Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

IMPLEMENTACIÓ D'UNA PLATAFORMA DE GESTIÓ I VISUALITZACIÓ DE CIBERATACS

Joan Travé Gordillo

Director: Raül Roca Cànovas
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 20 de juny de 2021

1. Introducció 4

2. Motivació 6

3. Objectius 6

4. Anàlisi 7

4.1 Històries d’usuari 7

4.2 Model de domini 7

5. Disseny i Implementació 9

5.1 Back end 9

5.2 Front end 22

5.3 Securització de la aplicació 27

5.4 Contenidors 29

6. Conclusions 43

7. Bibliografia 44

1. Introducció

Resum

En aquest treball de final de grau s'ha desenvolupat una plataforma per a desar i visualitzar ciberatacs, així com per a gestionar múltiples organitzacions en dita matèria. Sobre la visualització, aquesta s'ha fet mitjançant la projecció de les coordenades geogràfiques estratègiques - oficines, que són el destí dels atacs, i el mateix origen dels atacs - sobre dos mapes dinàmics, a partir de la llibreria *Leaflet* de representació de dades geogràfiques per a Javascript.

La responsabilitat de la securització de la API de recursos ha quedat repartida entre la mateixa API, que s'encarrega de diferenciar quins usuaris poden accedir a cada recurs, i l'API que proveu tokens, de manera que els usuaris del projecte no queden lligats a la API de recursos, i d'extendre's el conjunt de projectes de la hipotètica empresa que ha desenvolupat aquest treball, els usuaris podrien utilitzar els nous projectes amb el mateix compte que haurien fet servir per aquest.

S'ha realitzat una comparativa de tecnologies, que inclouen bases de dades, *frameworks* per a front end, back end, i contenidors, i s'ha acabat explorant les bases de dades MongoDB, PostgreSQL - així com el seu ecosistema de desenvolupament per a Windows -, el *framework* React per a javascript, el *framework* Flask per a Python, l'eina de documentació Swagger, el tipus de token bearer token i el contenidor Docker.

Com a context, s'ha après de la tipologia principal de ciberatacs en línia, així com les estratègies més comuns.

Resumen

En este trabajo de final de grado se ha desarrollado una plataforma para guardar y visualizar ciberataques, así como para gestionar múltiples organizaciones en dicha materia. Sobre la visualización, esta se ha hecho mediante la proyección de las coordenadas geográficas estratégicas - oficinas, que son el destino de los ataques, y el mismo origen de los ataques - sobre dos mapas dinámicos, a partir de la librería *Leaflet* de representación de datos geográficos para Javascript.

La responsabilidad de la securización de la API de recursos ha quedado repartida entre la misma API, que se encarga de diferenciar qué usuarios pueden acceder a cada recurso, y la API que provee tokens, de manera que los usuarios del proyecto no quedan atados a la API de recursos, y en caso de extenderse el conjunto de proyectos de la hipotetica empresa que ha desarrollado este trabajo, los usuarios podrían usar los nuevos proyectos con la misma cuenta que han creado para este.

Se ha realizado una comparativa de tecnologias, que incluyen bases de datos, *frameworks* para front end, back end, y contenedores, y se ha acabado por explorar las bases de datos MongoDB y PostgreSQL - así como sus respectivos ecosistemas de desarrollo para

Windows -, el *framework* React para Javascript, el *framework* Flask para Python, la herramienta de documentación Swagger, el tipo de token bearer token y el contenedor Docker.

Como contexto, se ha aprendido la tipología principal de ciberataques en línea, así como las estrategias más comunes por parte de los atacantes.

Abstract

In this final degree work, a platform has been developed to store and visualize cyber-attacks, as well as to manage multiple organizations in this field. The visualization has been done by projecting the strategic geographical coordinates - offices, which are the destination of the attacks, and the origin of the attacks itself - on two dynamic maps, using the *Leaflet* library of geographical data representation for Javascript.

The responsibility for securing the resource API has been divided between the API itself, which is responsible for differentiating which users can access each resource, and the API that provides tokens, so that the users of the project are not tied to the resource API, and in case of extending the set of projects of the hypothetical company that has developed this project, users could use the new projects with the same account that they have created for this one.

A comparison of technologies, including databases, frameworks for front end, back end, and containers, has been made, and has ended up exploring the MongoDB and PostgreSQL databases - as well as their respective development ecosystems for Windows -, the React *framework* for Javascript, the Flask *framework* for Python, the Swagger documentation tool, the bearer token type and the Docker container.

As context, the main typology of online cyber-attacks was learned, as well as the most common strategies by attackers.

2. Motivació

La principal motivació per a la realització d'aquest treball és la manca de plataformes gratuites a la web que puguin oferir la oportunitat d'una gestió personalitzada per a cada organització.

Les principals webs d'una funcionalitat semblant són els Threat Map de les grans empreses de seguretat, com Kaspersky, Checkpoint o Fortinet. Però aquestes no permeten mostrar els atacs d'una organització concreta.

A més, no val a oblidar que aquest és un treball de final de grau, i l'aprenentatge ha de ser una motivació prioritària. En aquest cas, el context de ciberseguretat en un treball de disseny i implementació de software, ha servit per a explorar el concepte de ciberatacs, així com la seva tipologia i estratègies.

3. Objectius

Els objectius per aquest projecte són implementar una plataforma per a gestionar la seguretat d'organitzacions en termes de ciberatacs per la xarxa, que pugui permetre visualitzar a l'administrador de seguretat de la mateixa l'origen geogràfic dels atacs, així com el tipus de l'atac i el material que ha estat sensible als mateixos.

Sobre disseny, és un objectiu realitzar les decisions sobre l'arquitectura de la manera més justificada possible, per a donar valor a l'enginyeria del software, doncs al cap i a la fi, aquest és el treball de final de grau d'una enginyeria.

4. Anàlisi

L'únic actor de la web és l'usuari administrador, que té permisos complets sobre les organitzacions que administra, així com els atacs registrats per a alguna de les seves organitzacions.

4.1 Històries d'usuari

S'ha decidit utilitzar les històries d'usuari com a metodologia per a organitzar els requisits funcionals. Els seus principals beneficis són la seva flexibilitat, doncs són fàcils de canviar si canvia el requeriment, i la seva facilitat de redacció.

- R01: Jo com a administrador de l'organització vull visualitzar el tipus d'atac que està rebent la meva organització.
- R02: Jo com a administrador de l'organització vull distingir quines oficines de l'organització poden estar compromeses per un atac.
- R03: Jo com a administrador de l'organització vull distingir quins terminals de cada oficina poden estar afectats per un atac
- R04: Jo com a administrador de l'organització vull saber quins usuaris poden haver tingut accés al terminal que ha rebut l'atac.
- R05: Jo com a administrador de l'organització vull identificar l'origen geogràfic de l'atac.
- R06: Jo com a administrador de l'organització vull administrar una o més organitzacions.

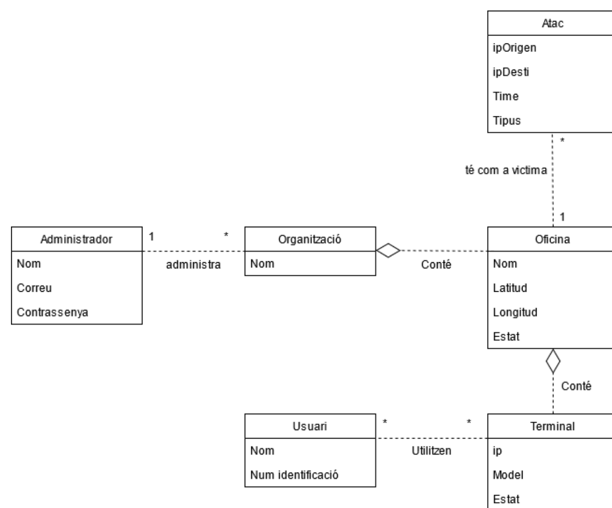
4.2 Model de domini

Basats en els requisits podem crear el diagrama del model de domini de la nostra aplicació. La primera classe que tindrem en consideració és la d'organització, que podem identificar amb un nom. De R02 se'n desprèn que de cada organització en depèn un llistat d'oficines, i de cadascuna d'aquestes, alhora, i a través de R03, en depèn un llistat de terminals. Per tant, oficina és una agregació a organització i terminal és una agregació a oficina. Cada terminal, per R04, pot haver estat usat per un nombre indeterminat d'usuaris, però seria un punt de vista més ampli, obrir la possibilitat de que un usuari pugui haver tingut accés a més d'un terminal. Així doncs, en aquest sentit, podem concloure que la relació entre usuaris i terminals és many-to-many.

De R01 s'entén la entitat Atac, que ha de ser d'un tipus, que com hem vist pot ser Exploit o Phishing, o en el cas que no encaixi en cap de les dues categories, malware genèric. Per R04, podem entendre que a més ha d'estar geolocalitzat. Per a geo localitzar un atac podem usar la latitud, així com la longitud d'on que prové. També podem fer servir la seva direcció IP, ho fem així. Un atac té com a destí una oficina i en concret una o varies terminals, és a dir, una altra direcció IP. Un atac només pot anar dirigit cap a una oficina segons aquests requisits, però una oficina pot ser víctima de un o més atacs. Així doncs, la relació entre oficina i atac és one-to-many.

Finalment, de R06 es dedueix que l'usuari de l'aplicació (no confondre amb l'usuari de les terminals d'una oficina) és l'administrador de diverses organitzacions. Podem identificar un administrador amb un nom, un mail i ja que serà usuari de l'aplicació, una paraula de pas. Aquí no s'especifica que una organització pugui ser administrada per més d'un administrador. Així, la relació entre les entitats administrador i organització serà one-to-many.

Així, amb tot, el corresponent diagrama quedaria de la següent forma:



5. Disseny i Implementació

5.1 Back end

5.1.1 Elecció base de dades

5.1.1.1 Requisits de la base de dades

Abans d'escollir la base de dades que usarem pel sistema, hem de definir quins requisits són importants per a la mateixa, i quins no ho seran tant.

Com que només hem definit a un administrador per organització, l'accés a la informació de les organitzacions serà linealment proporcional al nombre d'administradors registrats a la web.

Per cada organització hem definit només la propietat de nom, però fora interessant per a futures iteracions que cada organització pugui desar informació també rellevant de les mateixes, però diferent de la resta.

En canvi, per als administradors només necessitem la informació per a autentificar-se, i aquesta ha de poder ésser accedida de manera ràpida en lectura, doncs és possible que haguem de consultar moltes vegades la identitat de les peticions a la API. Si comparem les expectatives de consultes de lectura d'administradors en comparació amb les d'escriptura (registres a la plataforma), presumiblement el nombre de lectures serà molt superior.

Aquesta situació no es pot estendre als atacs. Si una organització és molt atacada, requerirà de moltes operacions d'escriptura a la base de dades, si un administrador té més d'una organització i aquestes tenen més d'una oficina, podem assumir que el nombre d'operacions d'escriptura d'atacs serà molt superior al nombre d'operacions d'escriptura d'administradors.

Com que d'aquest projecte no tenim previst treure benefici econòmic, el dèficit seria massa gran d'haver de fer front a una base de dades amb llicència de pagament, així doncs, cal que la base de dades sigui gratuïta.

Així, podem resumir els requisits:

- RBD01: Poder estendre els atributs de la classe organització i personalitzar-los independentment de la classe.
- RBD02: Alta velocitat de lectura per a administradors.
- RBD03: Alta velocitat d'escriptura per a la classe atacs.
- RBD04: Base de dades gratuïta.
- RBD05: Seguretat per als administradors.
- RBD06: Poder accedir mitjançant una API RESTful.

Ara que hem enumerat els principals requisits per a la base de dades, procedirem a fer un estudi de les diferents bases de dades al mercat i a comparar les característiques de les mateixes amb les nostres necessitats.

5.1.1.2 Comparativa de bases de dades

Si començem analitzant cada requisit, el primer que trobem és RBD01, segons el qual el poder d'estendre els atributs d'una classe ha de recaure en el mateix usuari. Entenent això, podem suposar una base de dades documental, on cada objecte està estructurat, però dita estructura no està tancada a la definició de cada classe.

Entre les principals bases de dades documentals podem trobar Redis, MongoDB, CouchDB i RavenDB. Procedim a comparar-les.

	Vel. Escriptura	Escalabilitat	Ús de memòria	Cost	Curva aprenentatge	Comunitat
Redis	Ràpida	Horitzontal	Alt	0	Fàcil	Estesa
MongoDB	Ràpida	Horitzontal	Molt alt	0	Fàcil	Estesa
CouchDB	Ràpida	Horitzontal	Alt	0	Fàcil	Poca
RavenDB	Ràpida	Horitzontal	Alt	789	Fàcil	Poca

Redis, però, té en contra que només es poden fer consultes a través de la clau, i no pas a través del contingut. En el cas d'atacs, per exemple, no podem fer consultes a través de les hores en les que s'han produït si aquestes formen part del valor desat. És per això que la hem de descartar.

En el cas de CouchDB i RavenDB, la seva poc numerosa comunitat dificulta enormement la resolució de problemes durant el desenvolupament. A més, RavenDB té cost per any, que entra en conflicte amb RBD04.

Per contra, MongoDB és molt intensiva en l'ús de memòria, però té al seu favor l'estesa comunitat que la suporta. Com que l'ús de memòria baix no és un requisit per l'*scope* d'aquest treball, ens decantarem per aquesta opció.

Tot i això, un altre requisit és la fiabilitat de les dades d'identificació, característica que cap base de dades documental ens aporta. Per aquest motiu ens decantarem pel model de bases de dades relacional.

Les bases de dades relacionals més habituals son MySQL, mariaDB, Oracle i PostgreSQL. De manera que procedirem a comparar-les de la mateixa forma que hem comparat les bases de dades documentals.

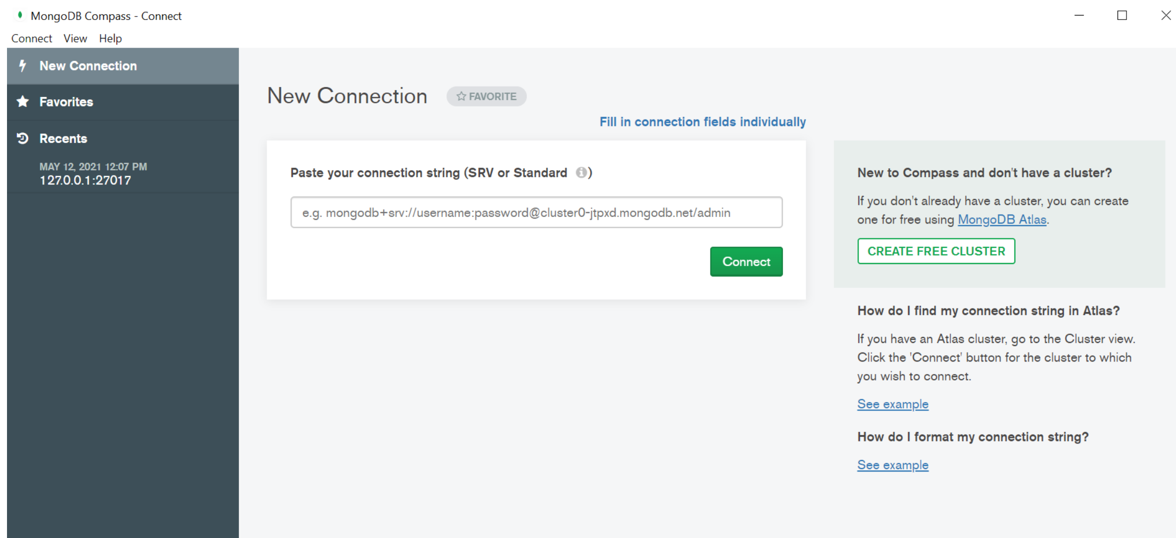
	Vel. Lectura	Escalabilitat	Ús de memòria	Cost	Curva aprenentatge	Comunitat
MySQL	Ràpida	Complexa	Alt	0	No	Estesa
MariaDB	Ràpida	Complexa	Alt	0	No	Estesa
PostgreSQL	Ràpida	Vertical	Alt	0	No	Estesa
Oracle	Ràpida	Vertical	Alt	150k	Alta	Estesa

Com podem apreciar, PostgreSQL és la opció més escalable entre les opcions sense cost. La curva d'aprenentatge de totes les opcions gratuïtes és nula perquè l'autor ja posseeix experiència amb totes elles.

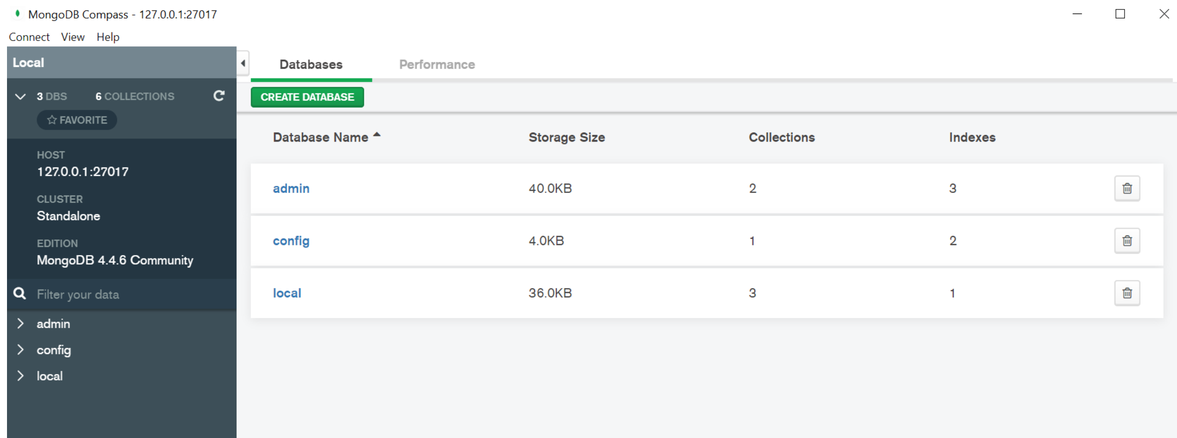
5.1.2 Ecosistema MongoDB

El software gestor de MongoDB que s'ha escollit per a aquest treball és l'eina oficial de Mongo, MongoDBCompass.

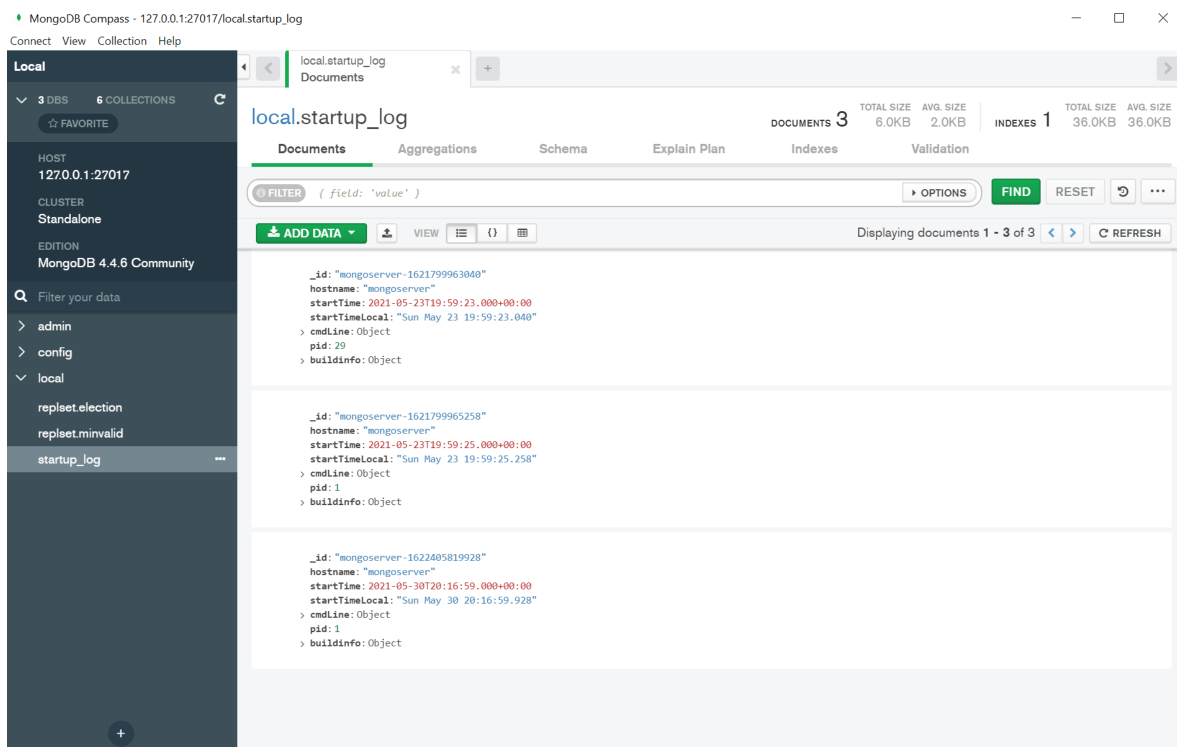
És un software amb GUI que permet gestionar diverses connexions simultàniament, i es pot estendre per a gestionar clusters sense cost extra.



Pantalla principal del software MongoDBCompass. A la dreta podem veure les connexions que ja tenim guardades. Al centre podem crear noves connexions.



Pantalla per a gestionar les bases de dades d'una connexió a MongoDBCompass. Per a cada base de dades podem veure les col·leccions desades, així com crear registres per cadascuna d'elles o crear noves col·leccions o bases de dades.



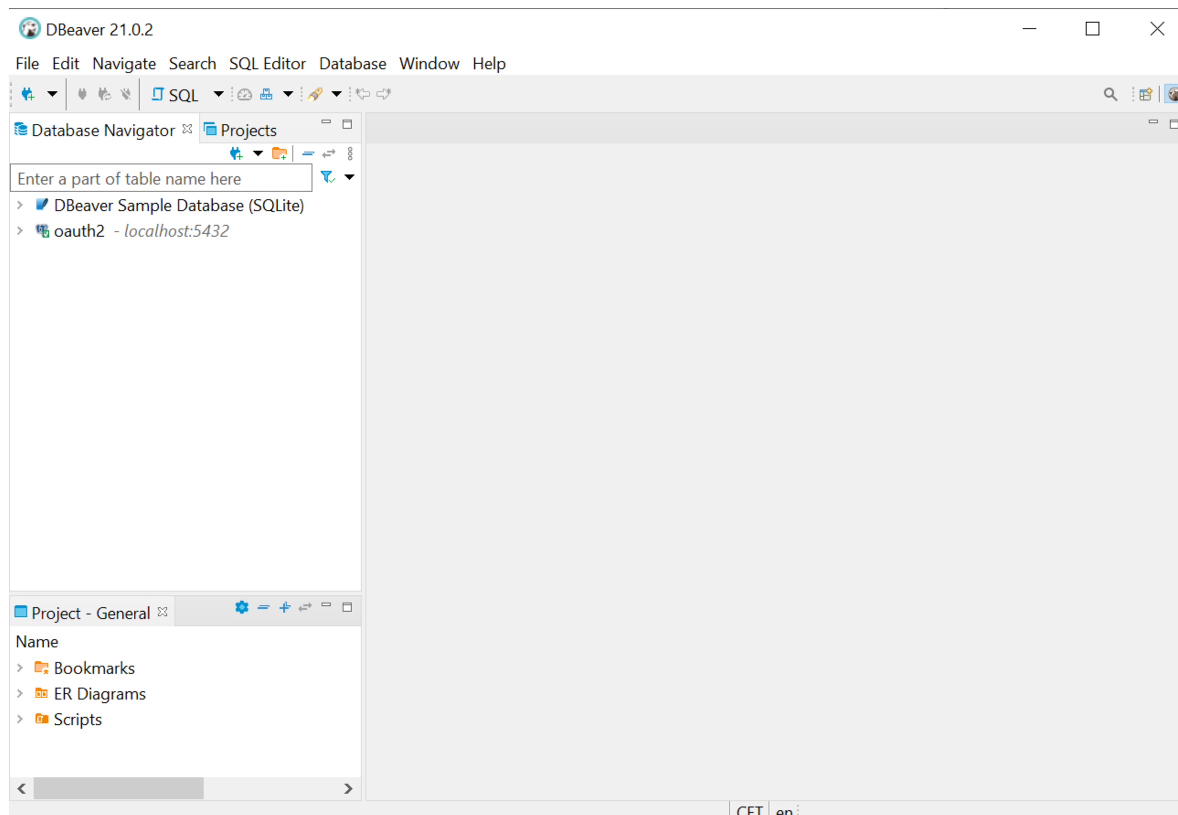
Col·lecció de logs de la base de dades local a MongoDB, podem observar els valors de cada document (objecte) de dita col·lecció, així com cercar-ne per valor al cercador.

5.1.3 Ecosistema PostgreSQL

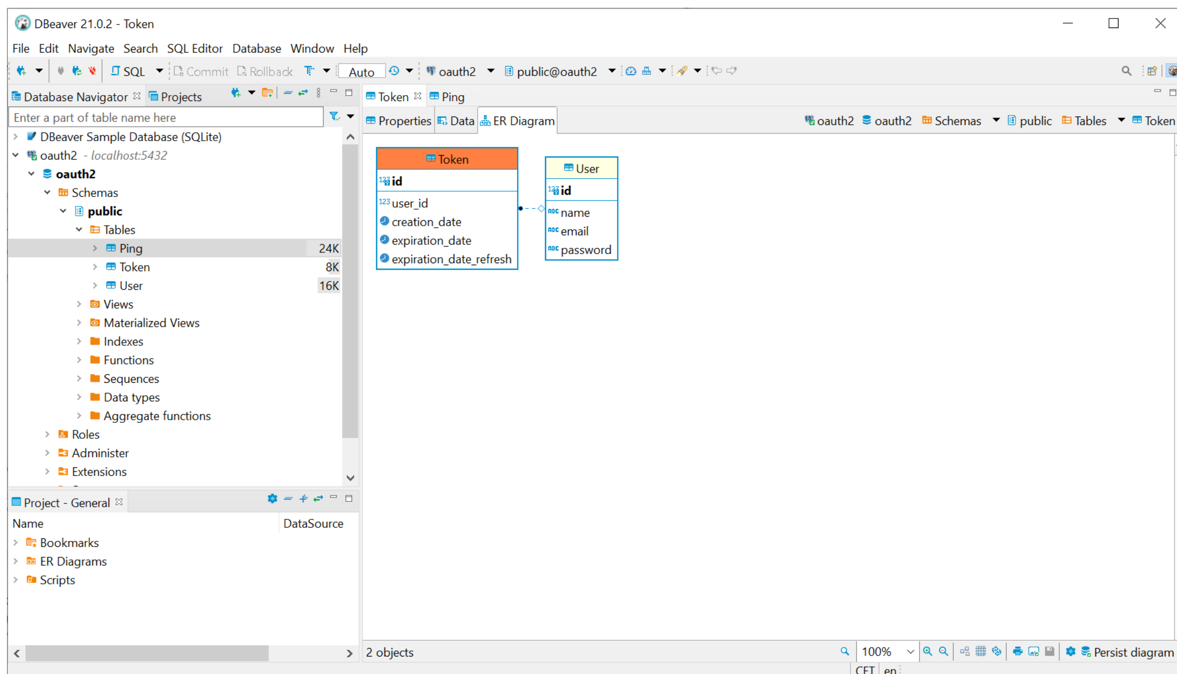
Per al sistema gestor de la base de dades postgresql, l'eina més extesa específica per a dita base de dades és pgAdmin, un software amb GUI al navegador a la seva versió 4. El programa, però, ha estat molt criticat per les seves pèrdues de connexió repentines, així com

els missatges criptics a l'hora de gestionar excepcions. Per això, s'ha usat l'alternativa amb suport per a multiples dades relacionals, el software DBeaver.

Entre les seves característiques destaquen el suport per múltiples connexions simultànies, poder editar valors de les taules a la mateixa vista que les consultes de lectura, sense necessitat de queries SQL, enllaç entre les taules a través de les claus foranes, memòria d'arxius SQL.



Pantalla principal del software DBeaver, podem veure les diverses bases de dades que gestionem, les URI's a cadascuna d'elles, així com el seu estat i comptem a una interfície d'usuari que ens dona la opció, entre d'altres, de crear una nova connexió o d'editar un fitxer SQL.



Pantalla del software DBeaver que permet veure el diagrama de classes de la taula seleccionada i de totes les que guarden claus foranes de manera recursiva. En aquest cas hem accedit a aquesta taula Token a través de navegar per l'arbre de fitxers que se'ns mostra a l'esquerra.

5.1.4 API Flask

5.1.4.1 Què és una API RESTful

Una API es pot definir de varies maneres, però a la pràctica, es tracta d'un conjunt de punts d'entrada accessibles pel client de dita API, però que el seu codi no es pot modificar per aquest.

REST és una arquitectura per API's que funcionen amb el protocol HTTP, definida per cinc principis:

1. Interfície uniforme: Els missatges han de ser autodescriptius, els recursos s'identifiquen a través de les URI i el client pot modificar els recursos sempre que tingui permisos.
2. Hi ha una separació entre client i servidor. El client no s'ha d'ocupar de l'emmagatzematge de les dades.
3. Sense estat: El servidor no guarda l'estat de la connexió amb el client.
4. Les respostes a les crides del servidor poden emmagatzemar en una caché per part del client.
5. S'obre la possibilitat a que el client i el servidor no actuin directament. Pot haver un hardware intermedi que gestioni les peticions entre aquests. Un exemple pot ser el

server d'Amazon Web Services, Load Balancer, que decideix a quina instància del servidor es dirigeix cada petició segons el volum d'aquestes que estigui administrant cadascuna.

Aquest treball està desenvolupat seguint l'arquitectura RESTful, amb un front end (client) i un back end (servidor). La tecnologia emprada per al back end és el framework Flask, que opera sobre el llenguatge de programació python.

5.1.4.2 Flask

Flask és un framework web, molt útil per a construir API's RESTful amb el llenguatge python de programació.

Respecte el seu principal competidor per al llenguatge python, Django, és un framework més lleuger, que no està limitat a una arquitectura model vista controlador, i amb una corba d'aprenentatge menys pronunciada. Com que per als recursos de l'aplicació no pensem seguir la arquitectura MVC, es tracta de l'opció escollida.

Entre les principals característiques de Flask, s'inclouen:

- Compatibilitat amb una arquitectura RESTful.
- Conté un servidor propi per a entorns de desenvolupament (la documentació oficial recomana usar el servidor Flask per a entorns de producció, doncs no disposa de multi-threading).
- Disposa de suport per a cookies.

Per a crear un servidor Flask només cal importar la clase Flask, instanciar-li i invocar el mètode *run* tal i com s'il·lustra en el següent exemple:

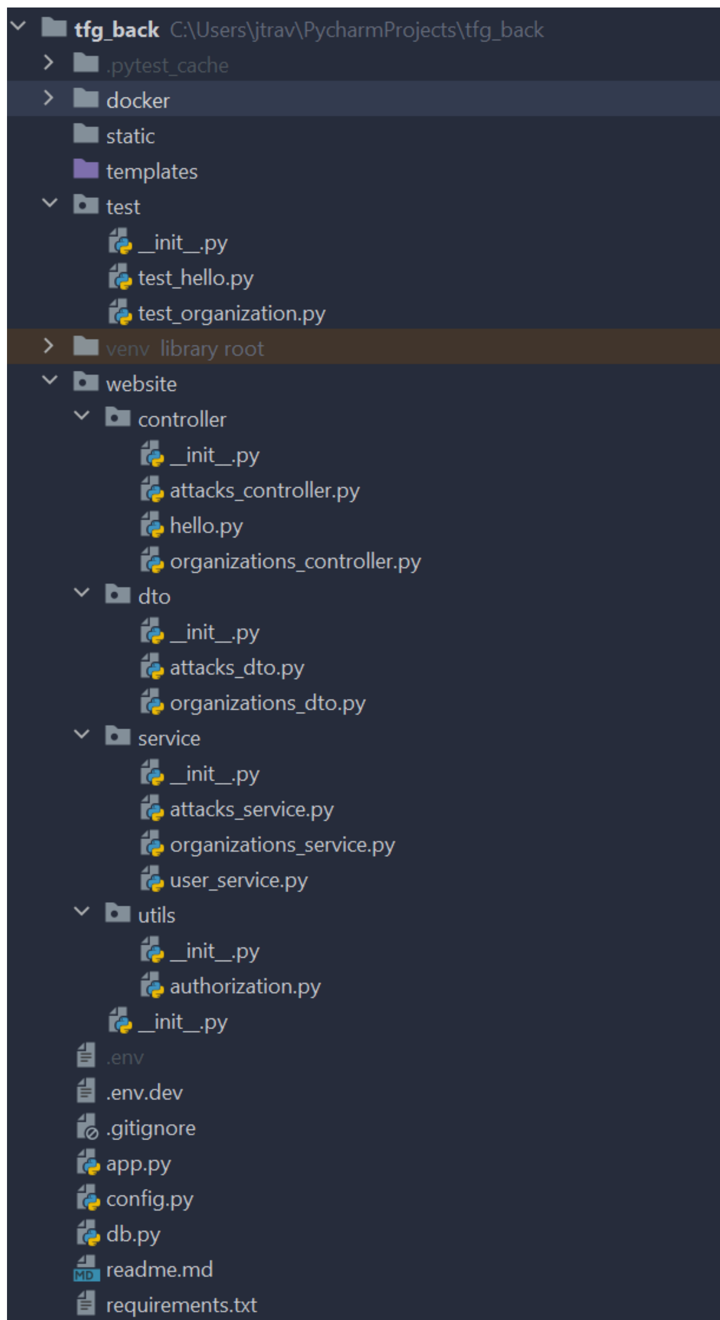
```
from flask import Flask


flask_app = Flask(__name__)

if __name__ == '__main__':
    flask_app.run(host='0.0.0.0')
```

En aquest exemple es mostra com iniciar un servidor de flask en quatre linies.

Per a executar-lo, només cal instal·lar el mòdul de python corresponent i executar el fitxer on es troba el codi. Per exemple: *app.py*:

```
pip install flask

python app.py
```

5.1.4.3 Arquitectura



Esquema de l'arquitectura del back end. El directori docker conté els fitxers necessaris per a crear els contenidors, que es veuran més endavant. El directori test, els test funcionals. El directori website, els scripts amb la lògica de l'aplicació. El subdirectori controller té la lògica de cada punt d'entrada; el subdirectori dto conté els models de dades que s'esperen als punts d'entrada, així com els models que el client ha d'esperar com a sortida; el subdirectori service les consultes a base de dades; el subdirectori utils té els decoradors d'autorització. El fitxer .env conté les variables d'entorn del servidor. El fitxer app.py és l'encarregat d'iniciar la aplciació; el fitxer config.py s'encarrega de desar les variables d'entorn al servidor, el fitxer db.py instancia la base de dades; i el fitxer requirements.txt és

un llistat dels mòduls necessaris per al funcionament de la app.

L'arquitectura escollida per al back end és la de dues capes: Controlador-Servei. Segons aquesta arquitectura, una primera capa de Controlador contindrà una definició de cada punt d'entrada per cada recurs, i la capa de servei s'encarregarà de realitzar les consultes pertinents a la base de dades, així com la lògica de cada recurs.

```
from flask import Flask
from flask_cors import CORS
from flask_restplus import Api

from config import Config, authorizations

flask_app = Flask(__name__)
CORS(flask_app)
flask_app = Config.init_app(flask_app)
app = Api(
    app=flask_app,
    authorizations=authorizations,
    security='apikey'
)

from website.controller.attacks_controller import api as attacks_ns
from website.controller.organizations_controller import api as organizations_ns

app.add_namespace(attacks_ns)
app.add_namespace(organizations_ns)

if __name__ == '__main__':
    app.app.run(host='0.0.0.0')
```

Fitxer app.py

Al nostre fitxer app.py, hem de crear l'objecte instància de Flask, per a continuació permetre la política de CORS (política per permetre l'accés al nostre servidor mitjançant crides HTTP i no només HTTPS), a continuació crearem un objecte Api, que ens permet definir els models d'entrada i sortida, que importem i associem a continuació (més detall al capítol dedicat a Swagger), i finalment executem el mètode *run*.

5.1.4.4 ORM

Una de les principals avantatges d'haver escollit una base de dades documental per al backend de l'aplicació, és el de no haver de definir els models de les col·leccions corresponents, de manera que podem gestionar la lògica de les mateixes en la capa de servei quan realitzem les consultes.

Abans, però, hem de crear una connexió amb la base de dades en qüestió. Donem un cop d'ull al nostre fitxer de variables d'entorn.

```
ENV=DEV
MONGO_DBNAME=attacks
MONGO_USER=myUserAdmin
MONGO_PASSWORD=supersecret
MONGO_HOST=mongoserver
MONGO_PORT=27017
STS_URI=http://tfg_oauth2:5000
MONGO_INITDB_ROOT_USERNAME="myUserAdmin"
MONGO_INITDB_ROOT_PASSWORD="supersecret"
MONGO_INITDB_DATABASE="admin"
```

Fitxer .env de variables d'entorn

Per a connectar-nos a la base de dades requerim de la URI corresponent, aquesta s'obté mitjançant les variables DBNAME, USER, PASSWORD, HOST i PORT.

```
class Config:

    def __init__(self):
        self.MONGO_DBNAME = env_conf('MONGO_DBNAME')
        self.MONGO_USER = env_conf('MONGO_USER')
        self.MONGO_PASSWORD = env_conf('MONGO_PASSWORD')
        self.MONGO_HOST = env_conf('MONGO_HOST')
        self.MONGO_PORT = env_conf('MONGO_PORT')
        self.MONGO_URI = f'mongodb://{self.MONGO_USER}:{self.MONGO_PASSWORD}@{self.MONGO_HOST}:{self.MONGO_PORT}/' \
                         f'{self.MONGO_DBNAME}?authSource=admin'
        self.STS_URI = env_conf('STS_URI')
```

Classe config, on podem veure la construcció de la URI corresponent a la base de dades a partir de les dades del fitxer de variables d'entorn.

Un cop construïda la URI, procedirem a crear el nostre client de mongoDB.

```
from flask_pymongo import MongoClient

from config import Config


mongo = MongoClient(Config().MONGO_URI)
```

Creació del client de MongoDB a partir del mòdul per a flask flask_pymongo a través de la URI que s'ha creat al constructor de la classe Config.

I un cop ja tenim feta la connexió, podem usar el objecte per a realitzar les consultes.

```python
def get_organization_by_name(name):
    organization = mongo.db.organizations.find_one({'name': name})
    return json.loads(json_util.dumps(organization))
```

Exemple d'operació find a mongoDB, en aquesta, a través de l'objecte mongo, el client de la bdd, accedim a la col·lecció organizations i en trobem una que coincideix amb el nom proporcionat. Finalment en retornem el json corresponent.

```python
def insert_organizations(data):
    if not data.get('name'):
        raise AttributeError('Not name')
    organization = get_organization_by_name(data.get('name'))
    if organization:
        raise ValueError('Organization already exists')
    organization_id = mongo.db.organizations.insert(data)
    new_organization = mongo.db.organizations.find_one({'_id': organization_id})
    return json.loads(json_util.dumps(new_organization))
```

Exemple d'inserció d'organització a la capa de servei, on a més de la crida ala base de dades ens encarreguem de la lògica corresponent, verificant els atributs obligatoris (name), llençant excepcions si ja existeix i retornant l'objecte en forma de json.

5.1.4.5 Swagger

Però la tasca del desenvolupament no acaba a la materialització de la lògica al codi, una feina tan important com programar és documentar el que s'ha escrit.

En el cas d'API's, una de les eines més utilitzades és swagger, que proveu d'un client per a navegador on es poden consultar, així com testejar, les crides als punts d'entrada que hem documentat.

Hi han diverses llibreries per a python que doten d'una interfície per a crear una documentació per a swagger, però aprofitant que estem usant el framework flask, escollirem la extensió oficial de swagger per a flask, inclosa en el paquet de python *flask_restplus*.

En primer lloc, definim els DTO per a cadascun dels objectes que esperem rebre o retornar a cada endpoint. Un DTO (Data Transfer Object) és un objecte que el seu únic objectiu és transportar dades entre clients i servidor. És una definició simple dels mateixos, on no s'utilitza gaire lògica.

En l'arquitectura del projecte, el mòdul destinat als DTO és al mateix nivell que la lògica de negoci, o les queries, en concret en el mòdul de nom *dto*. A més, separarem els fitxers destinats als objectes de transferència de dades per controladors. Veiem com a exemple el destinat a atacs.

```python
from flask_restplus import Namespace, fields


class AttackDto:
    attack_api = Namespace('attacks')
    attack_input = attack_api.model('Attack input', {
        "ipOrigen": fields.String,
        "ipDesti": fields.String,
        "origenLatitud": fields.Float,
        "origenLongitud": fields.Float,
        "organitzacioId": fields.String,
        "time": fields.DateTime,
        "type": fields.String
    })
    attacks_input = attack_api.model('Attacks input', {
        "data": fields.List(fields.Nested(attack_input))
    })
```

Podem observar a attack_input que el model corresponent al DTO conté la informació que vam definir al model de domini per a aquesta classe. A més hem definit un segon model, el attacks_input, que representa una llista de attacks, per a poder modelar el cas on volem transferir més d'un atac alhora.

Un cop hem definit els models que volem documentar per a un recurs, hem d'importar-los al controlador i crear una API per a cada recurs.

```python
from flask import jsonify, request
from flask_restplus import Resource

from website.dto.attacks_dto import AttackDto
from website.service.attacks_service import insert_attacks, attacks_query
from website.service.user_service import user_has_organization
from website.utils.authorization import login_required

api = AttackDto.attack_api
_attack_input = AttackDto.attacks_input
```

Capçalera del fitxer attacks_controller.py, on importem els DTO dels attacks i del espai de noms, on estan definits els models en fem la api del controlador.

I a continuació, després de crear l'api del controlador, cal documentar cada punt d'entrada. Això, de manera típica es fa especificant el format de l'entrada esperat, els requisits d'accés (la seguretat propia del punt d'accés) i la sortida que ha d'esperar el client.

```python
@api.route('/<organization_id>')
class AttacksOrganization(Resource):
    @api.response(201, 'Created', model=_attack_input)
    @api.response(400, 'Already exists')
    @api.expect(_attack_input)
    @api.doc(security='apikey')
    @login_required
    def post(self, organization_id, **kwargs):
        response = None
        data = request.json
        try:
            if not user_has_organization(kwargs.get('user').get('_id').get('$oid'), organization_id):
                response = jsonify({'message': 'User has no grants'})
                response.status_code = 403
            else:
                response = jsonify({'message': insert_attacks(organization_id, **data)})
                response.status_code = 201
        except ValueError as e:
            response = jsonify({'message': str(e)})
            response.status_code = 400
        finally:
            return response
```

Punt d'entrada del post al recurs d'atacs, on es documenta la ruta, aquesta ve donada per una barra i una variable que s'associa amb el paràmetre organization_id de la funció post. Es documenten les possibles respostes amb els codis HTTP 201, created, acompanyada amb un output amb el mateix format que el DTO llista d'atacs, i un codi 400 que significa que el recurs ja existeix. A més es documenta que s'espera el DTO d'entrada llista d'atacs, així com un model de seguretat que es veurà més endavant.

Finalment, cal importar la api de cada controlador i relacionar-la amb l'objecte de flask que executarà l'aplicació.

Per això cal crear una API que englobi tots els controladors i afegir-los com espais de noms.

```python
app = Api(
    app=flask_app,
    authorizations=authorizations,
    security='apikey'
)

from website.controller.attacks_controller import api as attacks_ns
from website.controller.organizations_controller import api as organizations_ns

app.add_namespace(attacks_ns)
app.add_namespace(organizations_ns)
```

Fragment del fitxer app.py on creem una API que executarà com a aplicació el nostre objecte flask, com es pot veure a la segona linia. A continuació s'importen les API de cada recurs i s'afegeixen a l'espai de noms de la API global.

5.2 Front end

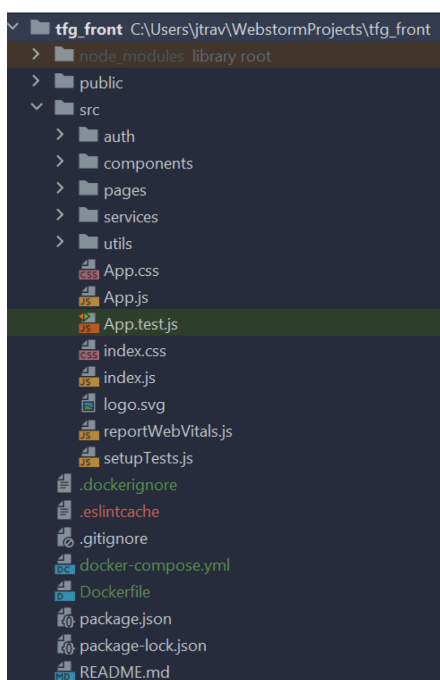
5.2.1 Frameworks

La banda del client, el front end, serà un projecte de javascript, però com a aplicació moderna, cal utilitzar un framework que permeti agilitzar el treball i reduir costos de temps i repetició de codi. Els frameworks més usats a javascript son Angular i React, tot i que Vue.js mostra una tendència cap a l'alça cada cop més pronunciada, no s'ha valorat la possibilitat de fer-lo servir en aquest projecte, doncs ja s'estudia en una assignatura de la mateixa facultat en el moment de redacció d'aquest treball.

Angular va ser llençat l'any 2010 per google, i es tracta d'un framework molt robust, especialitzat en aplicacions multi pàgina, usat per una comunitat molt àmplia i avalat per la empresa Google. Com a contras, podem mencionar la seva dura corba d'aprenentatge la obligació a desenvolupar en TypeScript en comptes de JavaScript.

React va ser llençat l'any 2012 per facebook, i es presenta com un framework amb una corba d'aprenentatge molt suau, així com una documentació extensa i un ecosistema compatible amb multitud de plugins. És ideal per a webs amb una sola pàgina. Com a punt en contra té el fet que precisament per a alleugerir el seu pes i delegar moltes de les tasques a plugins de tercers, el paquet base és incomplet per a la majoria de funcionalitats d'un client web.

Amb tot i tenint en compte que la nostra web només tindrà tres pàgines (login, signup, i mapa per a visualitzar els atacs), i a més l'autor no té prou experiència amb frameworks per a javascript i cap coneixement de typescript, s'ha decidit apostar per React.



Estructura del projecte amb React. A

la carpeta public desem les imatges estàtiques; a src el codi. A la carpeta auth tenim les funcions per a la securització del front end; a components desem els components que conformen cada pàgina, a pages la lògica de dites pàgines i a services les operacions per a cada recurs (atacs i organitzacions)

5.2.2 Estils

Pel que fa a l'apartat visual i d'estils, és habitual a les pàgines webs dedicades a la ciberseguretat els colors foscos com a principals, així com colors vius de secundàris. Això és degut a que s'associa el color negre amb l'autoritat, la maldat i la força, concepte que motiva o interessa a les persones que es dediquen a aquest àmbit.

5.2.3 Entorn per a projectar els mapes

Els requisits R02 i R05 del projecte, que s'identifiquen amb les necessitats de distingir a quines oficines s'han produït els atacs, i quin és el seu origen físic a través de la direcció IP de l'atacant, respectivament, desemboquen en un punt comú: la funcionalitat de poder visualitzar punts geogràfics de forma instantània. La solució més evident, i la implementada per la competència, és utilitzar un mapa dinàmic i interactiu per a que l'usuari, de manera simple, pugui accedir als seus recursos geo estratègics amb pocs clics i veure'n el detall.

En el nostre cas, els recursos a identificar geogràficament, és a dir, amb latitud i longitud definida en el sistema, són les oficines de l'organització. Però a més, ens hem d'enginyar un sistema per a que simultàniament es puguin visualitzar els atacs.

La proposta per a aquest repte és la de tindre dos mapes a pantalla, un de principal amb els recursos de l'usuari, interactiu per a que aquest pugui explorar de manera còmoda al voltant de la geografia mundial, amb el detall que li vulgui donar; i un de secundari, de tamany més reduït però què ens pugui servir per a identificar l'origen dels atacs.

5.2.3.1 Anàlisi d'entorns

Dues de les llibreries per a projecció de mapes més populars per a Javascript, i amb compatibilitat amb React son Mapbox i Leaflet. La primera, mapbox, privativa, és coneguda per la seva àmplia compatibilitat amb aplicacions mòbils, així com a clients de navegador. Aquesta característica, però, no serà gaire apreciada per a l'àmbit d'aquest treball, doncs no està definit cap requisit per a portar la aplicació a tecnologies mòbils ni ara ni en un futur. Leaflet, en canvi, de codi obert i mantinguda per una comunitat més que estable, destaca en el seu sistema de càrrega de mapes per caselles, imatges optimitzades per internet.

L'elecció d'utilitzar leaflet ve donada per la seva optimització per a navegadors, així com pel fet de ser gratuïta - mapbox deixa de ser-ho a partir de cinquanta mil càrregues mensuals -, i *open source*.

5.2.3.2 Leaflet

El funcionament de leaflet de la característica del mapa per caselles es resumeix de la següent manera:

- Es demana al servidor el mapa conformat per caselles.
- Quan el servidor detecta un canvi en el mapa identifica les caselles que es veuen afectades per aquest canvi.
- El servidor només envia al client

Pel que fa al seu ús a React, el primer pas és definir les dependències necessàries per al desenvolupament. Per això, editarem el fitxer *package.json*, i a la clau *dependencies* hi afegirem els paquets *leaflet, leaflet-curve* - per a dibuixar les corbes animades que representin l'origen i el destí d'un atac - i *react-leaflet*.

A continuació, podem importar les classes corresponents. Pel mapa principal, amb id *map_office* al codi, necessitarem la classe Marker, per a quan seleccionem una oficina, desplegar un modal amb la seva informació. Quan sigui seleccionada una oficina, a més, mostrarem un pop up amb el detall de l'atac que es pot estar produint.

Tant per al mapa principal - *map_office* -, com pel secundari- amb id *map_attack* -, necessitem la classe Map, que fa de contenidor per a la classe TileLayer, que conté el mapa de caselles, que haurem de referenciar, així com una sèrie de paràmetres per a la correcta visualització del mapa, com el zoom o el centre. Nosaltres ajustem dits paràmetres per a que es mostri el mapa principal de manera que ocupi tota la pantalla i el mapa secundari de manera que només ocupi un petit espai a la part inferior dreta de la pantalla, però prou gran per poder distingir els continents i que hi càpiguen les banderes dels països que realitzen l'atac.

Per a dibuixar les corbes és interessant l'ús de la llibreria *leaflet-curve*, que conté una interfície per a dibuixar corbes, donats dos punts del mapa i un punt mig. A més, la mateixa llibreria proveu d'una funció per ajustar la velocitat de l'animació, i així veure l'efecte de construcció de dita corba.

Com que el mapa secundari és petit en comparació amb el principal i per alguns pantalles pot ser que no es puguin distingir correctament les fronteres dels diferents països, s'ha utilitzat l'API de la pàgina web geonames.org, que a través d'una latitud i una longitud ens retornarà una icona amb la bandera del país.

5.2.4 Fitxers JSON d’entrada

5.2.4.1 Organització

Les dades definides per a l’organització són només el nom, però també cal pujar un esquema de la terminal i els usuaris, que incloem dins d’una clau anomenada data.

```
{
    "name": "OrganizationTest",
    "data": [
        {
            "Id": 0,
            "Nom": "Oficines Sabadell",
            "Adreça": "Av. De la Pau,120",
            "Municipi": "Sabadell",
            "Latitud": 41.553554,
            "Longitud": 2.067726,
            "Estat": "Sa",
            "Terminals": [
                {
                    "id": 0,
                    "ip": "88.369.45.2",
                    "model": "Toshiba",
                    "Estat": "Sa",
                    "users": [
                        {
                            "name": "Yuan",
                            "DNI": "12345678Q"
                        },
                        {
                            "name": "Alfredo",
                            "DNI": "78945612L"
                        },
                        {
                            "name": "Po",
                            "DNI": "222555888P"
                        }
                    ]
                },
```

Fitxer d’entrada per una organització, a data en desem les oficines, i a cada oficina, a Terminals desem les terminals de la mateixa i a users, per cada terminal, els usuaris que l’han usat.

5.2.4.2 Atacs

Els atacs els entrem con una llista d’objectes o cada objecte ha de tenir les dades definides al model de domini per a dita classe.

```json
{
    "data": [
        {
            "ipOrigen": "84.123.25.5",
            "ipDesti": "88.369.45.2",
            "origenLatitud": 33.83,
            "origenLongitud": 107.42,
            "nodeId": 0,
            "time": "08-02-2021 13:14:40",
            "type": "Exploit"
        },
        {
            "ipOrigen": "71.63.251.54",
            "ipDesti": "88.369.45.4",
            "origenLatitud": 44.33,
            "origenLongitud": -107.72,
            "nodeId": 1,
            "time": "08-02-2021 13:16:14",
            "type": "Exploit"
        },
        {
            "ipOrigen": "23.69.21.154",
            "ipDesti": "88.369.45.3",
            "origenLatitud": 25.08,
            "origenLongitud": -1.28,
            "nodeId": 1,
            "time": "08-02-2021 13:23:54",
            "type": "Exploit"
        }
    ]
}
```

Exemple de fitxer JSON d'entrada d'atacs.

5.2.5 Comunicació amb back end

La comunicació amb la banda del servidor s'ha fet a partir de la llibreria *axios*, que permet realitzar crides HTTP de manera ordenada. Cada crida es fa de forma asíncrona i retorna un objecte del tipus promesa (Promise) que podem definir una lògica per a que s'executi quan es produeix el retorn de la crida.

A nivell arquitectural, implementarem les operacions CRUD per als recursos d'atac i organització, en fitxers separats segons recursos.

5.3 Securització de la aplicació

5.3.1 Justificació de l'arquitectura

Hem decidit separar el repositori pròpiament del backend, que gestiona organitzacions i atacs, del repositori que gestiona els usuaris i els tokens per a la possibilitat de que l'organització per la qual desenvolupem aquest projecte estigui disposada a implementar-ne més en un futur. Així, una API especialitzada podria servir com a sistema de tokens, ja no només per al projecte desenvolupat, sinó també per a qualsevol projecte que dita organització desenvolupi.

Anomenem *tfg_sts* (STS, de *Simple Token Service*) al repositori de la API de la securització. La tecnologia emprada per a desenvolupar-lo serà el llenguatge de programació Python i el framework flask, com en el back end del projecte. L'arquitectura serà la mateixa amb la diferència que afegirem, junt al servei, el dto i el controlador, el mòdul anomenat model, amb les definicions de les taules corresponents al servei. La llibreria que utilitzarem com a ORM és SQLAlchemy. Aquesta variació és deguda a que, com hem vist al capítol dedicat al disseny de la base de dades, per als administradors de la web i als tokens és un PostgreSQL, així que té sentit definir el model tancat de les taules per facilitar la seva interacció amb el codi. El repte d'aquesta arquitectura és que els canvis a l'estructura de les taules afectaran directament al codi, al contrari que passa amb MongoDB.

5.3.2 Protocol bearer token

El header de totes les peticions cap al back end del projecte - excepte les de registre i login - ha de contenir un camp de nom *Authorization* que contingui un token, codificat en HS256 pel servidor de STS. En cas que l'usuari que realitza la petició no estigui registrat en el sistema del back end, si el servidor STS que és un usuari autoritzat, aquest es crearà dins d'una col·lecció a MongoDB, per a tenir altra informació a més de la desada a la base de dades d'administradors, i única pel nostre projecte.

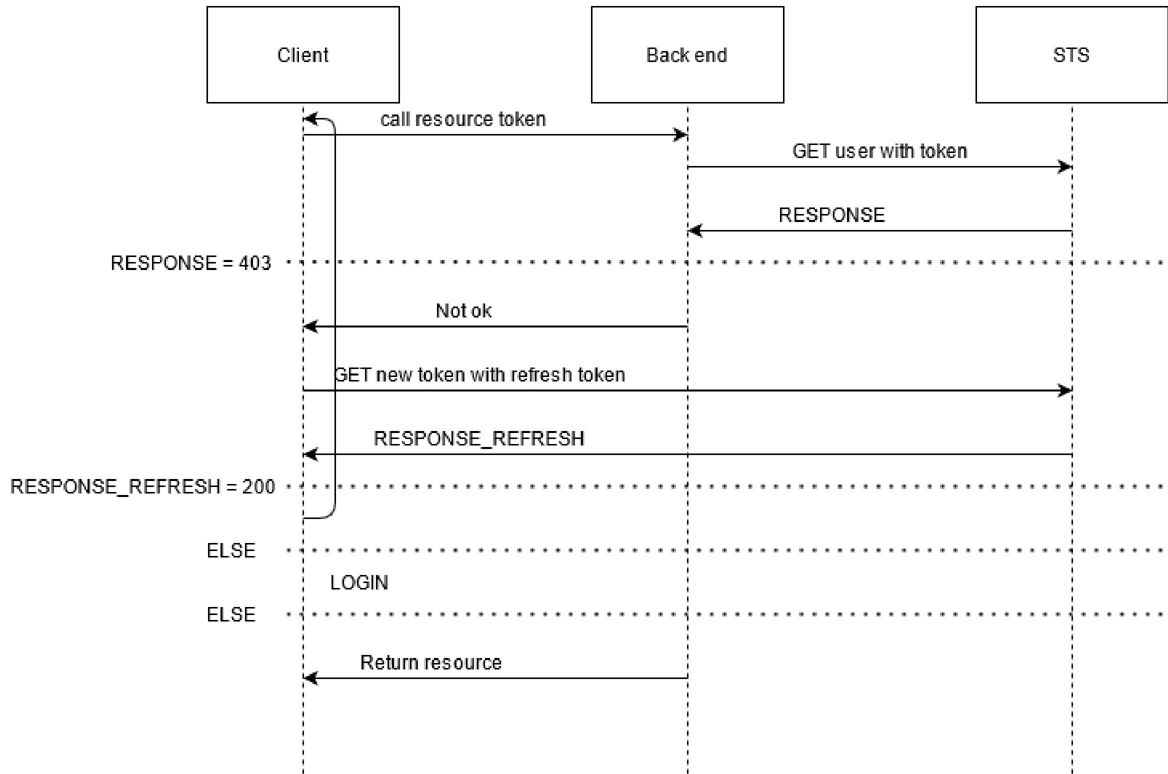


Diagrama de seqüència entre el client (front end), back end i servidor STS de la aplicació durant el procés de petició d'un recurs del front end al back end.

5.4 Contenidors

5.4.1 Introducció

En el context de l'enginyeria de software, hom parla de **contenidors de software** de les instàncies d'**espais d'usuaris**, resultants d'un procés de **virtualització a nivell de sistema operatiu** - és a dir, els recursos del *host* són fraccionats pel sistema operatiu -.

Dits contenidors s'executen dins un *sandbox*, és a dir, un entorn aïllat (per exemple, applets de Java o màquines virtuals), anomenat FUSE (*Full-System Emulation*), és a dir, software que emula cadascun dels components de l'entorn virtual (per exemple, execució d'instruccions al processador, perifèrics, xarxes virtuals...).

En sí, els contenidors de software son binaris executables, portables i contenen tot el que necessiten per executar-se: llibreries del sistema, eines i paquets de la aplicació a executar i l'aplicació mateixa. Teòricament, un contenidor s'executarà amb el mateix resultat independentment del *host* sobre el que s'executi.

Com que el context d'aquest projecte és el de la ciberseguretat, val la pena fer notar al lector que, tot i que el FUSE emula les pròpies instruccions del processador, aquesta emulació, així com l'aplicació conteneritzada s'executen al processador físic de la màquina que el conté. Així doncs, els forats de seguretat del mateix poden ésser aprofitats per **malware**. No hem d'estar presos per una sensació de falsa seguretat davant d'*exploits*.

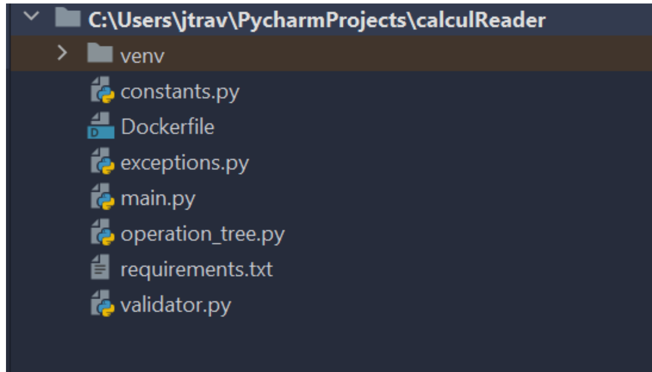
5.4.2 Docker

És el mètode de virtualització a nivell de sistema operatiu més utilitzat amb diferència respecte els seus competidors directes. Es tracta d'un projecte Open Source escrit en llenguatge de programació *Go*, que aprofita les característiques d'aïllament del kernel de linux - com espais de noms -, de tal manera que resulta molt lleuger i fàcil crear contenidors per aquest sistema operatiu - tot i que això no limita el seu ús, doncs es poden crear contenidors per a *Windows* també -.

El seu ús estès desemboca en dues de les seves principals avantatges, la primera, el suport de la comunitat en pàgines web de dubtes online, *chats*, i en l'entorn de treball. La segona, la integració amb sistemes de CI/CD com CircleCI, GitHub, AWS ECS, Kubernetes i d'altres.

5.4.2.1 Dockerfile

És el fitxer de configuració del contenidor que volem aixecar. Si la nostra aplicació només té un dockerfile, aquest ha d'estar al nivell de la *root* del mateix i anomenar-se, per defecte, *Dockerfile*.



Exemple de la estructura d'una aplicació de python que ha de desembocar en un únic contenidor, i per tant només necessita un Dockerfile

El format del Dockerfile és un conjunt de comandes i arguments que docker executa de manera seqüencial. Les comandes s'escriuen en majúscules i els arguments en minúscules.

Els comentaris són d'una única línia i van precedits del caracter coixinet.

Segons la referència del fitxer Dockerfile, la primera comanda en dit fitxer ha de ser una instrucció FROM, precedida del tag de la imatge pare. En el cas d'una aplicació simple de Python, el tag de la imatge pare és, normalment, python:<versió de python>.

La instrucció per executar un arxiu és RUN, i la seva sintaxis té dues formes alternatives:

- La forma de shell: RUN <comanda>, per exemple: RUN mkdir /usr/src/app/ per crear un directori al contenidor.
- La forma exec: RUN ["executable", "param1", "param2"], per exemple: RUN ["python", "main.py"], per executar un arxiu de python anomenat main.py.

Una altre instrucció molt important és la instrucció ENV, amb la sintaxis ENV <key>=<value>, que permet desar variables d'entorn al sistema operatiu del contenidor.

Amb COPY copiem arxius del origen (el nostre *host*) cap a una carpeta del sistema de fitxers del contenidor. Té dues formes de sintaxis:

- COPY [--chown=<user>:<group>] <src>...<dest>.
- COPY [--chown=<user>:<group>] ["<src>", ... "<dest>"] per rutes amb espais en blanc.

Una altre comanda molt important és WORKDIR, que estableix el directori actual per a comandes com RUN, CMD, ENTRYPOINT, COPY i ADD. Si no especifiquem un WORKDIR, s'agafa per defecte el directori arrel.

La sintaxi per a aquesta comanda és WORKDIR /path/to/workdir.

La comanda CMD, com RUN, executa una comanda amb arguments. La diferència principal entre CMD i RUN és que, mentre RUN s'executa en temps de construcció de la imatge - i per tant només un cop -, CMD s'executa cada cop que executem la imatge ja construïda.

Per a construir el contenidor usem la instrucció build, seguida de la ruta fins a l'arrel del projecte:

docker build /path/to/project

Per a executar un projecte usem la comanda run, seguida del tag del contenidor construït:

docker run <tag>

Per a obtenir una llista de totes les imatges de contenidors construides - amb altre informació, com el tag de cadascuna, usem:

docker image ls.

```
FROM python:3.8
RUN mkdir /usr/src/app/

COPY . /usr/src/app/
WORKDIR /usr/src/app/
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

Format de un fitxer Dockerfile per a un projecte de python. Les instruccions s'executen de manera seqüencial.

5.4.3 Docker-compose

Si la nostra aplicació és prou gran, necessita més d'un contenidor per a les diferents parts que el conformen. En concret, el treball que ens ocupa necessita de cinc contenidors diferents, a llistar:

- Frontend, un contenidor pel react.
- Backend: quatre contenidors:
  - API REST de les dades.
  - API REST del servei de tokens.
  - Base de dades dels atacs i les organitzacions.
  - Base de dades d'usuaris del servei de tokens.

Docker-compose és una eina per a gestionar de forma senzilla aquestes aplicacions multi contenidor, de manera que ens redueix el nombre de comandes per iniciar el sistema de dues per contenidor si usem només Dockerfile a una per repositori (independentment del nombre de contenidors del mateix). Nosaltres tenim tres repositoris, així doncs, en particular passem d'haver d'usar deu comandes a només tres.

Aquesta eina utilitza una fitxer yml, que s'ha d'anomenar docker-compose.yml, que ha d'estar desat no necessàriament a l'arrel del projecte, però si a la carpeta on executem la comanda:

docker-compose up --build -d

Que ens construirà lesimatges corresponents (flag build) i les executarà totes (argument up) en forma de dimoni (flag d).

5.4.4 Contenidors al STS

```yaml
services:
    postgresql:
        image: postgres:latest
        container_name: oauth2_db
        hostname: oauth2_db
        env_file:
          - ../.env
        ports:
            - "5432:5432"
    tfg_oauth2:
      container_name: tfg_oauth2
      hostname: tfg_oauth2
      build:
        context: ../.
        dockerfile: ./docker/appDockerfile
      restart: always
      ports:
        - "5001:5000"
      env_file:
        - ../.env
      depends_on:
        - postgresql
```

docker-compose.yml al sts

El nostre STS està composat de dos contenidors: un per la base de dades dels usuaris i l'altre per la API REST en Python. Així doncs, necessitem definir dos serveis al nostre fitxer *docker-compose.yml*, un per cada contenidor.

5.4.4.1 PostgreSQL

Procedim a explicar la raó de cada tag:

- image: especifiquem el nom de la imatge pare, en aquest cas postgres, i a més a més, la última versió.
- container_name: el nom que permetrà als demés contenidors identificar-ne aquest.
- hostname: el nom que permet al contenidor identificar-se a si mateix (per defecte, localhost).
- env_file: la ruta relativa al fitxer de configuració.
- ports: en aquest cas, el port 5432 (el port per defecte de postgresql) del contenidor es connectarà al contenidor 5432 de la nostra màquina. Hem pogut prendre aquesta decisió perquè no tenim cap altre postgresql corrent a la nostra màquina.

5.4.4.2 Python

En el cas del contenidor de nom - nom tant de contenidor com de *host* - tfg_oauth2, procedim a explicar els tags més interessants:

- build: aqui es defineix la configuració per construir una imatge a partir d'altres fonts fora de compose. En particular:
  - context: és un valor requerit des d'on s'executarà el procés de construcció per a totes les fonts referides dins el tag de *build*.
  - dockerfile: la ruta al fitxer amb format Dockerfile (vist a l'apartat anterior), relatiu a la ruta especificada en el context. En el nostre cas es tracta del fitxer appDockerfile, a la mateixa carpeta que docker-compose.yml.
- restart: es defineix en aquest tag quan s'ha de tornar a encendre el contenidor. En el nostre cas, volem que sempre es reiniciï, però en el cas, per exemple, d'una base de dades, pot no interessar que es torni a reiniciar cada vegada, doncs com que teòricament cada execució d'un contenidor és independent de totes les anteriors, es perdran totes les dades desades en el mateix en cada inici.
- ports: a diferencia del servei postgresql, en la nostra màquina si voldrem executar més d'un servei de flask, que per defecte escolta al port 5000. Per això, en aquest cas haurem de connectar el port 5000 del contenidor amb un altre de la màquina, en particular el 5001.
- depends_on: especifiquem aquí els serveis del qual el nostre n'és dependent, en aquest cas, la nostra aplicació de tokens necessita que la base de dades estigui aixecada per a poder funcionar correctament. Així, ho indiquem en aquest tag, de manera que el servei tfg_oauth2 només s'aixecarà en cas de que el server postgresql estigui aixecat també.

5.4.4.3 Dockerfile per a tfg_oauth2

```
FROM python:3.8
RUN mkdir /usr/src/app/

COPY . /usr/src/app/
WORKDIR /usr/src/app/
EXPOSE 5000
RUN pip install -r requirements.txt
CMD ["python", "app.py", "run"]
```

Fitxer appDockerfile, dins la carpeta docker, el Dockerfile pel server tfg_oauth2

A continuació s'explicarà el dockerfile definit al tag build del docker-compose per tfg_oauth2.

- FROM python:3.8, definim el contenidor pare com el oficial de python en la seva versió 3.8.
- RUN mkdir /usr/src/app/, executem la comanda per crear el directori arrel de l'aplicació.
- COPY . /usr/src/app/, copiem tots els fitxers del projecte a la nostra màquina cap al directori arrel de l'aplicació al contenidor, creat anteriorment.
- WORKDIR /usr/src/app/, definim dit directori com la ruta relativo des d'on s'executaran totes les comandes a partir d'ara.
- EXPOSE 5000, obrim el port 5000 del contenidor, que justament és el que utilitza flask per a escoltar peticions.
- RUN pip install -r requirements.txt, instal·lem totes les dependències de python per a aquest projecte.
- CMD [“python”, “app.py”, “run”], la comanda que inicia una aplicació flask, la definim per a executar-se quan s'engegui el contenidor.

5.4.5 Contenidors a la API REST

```yaml
services:
  mongodb:
      image: mongo:latest
      container_name: mongoserver
      hostname: mongoserver
      env_file:
        - ../.env
      ports:
          - "27017:27017"
      command: [--auth]
  tfg_back_rest:
    container_name: tfg_back_rest
    hostname: tfg_back_rest
    build:
      context: ../.
      dockerfile: ./docker/appDockerfile
    restart: always
    ports:
      - "5000:5000"
    env_file:
      - ../.env
    depends_on:
      - mongodb
```

docker-compose.yml al nostre webservice

De manera similar al repositori del servei de tokens, el back end de la nostra aplicació està format per dos contenidors: un per la base de dades, en aquest cas un MongoDB; i un altre per a un webservice de python en flask. A continuació s'explicarà la definició de cadascú d'ells.

5.4.5.1 MongoDB

S'ha definit de manera molt similar a la base de dades postgresql per al servei de tokens. Les seves diferències es llisten a continuació.

- image: s'ha agafat la imatge oficial de mongo, en la seva última versió.
- container_name i hostname: el nom per aquest contenidor és mongoserver.
- ports: el port 27017, el port per defecte a mongodb al contenidor, s'ha connectat al mateix port de la màquina, doncs no s'ha d'executar cap altre servei de mongodb en la nostra màquina.
- command: la gran diferència respecte el servei postgresql. És l'equivalent a compose a CMD al Dockerfile, una llista definida entre claudàtors, amb la llista de comandes a executar quan s'inicia el servei. MongoDB, per defecte, no demana credencials a cap connexió. Per revertir aquesta situació s'ha de passar el flag --auth a l'execució del contenidor.

5.4.5.2 Python

```
FROM python:3.8
RUN mkdir /usr/src/app/

COPY . /usr/src/app/
WORKDIR /usr/src/app/
EXPOSE 5000
RUN pip install -r requirements.txt
CMD ["python", "app.py", "run"]
```

Fitxer appDockerfile a la carpeta docker del webservice del nostre back end. Exactament igual al del servei de tokens.

El cas del servei del nostre back end, anomenat tfg_back_rest, és pràcticament igual al del servei de tokens, és a dir: tfg_oauth2. De fet, la similitud es posa obertament de manifest quan es nota que ambdós fitxers appDockerfile tenen el mateix contingut.

Es llisten les diferències principals:

- ports: aquí es connecta el port 5000 del contenidor, per defecte a flask, amb el 5000 de la nostra màquina (el 5001 està ocupat pel servei de tokens).
- depends_on: aquest servei també depèn d'una base de dades, en aquest cas es tracta de la base de dades d'organitzacions i atacs, desada en la base de dades MongoDB i que correspon al servei definit més amunt al mateix docker-compose.yml, mongodb.

5.4.6 Contenidors al front end

```yaml
version: '3.7'

services:

  react:
    container_name: tfg_front
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - '.:/app'
      - '/app/node_modules'
    ports:
      - 3001:3000
    environment:
      - CHOKIDAR_USEPOLLING=true
```

docker-compose.yml a la carpeta arrel del repositori tfg_front

El repositori de front end està composat d'un sol contenidor: el del propi front, implementat en react.

Degut al temps que es tarda en desplegar cada cop el contenidor de react, hem afegit el valor de connectar el nostre codi en local amb el de la app, de manera que quan canvia el nostre codi es reflexi automàticament al contenidor.

Per això hem usat el tag volumes, i hem connectat el repositori local, representat amb un punt, amb el que hi ha a /app.

A més hem establert la variable d'entorn CHOKIDAR_USEPOLLING a true, que s'encarrega de recargar l'aplicació de react quan es detecta un canvi al codi.

Com es pot veure, hem connectat el port 3000 del contenidor, el corresponent al que escolta react per defecte, amb el 3001 de la nostra màquina, per si volguessim utilitzar una altre aplicació de react a la nostra màquina local.

A més, com podem veure al tag build, el context és el mateix on hem desat el docker-compose.yml, és a dir el repositori arrel i disposem d'un Dockerfile que estudiarem a continuació.

```
FROM node:13.12.0-alpine

WORKDIR /app

ENV PATH /app/node_modules/.bin:$PATH

COPY package.json ./
COPY package-lock.json ./
RUN npm install --silent
RUN npm install react-scripts@3.4.1 -g --silent

COPY . ./

CMD ["npm", "start"]
```

Fitxer Dockerfile al directori arrel de tfg_front

Estudiem per parts el fitxer Dockerfile corresponent al repositori tfg_front:

- FROM node:13.12.0-apline, on especifiquem el contenidor pare, en aquest cas, el contenidor oficial de node.
- WORDIR /app, establim el directori des d'on executarem les següents comandes, corresponent al codi del repositori.
- ENV PATH /app/node_modules/.bin:$PATH, afegim la carpeta on és l'executable de node al PATH del contenidor, així, per invocar-lo en un futur, simplement el podem invocar amb la comanda npm.
- COPY package.json i RUN npm install, copiem els fitxers de dependències i les instal·lem usant el executable de node.
- COPY . ./, copiem els fitxers del repositori al nostre directori actual definit anteriorment a WORKDIR.
- CMD ["npm", "start"], definim la comanda que s'executarà quan aixequem el contenidor amb la comanda up de docker-compose.

5.4.7 Com aixecar el servei

L'objectiu d'aquest apartat és proveir al lector de les instruccions pertinents per aixecar el servei complet a partir del codi acabat de clonar.

```
PS C:\Users\jtrav\Documents\tfg\codi> git clone https://github.com/joanTrave/tfg_back.git
Cloning into 'tfg_back'...
remote: Enumerating objects: 138, done.
remote: Counting objects: 100% (138/138), done.
remote: Compressing objects: 100% (73/73), done.

Receiving objects: 100% (138/138), 18.10 KiB | 1.29 MiB/s, done.
Resolving deltas: 100% (76/76), done.
PS C:\Users\jtrav\Documents\tfg\codi> git clone https://github.com/joanTrave/tfg_front.git
Cloning into 'tfg_front'...
remote: Enumerating objects: 109, done.
remote: Counting objects: 100% (109/109), done.
remote: Compressing objects: 100% (77/77), done.

Receiving objects: 100% (109/109), 318.00 KiB | 211.00 KiB/s, done.
Resolving deltas: 100% (41/41), done.
PS C:\Users\jtrav\Documents\tfg\codi> git clone https://github.com/joanTrave/tfg_sts.git
Cloning into 'tfg_sts'...
remote: Enumerating objects: 44, done.
remote: Counting objects: 100% (44/44), done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 44 (delta 10), reused 44 (delta 10), pack-reused 0
Unpacking objects: 100% (44/44), done.
PS C:\Users\jtrav\Documents\tfg\codi> ls


    Directorio: C:\Users\jtrav\Documents\tfg\codi


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        23/05/2021     21:38                tfg_back
d-----        23/05/2021     21:39                tfg_front
d-----        23/05/2021     21:39                tfg_sts


PS C:\Users\jtrav\Documents\tfg\codi>
```

Procés de clonació de tots els repositoris d'aquest treball de final de grau.

Així, partim d'una carpeta que anomenarem arrel, d'on trobem *tfg_front, tfg_back* i *tfg_sts*.

En el cas del backend ens hem d'assegurar que existeixi el fitxer d'entorn .env. Podem copiar el fitxer .env.local de cada repositori i escriure-hi les variables d'entorn que corresponen. A continuació es proporcionen els continguts de cada fitxer d'entorn.

```
ENV=DEV
MONGO_DBNAME=attacks
MONGO_USER=myUserAdmin
MONGO_PASSWORD=supersecret
MONGO_HOST=mongoserver
MONGO_PORT=27017
STS_URI=http://tfg_oauth2:5000
MONGO_INITDB_ROOT_USERNAME="myUserAdmin"
MONGO_INITDB_ROOT_PASSWORD="supersecret"
MONGO_INITDB_DATABASE="admin"
```

Fitxer d'entorn del repositori tfg_back.

```
ENV=DEV
POSTGRES_USER=myUserAdmin
POSTGRES_PASSWORD=supersecret
POSTGRES_DB=oauth2
POSTGRES_URI=postgresql://myUserAdmin:supersecret@oauth2_db:5432/oauth2
EXPIRATION_SECONDS=3600
EXPIRATION_REFRESH_SECONDS=86400
SECRET_KEY=superlol
BUILD_DATABASE=True
```

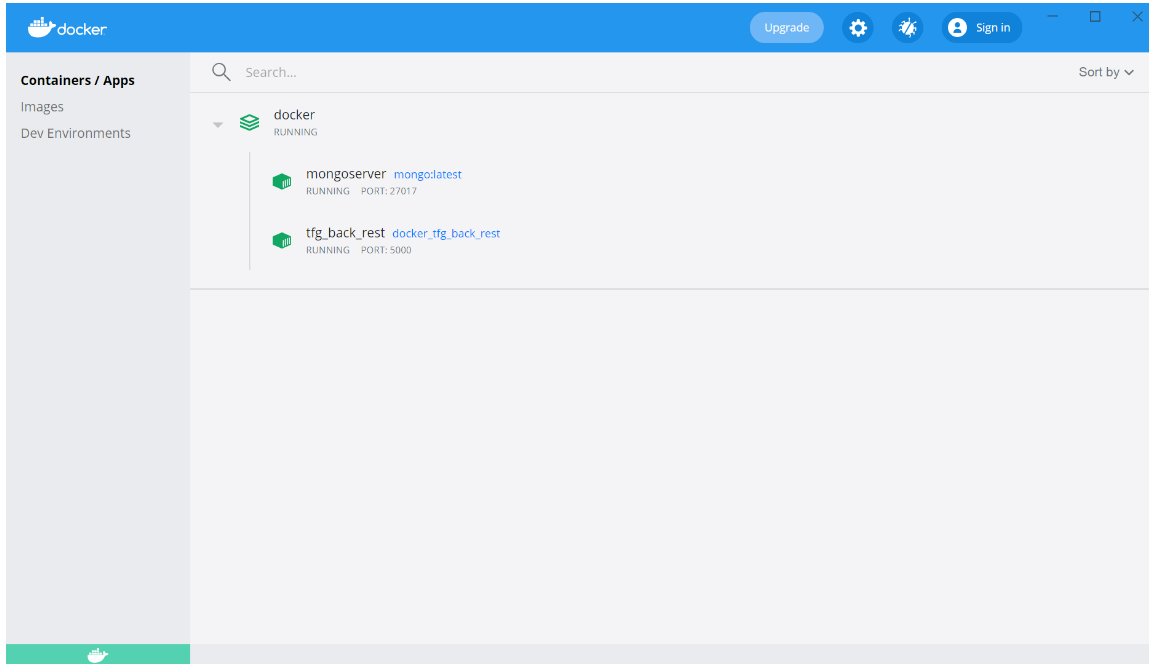
Fitxer d'entorn del repositori tfg_sts.

Per a aixecar els serveis d'un repositori, hem de desplaçar-nos a la carpeta anomenada docker del repositori en qüestió (ho podem fer amb la comanda cd del *powershell* de windows o la terminal bash de linux), i escriure la comanda: docker-compose up --build -d.

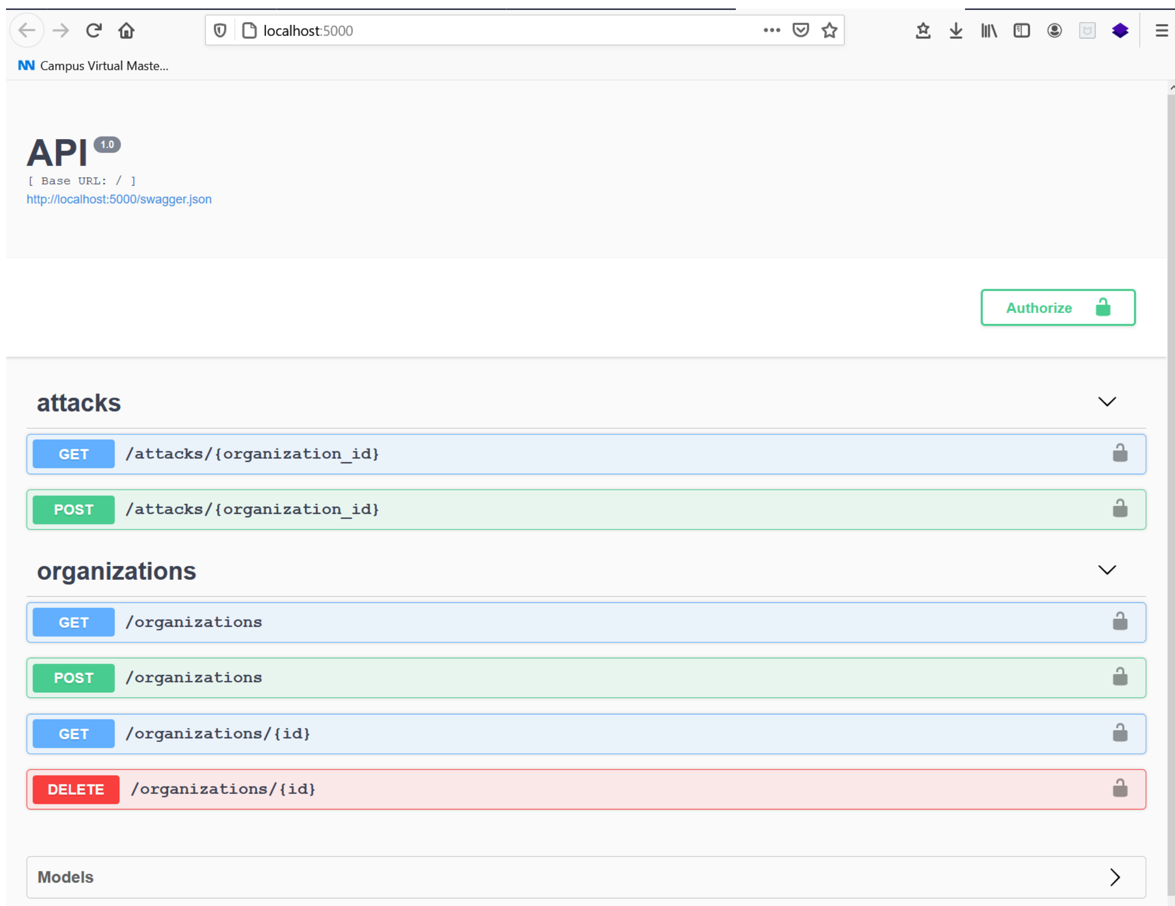
Si treballem a Windows i disposem del client amb GUI per a windows de docker podem gestionar els contenidors generats des d'allà.

```
PS C:\Users\jtrav\Documents\tfg\codi> cd .\tfg_back\docker\
PS C:\Users\jtrav\Documents\tfg\codi\tfg_back\docker> docker-compose up --build -d
Docker Compose is now in the Docker CLI, try `docker compose up`

Pulling mongodb (mongo:latest)...
latest: Pulling from library/mongo
01bf7da0a88c: Pull complete
f3b4a5f15c7a: Pull complete
57ffbe87baa1: Pull complete
77d5e5c7eab9: Pull complete
43798cf18b45: Pull complete
67349a81f435: Pull complete
590845b1f17c: Pull complete
1f2ff17242ce: Pull complete
6f11b2ce0594: Pull complete
91532386f4ec: Pull complete
705ef0ab262e: Pull complete
e6238126b609: Pull complete
Digest: sha256:8b35c0a75c2dbf23110ed2485feca567ec9ab743feee7a0d7a148f806daf5e86
Status: Downloaded newer image for mongo:latest
Building tfg_back_rest
failed to get console mode for stdout: The handle is invalid.
[+] Building 20.8s (10/10) FINISHED
 => [internal] load build definition from appDockerfile                    0.1s
 => => transferring dockerfile: 212B                                       0.0s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [internal] load metadata for docker.io/library/python:3.8              1.4s
 => [internal] load build context                                          0.2s
 => => transferring context: 69.25kB                                       0.2s
 => [1/5] FROM docker.io/library/python:3.8@sha256:4a3ccab7b427dbae84782e  0.0s
 => CACHED [2/5] RUN mkdir /usr/src/app/                                   0.0s
 => [3/5] COPY . /usr/src/app/                                             0.3s
 => [4/5] WORKDIR /usr/src/app/                                            0.0s
 => [5/5] RUN pip install -r requirements.txt                             18.0s
 => exporting to image                                                     0.7s
 => => exporting layers                                                    0.7s
 => => writing image sha256:9a03b41d775f2b11f854a3a32152596829ccdbf95e508  0.0s
 => => naming to docker.io/library/docker_tfg_back_rest                    0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Successfully built 9a03b41d775f2b11f854a3a32152596829ccdbf95e5082684092c1b105cd8357
Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
Creating mongoserver ... done
Creating tfg_back_rest ... done
PS C:\Users\jtrav\Documents\tfg\codi\tfg_back\docker>
```

Procediment per aixecar els serveis del repositori de back end des de terminal.



Podem comprovar des del client de docker per a Windows que els serveis per al repositori de back end estan aixecats al port corresponent.



De la mateixa manera, ho podem comprovar empíricament accedint al servei de swagger de la API de back end.

6. Conclusions

Havent comprovat que s'han satisfet tots els requisits funcionals, val a concloure que la implementació ha estat un èxit.

De la mateixa forma es pot dir que l'objectiu de crear una plataforma mitjançant rigorositat a l'hora d'escollir l'arquitectura i les eines ha estat completat, doncs la majoria de decisions importants han estat preses a través de comparatives.

A nivell personal, l'autor està satisfet d'haver après a usar el framework React per javascript, i les tecnologies per a representar dades geogràfiques. A més, valora molt positivament l'aprenentatge i posterior ús les eines de conteniriització com Docker o Docker compose, doncs estalvien temps i més important encara, permeten una compatibilitat entre sistemes més sòlida, i milloren enormement la portabilitat de les aplicacions.

En el camí continu de proveir una plataforma per a que administradors de seguretat puguin saber l'estat de la seva organització de manera inmediata, sembla un pas natural el eliminar totes les llistes de recursos possibles i substituir-les per diagrames. La següent llista a la que s'enfronta l'administrador és la de terminals i usuaris dins una mateixa oficina, que fora interessant substituir-la pel diagrama de les instal·lacions de la mateixa, de manera que es pugui observar amb encara més rapidesa quines son les relacions entre terminals i cercar possibles punts d'entrada de malware.

Una altra evolució natural podria ser implementar a cada organització un sistema de log reporting, com Logstash, que informi directament a la API de recursos dels atacs, de manera que no calgui que l'administrador pengi els json corresponents.

També es podria desenvolupar un mode de visualització a temps real, que pugui permetre controlar l'estat de l'organització en viu.

Difícilment arribarem a cap projecte a un nivell complet d'automatisme. Tot i això, en aquest treball només hem rascat la superficie del món de DevOps, els contenidors i aquest, tant cercat, automatisme. Es defineixen a continuació possibles millores per a iteracions immediates, usades avui en dia per la indústria i amb una corba d'aprenentatge no gaire pronunciada.

- Ús d'una eina d'orquestació de tasques com Jenkins o Kubernetes per a fer el deployment de manera automàtica.
- Ús d'una eina de validació de codi com SonarCloud per a no desplegar a no ser que es passin tots els *quality gates*.
- Ús d'una eina de manteniment de repositoris per a tenir les imatges sempre desades a la xarxa com a backup, per exemple, Nexus.

7. Bibliografia

Análisis de malware para sistemas windows, Mario Guerra Soto, ISBN: 9788499647661

Clean Code, Robert Cecil, ISBN: 9788025142769

MongoDB: The Definitive Guide: Powerful and Scalable Data Storage, Christina Chodorow, ISBN: 9781449344689

Fullstack React: The Complete Guide to ReactJS and Friends: Accomazzo, Anthony, Murray, Nate, Lerner, Ari, ISBN: 9780991344628

Kasperky Threat map

https://cybermap.kaspersky.com/

Fortinet threat map

https://threatmap.fortiguard.com/

Checkpoint threat map

https://threatmap.checkpoint.com/

MongoDB documentation

https://docs.mongodb.com/

Redis documentation

https://redis.io/documentation

CouchDB documentation

https://docs.couchdb.org/en/stable/

RavenDB documentation

https://ravendb.net/docs/article-page/5.1/csharp

MySQL documentation

https://dev.mysql.com/doc/

MariaDB documentation

https://mariadb.com/kb/en/documentation/

PostgreSQL documentation

https://www.postgresql.org/docs/

Oracle documentation

https://docs.oracle.com/en/database/

Flask API documentation

https://flask.palletsprojects.com/en/2.0.x/

React

https://reactjs.org/docs/getting-started.html

Vue.JS

https://vuejs.org/v2/guide/

Angular

https://angular.io/docs

Oauth2 protocol

https://oauth.net/2/

Oauth2 google docs

https://developers.google.com/identity/protocols/oauth2

Bearer token specification

https://datatracker.ietf.org/doc/html/rfc6750

Docker documentation

https://docs.docker.com/

rkt

https://www.openshift.com/learn/topics/rkt

cri-o

https://cri-o.io/

containerd

https://containerd.io/docs/