extendido para verificar esta información.
- **Esquema de Autenticación:** El método `getAuthenticationScheme()` devuelve `BASIC_AUTH`, lo que sugiere que el esquema de autenticación utilizado es Basic Authentication, un método estándar para autenticar usuarios en HTTP mediante el envío de credenciales codificadas en base64.

2) Métodos:

- **Constructor UserSecurityContext(UserPrincipal principal):** Inicializa la clase con un objeto `UserPrincipal`, que contiene la identidad del usuario autenticado.
- **getUserPrincipal():** Devuelve el objeto `UserPrincipal`, que representa la identidad del usuario autenticado.
- **isUserInRole(String role):** Devuelve `true`, lo que indica que en esta implementación particular no se realiza una verificación específica de roles.
- **isSecure():** Devuelve `false`, lo que sugiere que la conexión no es evaluada como segura en esta implementación.
- **getAuthenticationScheme():** Devuelve `BASIC_AUTH`, indicando que el esquema de autenticación utilizado es Basic Authentication.

3) Usos Comunes:

- **Contexto de Seguridad en Solicitudes HTTP:** `UserSecurityContext` se utiliza para manejar la autenticación y proporcionar información sobre el contexto de seguridad en solicitudes HTTP dentro del sistema Traccar.
- **Autenticación de Usuarios:** Proporciona una forma estándar de autenticar usuarios y verificar su identidad dentro del sistema, utilizando un esquema de autenticación básico.
- **Autorización Basada en Roles:** Aunque simplificada en esta implementación, podría ser extendida para manejar autorizaciones basadas en roles en sistemas más complejos.

Listing 8. UserSecurityContext.java

```
/*
* Copyright 2015 - 2022 Anton Tananaev (
    anton@traccar.org)
*
* Licensed under the Apache License, Version 2.0 (
    the "License");
* you may not use this file except in compliance
    with the License.
* You may obtain a copy of the License at
*
* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in
    writing, software
* distributed under the License is distributed on an
     "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
    either express or
* implied.
* See the License for the specific language
    governing permissions and
* limitations under the License.
*/
package org.traccar.api.security;

import jakarta.ws.rs.core.SecurityContext;
import java.security.Principal;

public class UserSecurityContext implements
    SecurityContext {
    private final UserPrincipal principal;

    public UserSecurityContext(UserPrincipal
        principal) {
        this.principal = principal;
    }

    @Override
    public Principal getUserPrincipal() {
        return principal;
    }

    @Override
    public boolean isUserInRole(String role) {
        return true;
    }

    @Override
    public boolean isSecure() {
        return false;
    }

    @Override
    public String getAuthenticationScheme() {
        return BASIC_AUTH;
    }
}
```

I. UserController.js

`UserController` es un controlador de la vista (view controller) dentro del marco de Ext JS, utilizado para gestionar eventos y lógica de negocios relacionados con la gestión de usuarios en la interfaz de usuario. Esta clase maneja la habilitación de ciertos campos del formulario según el tipo de usuario, realiza pruebas de notificaciones y gestiona la operación de guardar los datos del usuario, ya sea creando un nuevo usuario o actualizando uno existente.

1) Funcionalidades Principales:

- **Inicialización del Controlador (`init`):** El método `init` se ejecuta cuando se carga la vista asociada al controlador. Este método se encarga de habilitar o deshabilitar ciertos campos del formulario de usuario en función de si el usuario actual es un administrador o si está editando sus propios datos.
  - **Campos afectados:** `adminField`, `deviceLimitField`, `userLimitField`, `readonlyField`, `disabledField`, `expirationTimeField`, `deviceReadonlyField`, `limitCommandsField`, `disableReportsField`.
- **Prueba de Notificaciones (`testNotification`):** El método `testNotification` envía una solicitud POST al endpoint `api/notifications/test` para probar las notificaciones del sistema. Si la solicitud falla, se muestra un error al usuario.
- **Guardado de Datos de Usuario (`onSaveClick`):** El método `onSaveClick` maneja el evento de guardar los cambios realizados en el formulario de usuario. Actualiza el registro del usuario con los valores del formulario y, dependiendo de si el registro es nuevo o existente, lo guarda directamente o lo agrega a la tienda de usuarios antes de sincronizar los cambios con el servidor.

- **Flujo del método:**
  1) Actualiza el registro del formulario.
  2) Si el registro es el usuario actual, lo guarda directamente.
  3) Si es un nuevo registro (`phantom`), lo añade a la tienda `Users`.
  4) Sincroniza la tienda con el servidor y maneja posibles errores.

2) Usos Comunes:

- **Gestión de Formularios de Usuario:** Este controlador se utiliza para gestionar la lógica detrás de los formularios de usuario en la interfaz de usuario de Traccar, controlando la habilitación de campos y la operación de guardado.
- **Pruebas de Configuración de Notificaciones:** Permite a los administradores probar las configuraciones de notificaciones del sistema directamente desde la interfaz de usuario.
- **Sincronización de Datos con el Backend:** Maneja la lógica de enviar los datos de usuario al backend para guardarlos o actualizarlos, asegurando que la interfaz esté sincronizada con el estado del servidor.

Listing 9. UserController.js

```
/*
* Copyright 2015 - 2017 Anton Tananaev (
    anton@traccar.org)
*
* This program is free software: you can
    redistribute it and/or modify
* it under the terms of the GNU General Public
    License as published by
* the Free Software Foundation, either version 3 of
    the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it
    will be useful,
* but WITHOUT ANY WARRANTY; without even the implied
     warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR
    PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General
     Public License
* along with this program. If not, see <http://www.
    gnu.org/licenses/>.
*/
Ext.define('Traccar.view.dialog.UserController', {
    extend: 'Traccar.view.dialog.MapPickerController
        ',
    alias: 'controller.user',

    init: function () {
        if (Traccar.app.getUser().get('administrator
            ')) {
            this.lookupReference('adminField').
                setDisabled(false);
            this.lookupReference('deviceLimitField')
                .setDisabled(false);
            this.lookupReference('userLimitField').
                setDisabled(false);
        }
        if (Traccar.app.getUser().get('administrator
            ') ||
            !this.getView().selfEdit) {
            this.lookupReference('readonlyField').
                setDisabled(false);
            this.lookupReference('disabledField').
                setDisabled(false);
            this.lookupReference('
                expirationTimeField').setDisabled(
                false);
            this.lookupReference('
                deviceReadonlyField').setDisabled(
                false);
            this.lookupReference('limitCommandsField
                ').setDisabled(false);
            this.lookupReference('
                disableReportsField').setDisabled(
                false);
        }
    },

    testNotification: function () {
        Ext.Ajax.request({
            url: 'api/notifications/test',
            method: 'POST',
            failure: function (response) {
                Traccar.app.showError(response);
            }
        });
    },

    onSaveClick: function (button) {
        var dialog, record, store;
        dialog = button.up('window').down('form');
        dialog.updateRecord();
        record = dialog.getRecord();
        if (record === Traccar.app.getUser()) {
            record.save();
        } else {
            store = Ext.getStore('Users');
            if (record.phantom) {
                store.add(record);
            }
            store.sync({
                failure: function (batch) {
                    store.rejectChanges();
                    Traccar.app.showError(batch.
                        exceptions[0].getError().
                        response);
                }
            });
        }
        button.up('window').close();
    }
});
```

J. UserReportStyles.js

El archivo `UserReportStyles.js` define un conjunto de estilos personalizados utilizando `makeStyles` de la biblioteca `@mui/styles` en el marco de Material-UI (MUI). Estos estilos están diseñados para aplicarse a los componentes relacionados con la visualización de informes de usuarios en la interfaz de usuario, probablemente en un entorno de React.

1) Uso de `makeStyles`: `makeStyles` es una función de `@mui/styles` que permite crear un conjunto de estilos CSS-in-JS que pueden aplicarse a los componentes de React. Estos estilos se generan dinámicamente y pueden incluir referencias al tema (`theme`) de Material-UI para mantener la consistencia visual en la aplicación.

2) Estilos Definidos: Este archivo define estilos para varios contenedores y elementos clave en la interfaz de usuario relacionada con los informes de usuarios. Cada clase de estilo está destinada a un propósito específico, como la organización de los contenedores principales, la cabecera, las columnas de acción, los filtros y los gráficos.

- **container:** Define un contenedor que ocupa el 100% de la altura disponible y utiliza un diseño de columna flexible.
- **containerMap:** Estilo para un contenedor que tiene un tamaño base del 40% de su espacio disponible, y no se reduce si el espacio disponible se contrae.
- **containerMain:** Estilo para un contenedor principal con desplazamiento (`overflow: 'auto'`), lo que permite que el contenido sea desplazable si es más grande que el área visible.
- **header:** Estilo para la cabecera, que es de posición fija (pegajosa) a la izquierda, con un diseño flexible en columna y estirado para alinear sus elementos.
- **columnAction:** Estilo para una columna de acción con un ancho mínimo (1%), asegurando que ocupe solo el espacio necesario, y con un padding a la izquierda definido por el espaciado del tema (`theme.spacing(1)`).
- **filter:** Define un contenedor para los filtros, con un diseño flexible en línea y envolvente (`flexWrap: 'wrap'`). Utiliza un espacio (`gap`) y padding definidos por el tema.
- **filterItem:** Estilo para un elemento de filtro, que ocupa el espacio flexible disponible según la configuración del tema.
- **filterButtons:** Define un contenedor para los botones de filtro, con un diseño flexible y un espacio (`gap`) entre los botones.
- **filterButton:** Estilo para un botón de filtro que ocupa todo el espacio disponible en su contenedor (`flexGrow: 1`).
- **chart:** Define un contenedor para gráficos, que crece para ocupar el espacio disponible y oculta el desbordamiento del contenido.

3) Usos Comunes:

- **Aplicación en Componentes de React:** Los estilos definidos en `UserReportStyles.js` se aplican a los componentes de React que gestionan la visualización de informes de usuarios. Estos estilos aseguran que la interfaz sea coherente, flexible y responsiva.
- **Personalización Basada en el Tema:** Los estilos aprovechan el tema de Material-UI, lo que permite que se adapten automáticamente a las variaciones del tema (como el modo oscuro o claro).

Listing 10. UserReportStyles.js

```
import { makeStyles } from '@mui/styles';

export default makeStyles((theme) => ({
    container: {
        height: '100%',
        display: 'flex',
        flexDirection: 'column',
    },
    containerMap: {
        flexBasis: '40%',
        flexShrink: 0,
    },
    containerMain: {
        overflow: 'auto',
    },
    header: {
        position: 'sticky',
        left: 0,
        display: 'flex',
        flexDirection: 'column',
        alignItems: 'stretch',
    },
    columnAction: {
        width: '1%',
        paddingLeft: theme.spacing(1),
    },
    filter: {
        display: 'inline-flex',
        flexWrap: 'wrap',
        gap: theme.spacing(2),
        padding: theme.spacing(3, 2, 2),
    },
    filterItem: {
        minWidth: 0,
        flex: `1 1 ${theme.dimensions.
            filterFormWidth}`,
    },
    filterButtons: {
        display: 'flex',
        gap: theme.spacing(1),
        flex: `1 1 ${theme.dimensions.
            filterFormWidth}`,
    },
    filterButton: {
        flexGrow: 1,
    },
    chart: {
        flexGrow: 1,
        overflow: 'hidden',
    },
}));
```

IV. Conclusiones y Discusión

El desarrollo y documentación de la API de gestión de usuarios en la arquitectura Traccar ha permitido la creación de un sistema robusto y escalable, capaz de gestionar de manera eficiente las operaciones relacionadas con usuarios, como la autenticación, la autorización, y la asignación de roles. La integración de tecnologías de vanguardia como JSON Web Tokens (JWT) para la autenticación y Google Authenticator para la generación de contraseñas de un solo uso (TOTP) ha añadido una capa significativa de seguridad a la API, mitigando riesgos comunes como el acceso no autorizado y los ataques de intermediario.

A. Discusión

1) Desafíos en la Implementación de Seguridad: Durante el desarrollo, uno de los principales retos fue la correcta implementación de mecanismos de autenticación y autorización. La integración de JWT permitió asegurar las comunicaciones entre el cliente y el servidor; sin embargo, la gestión de permisos para usuarios con roles específicos representó un desafío adicional, especialmente en entornos con múltiples microservicios.

La adición de TOTP aumentó significativamente la seguridad, pero también introdujo complejidad en la gestión de usuarios. A pesar de este reto, el uso de Google Authenticator proporcionó una solución eficiente y fácil de integrar, lo que mejoró la autenticación multifactor.

2) Documentación y Mantenimiento: La creación de una documentación exhaustiva utilizando Swagger facilitó tanto el desarrollo como la interacción entre los diferentes miembros del equipo y futuros desarrolladores. Además, esta documentación permite una escalabilidad continua del proyecto, proporcionando una referencia clara y actualizada sobre los endpoints disponibles y su correcto uso.

La implementación de pruebas unitarias y de integración garantizó que la API de usuarios cumpliera con los requisitos de funcionalidad, y permitió la rápida identificación y corrección de errores durante el proceso de desarrollo.

3) Escalabilidad y Rendimiento: Uno de los mayores logros de este proyecto fue el diseño de una arquitectura escalable. La modularidad de la API, junto con el uso de contenedores Docker, permite un despliegue eficiente en entornos productivos y asegura que el sistema puede manejar un número creciente de usuarios sin comprometer el rendimiento.

B. Consideraciones Futuras

A futuro, se podría mejorar la API integrando mecanismos de monitoreo en tiempo real para detectar intentos de acceso no autorizados o actividades sospechosas de manera proactiva. También sería recomendable considerar la implementación de técnicas de encriptación más avanzadas para datos sensibles almacenados en la base de datos.

C. Conclusión

En resumen, el proyecto de la API de gestión de usuarios sobre la arquitectura Traccar ha demostrado ser una solución efectiva para la gestión de usuarios en sistemas distribuidos. Las estrategias implementadas para asegurar la autenticación y autorización de usuarios, combinadas con un enfoque modular en la arquitectura y una documentación sólida, han facilitado tanto el desarrollo como la escalabilidad futura del sistema. Sin embargo, la evolución continua del proyecto requerirá una atención constante a las mejoras en seguridad y rendimiento.

REFERENCIAS

[1] A. Rashid, A. Masood, and A. R. Khan, "Rc-aam: blockchain-enabled decentralized role-centric authentication and access management for distributed organizations," *Cluster Computing*, vol. 24, pp. 3551–3571, 2021.

[2] R. A. Al-Wadi and A. Maaita, "Authentication and role-based authorization in microservice architecture: A generic performance-centric design," *Journal of Advances in Information Technology*, vol. 14, pp. 758–768, 2023.

[3] P. Kamboj, S. Khare, and S. Pal, "User authentication using blockchain based smart contract in role-based access control," *Peer-to-Peer Networking and Applications*, vol. 14, pp. 2961–2976, 2021.

[4] C. Pasomsup and Y. Limpiyakorn, "Ht-rbac: A design of role-based access control model for microservice security manager," *2021 International Conference on Big Data Engineering and Education (BDEE)*, pp. 177–181, 2021.

[5] L. H. Pramono and Y. K. Y. Javista, "Firebase authentication cloud service for restful api security on employee presence system," *2021 4th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, pp. 1–6, 2021.