UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

GradeMe: Sistema de correcció automàtica de codi

Autor: Guillem Mascarell Ruiz

Director: Dr. Lluís Garrido

Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. UB

Barcelona, 27 de juny de 2019

Agraïments:

Primerament, agrair el suport de la meva família no tant sols durant aquest projecte sinó durant tot el transcurs de tot el grau universitari.

Agrair també a Guillem Pascual que va permetre participar en aquest projecte i tot ho obert que ha estat amb mi, i també la llibertat amb la que m'ha deixat treballar juntament amb tots els correus que m'ha enviat responent a les meves preguntes.

Per últim però no menys important; agrair el suport constant i la dedicació del meu tutor i antic professor Lluís Garrido.

Resum:

GradeMe és un sistema que testeja codi extret de github en caixa negra i mostra per una pàgina web si el resultat es l'esperat o no. Aquest sistema funciona dins de contenidors dockers i consta de quatre serveis: una base de dades mongoDB, el servei de testeig de codi anomenat webhook, un bróker de missatges anomenat broadcaster i un servidor web per mostrar els resultats al usuari. Tota aquest aplicació l'ha desenvolupat en Guillem Pascual i el fa servir per a l'assignatura de TNUI de la qual és professor.

El llenguatge de programació utilitzat es Python per a webhook i el broadcaster, la pàgina web està feta en html dins d'un servidor flask. Els serveis de mongoDB, webhook i broadcaster funcionen dins de contenidors dockers, però la pàgina web no.

Tot i que el sistema funciona, no es escalable i costós de mantenir, per això es van proposar els següents canvis:

- Refactoritzar el sistema de broadcast fent servir un servidor redis.
- Afegir un servidor de socket flask que gestiona la informació.
- Canviar la plataforma que serveix la pàgina web.
- Afegir una lògica de comunicació entre el servidor de sockets i la pàgina web.
- Fer que tota l'aplicació funcioni dins de docker.

Tots aquest canvis s'han realitzat i el sistema funciona dins d'un entorn de contenidors docker amb tots els serveis comunicant-se entre ells, i des de la pàgina web es pot veure el resultat d'un commit.

Resumen:

GradeMe es un sistema que testea código extraído de github en caja negra y muestra por una página web, si el resultado es el esperado o no. Este sistema funciona dentro de contenedores docker y consta de cuatro servicios: una base de datos mongoDB, el servicio de testeo de código llamado webhook, un bróker de mensajes llamado broadcaster y un servidor web para mostrar los resultados al usuario. Toda esta aplicación la ha desarrollado Guillem Pascual y la usa para la asignatura de TNUI de la cual es profesor.

El lenguaje de programación utilizado es Python para webhook y el broadcaster, la página web está hecha en html en un servidor flask. Los servicios de mongoDB, webhook y broadcaster funcionan dentro de contenedores docker mientras que la página web no.

Aunque el sistema funciona, no es escalable y es costoso de mantener, por eso se propusieron los siguientes cambios:

- Refactorizar el sistema de broadcast usando un servidor redis.
- Añadir un servidor de sockets flask que gestiona la información.
- Cambiar la plataforma que sirve páginas web.
- Añadir una lógica de comunicación entre el servidor de sockets y la página web.
- Hacer que toda la aplicación funcione dentro de docker.

Todos estos cambios se han realizado y el sistema funciona dentro de un entorno de contenedores docker con todos los servicios comunicándose entre ellos y desde la página web se puede ver el resultado de un commit.

Abstract:

GradeMe is a code testing system which black box tests code from github and shows the result in a web page. This system works inside a docker container and it has four parts: a mongoDB database server, the code testing service named webhook, a message broker named broadcaster and a web page to show the result to the user. All this application was developed by Guillem Pascual and he uses it in TNUI subject whom he's the professor.

The programming language used is python for webhook and the broadcaster, the web page is done in html and is provided by a flask server. MongoDB, webhook and broadcaster services works inside docker container but it isn't the case for the web server.

Even though the system works, it has no scalability and the cost to maintain is high, that is why the following changes were suggested:

- Broadcast system refactor using a redis server.
- Add a socket service flask which manages the information.
- Change the web platform.
- Add communication logic between the socket server and the web page.
- Make it all work inside docker containers.

All suggestions were made and the system works fully inside docker containers with all services communicate with each other. It is possible to see the result of a commit in a webpage.

ÍNDEX DE CONTINGUTS

1. INTRODUCCIÓ:

Començar el grau universitari d'enginyeria informàtica pot ser molt dur, especialment les assignatures de programació si no es té algun tipus d'experiència prèvia. Molts alumnes ho passen realment malament i en alguns casos arriben a deixar-ho durant

el primer any ja que començar a programar sense tenir a ningú darrera guiant-te pot ser una tasca molt difícil d'assolir. Per això considero que existeix la necessitat d'oferir eines als alumnes per a que puguin aprendre a programar amb exercicis pràctics sense que un professor hagi d'estar revisant el codi d'uns 100 alumnes.

GradeMe (creat per Guillem Pascual) ofereix aquesta possibilitat i a més ho fa a partir de GitHub (una altre eina que considero que els alumnes haurien d'aprendre a fer servir la primera setmana de classes) cosa que facilita bastant les tasques tant pels professors com per als alumnes.

Aquest projecte (GradeMe) ha sorgit de la necessitat d'assistir als estudiants de la facultat (concretament als de la assignatura de TNUI) a l'hora de corregir el seu codi a nivell de si funciona o no, sense fer cap mena de control de qualitat i sense donar un feedback a nivell personal. L'aplicació funciona i ell (en Guillem Pascual) la fa servir per alsseus alumnes, però aquesta necessitat que li va sorgir la tenen (o poden tenir) molts altres professors d'altres assignatures especialment de primer any del grau. Tal i com estava muntat GradeMe només el podia fer servir el Guillem, ja que hi havia molta part hardcodejada i moltes d'altres que havies d'entendre bé com funcionaven per fer-ho servir, i era molt poc pràctic. Per això aquest projecte, si es vol fer servir per a més gent, té la necessitat de facilitar-ne l'ús tant pels professors com pels alumnes (els usuaris) i per això es necessari fer canvis per solucionar aquest problema.

1.1. MOTIVACIONS:

Continuar aquest projecte des del punt on es trobava era poc viable; afegir funcionalitats o mantenir-ne les que hi havien era molt costos i, a més, necessitava d'una refacció del codi important a nivell de com comunicar diferents serveis. La pàgina web que hi havia no es podia mantenir i li falten moltes funcionalitats importants per a que sigui apte per l'ús dels usuaris (especialment els professors).

Posar a disposició de la Universitat una eina de correcció de codi eficient fent servir una eina ja desenvolupada però que necessitava alguns canvis importants i fer més accessible el futur desenvolupament d'aquesta aplicació, feien que aquest projecte fos interesant i útil.

1.2. OBJECTIUS:

Les tasques principals del projecte són per una part implementar canvis en una aplicació que ja era operativa proposant millores a la mateixa, i per a una altra facilitar que en un futur una altre persona pugui continuar desenvolupant el projecte.

L'aplicació necessita una refacció de part del seu funcionament per a facilitar futures intervencions i poder afegir més usabilitats especialment al entorn d'usuari.

L'objectiu principal del projecte és que, després de tots els canvis, un usuari pugui veure un missatge de resposta per part del sistema com a resposta a un commit.

1.3. ESTRUCTURA DEL DOCUMENT

Aquest document està compost per les següents parts:

-**Planificació**: Explicació de com s'ha desenvolupat el projecte indicant quan s'ha trigat a cada etapa.

-**Esquema inicial**: Com funcionava abans d'aquest projecte i les característiques que tenia.

-**Tecnologies**: Explicació de quines són les tecnologies que s'han afegit o canviat per a millorar o ampliar l'aplicació.

-**Diseny**. Explicació de com funciona el projecte amb les tecnologies explicades prèviament utilitzades i explicació de parts importants de la configuració per aixecar l'aplicació.

-**Demostració**. demostració d'un cas explicant com funciona i mostrant els resultats.

-**Línies futures**. Explicació de la feina que queda per fer.

-Conclusions.

-**Glossari**.

-**Bibliografia**.

1.4. PLANIFICACIÓ

Durant el desenvolupament d'aquest projecte hi han hagut reunions bisetmanals amb el tutor per comentar com s'estava portant a terme les diferents etapes.
Les etapes i les seves duracions han estat les següents:

1- Reconeixement del projecte: GradeMe ja existia abans de començar i es va haver d'invertir temps tant en entendre com funcionava, com en implementar-ho en una màquina per poder fer les primeres proves.
De 2019-02-01 a 2019-03-01 aproximadament.

2- Afegir servei de broker redis: Un cop tot assimilat i entès el que era necessari, ho primer i més assequible era implementar el servidor redis.
De 2019-03-01 a 2019-04-01 aproximadament.

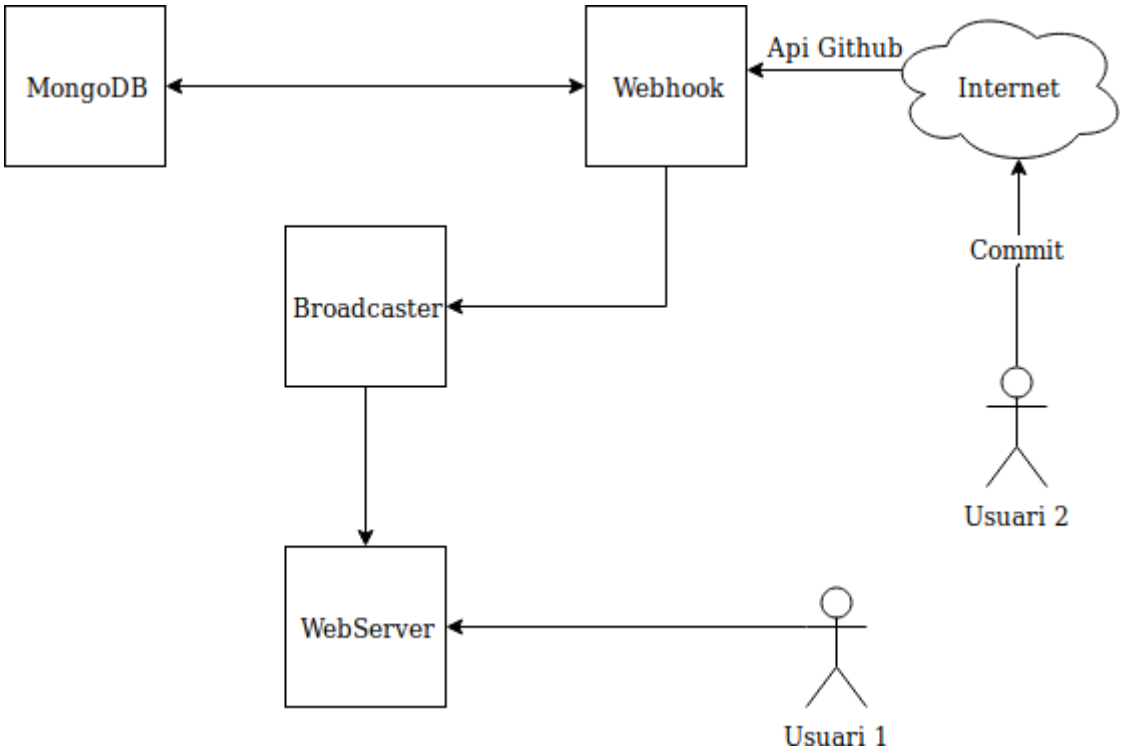
3- Afegir un servidor de sockets flask: Amb un servidor redis ja afegit, ara calia implementar el servidor que donés ús a redis.
De 2019-04-01 a 2019-05-01 aproximadament.

4- Lògica de connexió flask socketio: Flask ja rebia la informació però no la treia fora, així que es va haver d'implementar un sistema de resposta a esdeveniments al servidor flask i investigar com fer-ho servir en pàgines web.
De 2019-05-01 a 2019-06-01 aproximadament.

5- Retocs finals i memòria: Ja amb tot afegit, va caldre implementar-ho tot plegat i fer la prova del commit per obtenir resposta a més de fer la memòria i entregar-la el dia 27.
De 2019-06-01 a 2019-06-27 aproximadament.

2. ESQUEMA GENERAL

En un principi l'aplicació tenia la seguent estructura:



1.Esquema inicial de l'aplicació GradeMe.

L'aplicació constava de varies parts:

-Servidor webhook que gestiona els commit a partir de la api de github.

-Servidor mongodb on es guarden les dades.

-Servei de broadcast que reenvia les dades a tots els serveis.

-Servidor flask que fa a més de servidor web.

Quan un usuari fa un commit a través de github al repositori d'una assignatura registrada, l'aplicació de webhook es baixa aquest codi, l'executa en un entorn segur, i comprova que el resultat sigui el correcte. Independentment del resultat obtingut, webhook escriu al servidor de base de dades mongoDB la informació d'aquesta execució. Alhora, webhook envia aquesta informació a un servei de broadcast que la envia a tots els seus membres, en aquest cas, al servidor web.

El servidor webhook (realitzat per Guillem Pascual) és un servei que a partir de la api de github, es baixa el codi dels alumnes, i executa un test per extreure'n feedback i mostrar els resultats.

Webhook está fet amb python i té principalment 2 parts, el programa principal (main) i la gestió de tasques d'esdeveniments (jobs)

De forma seqüencial quan s'encén el servei de webhook, el primer que s'executa és una funció main que agafa els valors dels paràmetres com l'organització del github o el nom del host de la base de dades entre molts d'altres i inicialitza tot per després engegar el servei.

Jobs fa totes les tasques del procés de llegir codi de github, comprovar i retornar feedback, en diferents fases, això permet mostrar l'estat del procés a l'usuari en temps real.

Per realitzar la connexió amb github, es necessita haver creat prèviament una organització des de la interfície de github; aquesta organització s'ha d'especificar a l'hora d'iniciar webhook. L'organització, des del punt de vista dels usuaris, seria una assignatura (tot i que es pot tractar de forma diferent) on el professor crea l'organització i dona accés al repositori als alumnes. Tota aquesta gestió d'organització s'ha de fer des de github, però per a que funcioni l'aplicació és necessari obtenir el codi de l'organització.

Un cop es té la organització creada, un membre de la organització (alumne) pot realitzar un commit a github en el repositori adient fent que el servei de webhook es baixi el codi, executi un test i obtingui el resultat de l'execució.

La base de dades està feta utilitzant un servidor MongoDB; i la major part de la gestió d'aquesta es fa utilitzant python. MongoDB rep dades de webhook referents als resultats dels tests tan aviat com webhook acaba la tasca, i envia aquests mateixos resultats a flask a petició de l'usuari de la pàgina

web. A diferència de redis, les dades són persistents a disc, és a dir, no es perdrien en cas de reiniciar el sistema.

El broadcaster és un broker de missatges no reactiu fet amb python. S'executa abans que webhook escolti de github i es posa a escoltar i acceptar connexions de diferents serveis (en aquest cas webhook i flask). Després es posa a escoltar missatges que li arriben per les connexions i quan un missatge arriba, el reenvia a la resta de connexions. La gestió de reenviament es fa mitjançant un bucle for iterant a través de totes les connexions.

L'aplicació funciona i s'està usant, però si es vol introduir una sèrie de millores les quals són necessàries, fa falta canviar algunes coses; a més, hi ha certes parts que s'haurien de refactoritzar per fer-les més eficients i més sostenibles; com és el cas del broadcaster o del servidor web. En resum, el que seria convenient seria implementar un brocker de missatges i afegir un servei que gestioni la informació per a poder fer un servidor web que estigui separat de tota la lògica del sistema.

3. TECNOLOGIES

3.1 REDIS:

Redis es un servei d'emmagatzematge de dades en memòria de codi obert i que s'usa normalment per a caches o com a distribuïdor de missatges. Té l'avantatge que és molt senzill i molt lleuger, a part de que al ser codi obert hi ha molta

documentació i això facilita l'ús. A més és molt portable a python que és el llenguatge amb el que està fet gran part de GradeMe.

En un principi l'aplicació venia amb un broadcaster per a la comunicació entre serveis; però un broadcaster és, en aquest cas, poc eficient i consumeix més recursos dels que convindria (tot i que funciona perfectament); a més, degut a que es voldria poder dockeritzar(se'n parlarà més endavant), no permetria la rèplica d'instàncies del servei. Per això vaig decidir (seguint el consell d'en Guillem) implementar un servei de redis que comuniqués els serveis de forma més eficient i més flexible.
Al fer servir redis en comptes d'un broadcaster, es van obtenir principalment els següents 2 beneficis:

-Alliberar recursos del sistema, tant a nivell de xarxa com a nivell de memòria.
El broadcaster executa 2 threads constantment; un per acceptar serveis dels quals escoltarà i retransmetrà missatges, i l'altre per retransmetre missatges. Per retransmetre missatges, el broadcaster executa un bucle while que es manté mentre l'aplicació estigui encesa; dins d'aquest bucle while hi ha un bucle for que recorre tots els serveis connectats i comprova un per un si hi ha algun missatge nou rebut, de ser així l'envia per la xarxa i, en cas d'error (no es pot posar en contacte amb el servei), l'elimina de la llista de contactes. El sistema d'iteració per fer polling, a més, fa que si durant el tractament d'una connexió alguna cosa triga més del que és normal, la resta

de connexions es queden esperant i pot acabar en coll d'ampolla saturant el sistema sencer.

Tot això comporta que el sistema té dos fils executant-se en memòria i peticions constants a la xarxa tant per comprovar si hi ha missatges com per reenviar-los; si no hi ha massa usuaris i es té accés total al servidor això pot no ser un problema, però pot no ser així per aquest motiu es va decidir implementar una alternativa que alliberés de càrrega al servidor.

Redis funciona diferent. Qualsevol servei que vulgui fer servir redis, ha de conectarse al servidor (redis) i un cop fet pot, o bé subscriure's a un canal, o bé publicar un missatge a un canal, tot això es fa amb threads igual que a broadcaster, de manera que quan passa algun event (un commit d'un usuari a github) es publica un missatge al canal que toqui, i els serveis subscrits a aquest canal, obtenen dita informació.

Es segueixen fent peticions en un bucle while però de forma més ordenada i només qui realment ho necessita, hi ha molt menys trànsit d'informació per la xarxa ja que ara només s'envia la informació a qui toca. Cada connexió s'executa i tracta dades al moment que li toca i no necessita esperar el seu torn en la iteració de les connexions.

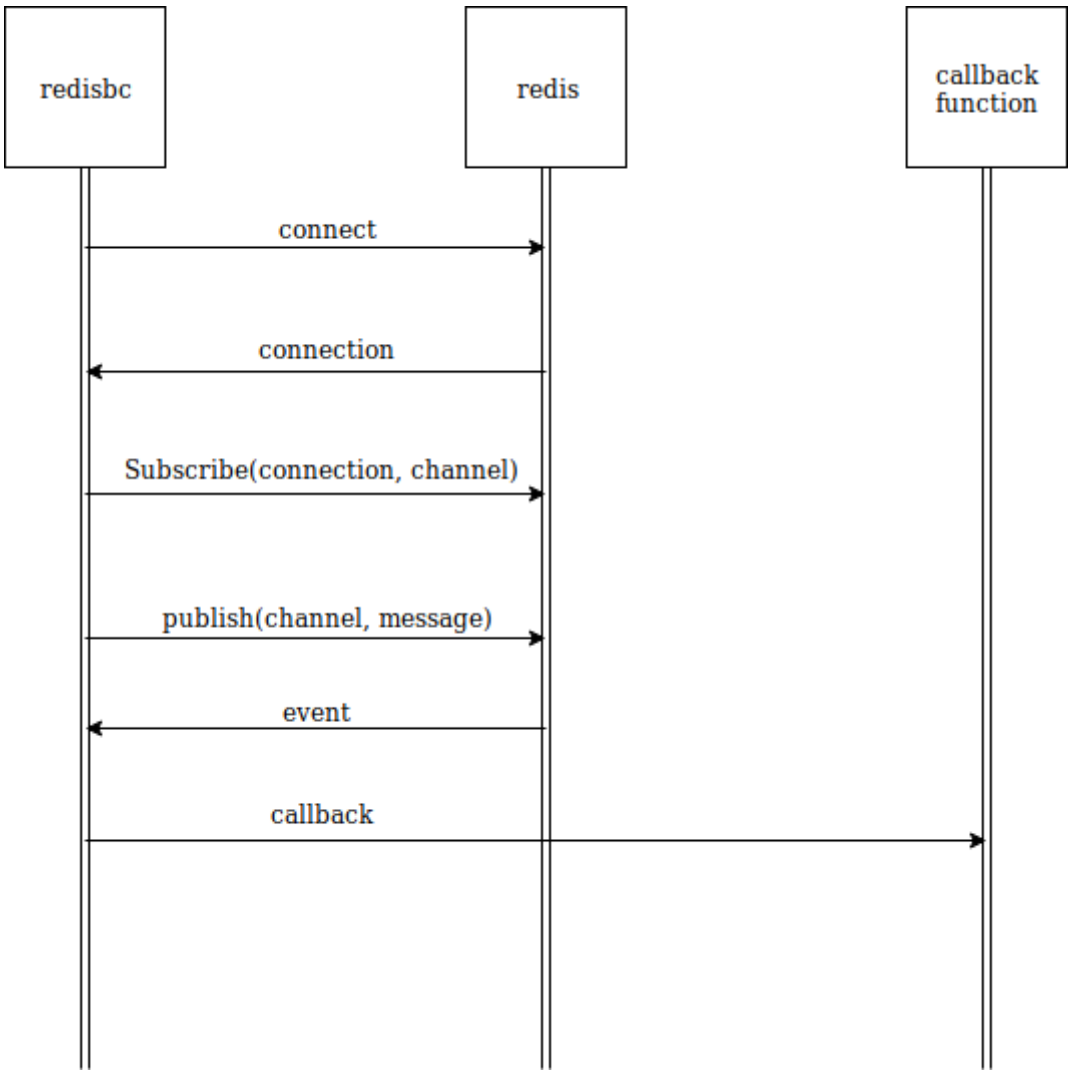
-Refacció del codi i incorporació de middleware.

Amb broadcaster hi havien tres serveis i es comunicaven a través d'un polling de connexions. Amb aquest sistema s'iterava a través de varies connexions i això feia que no fos del tot reactiu.

Al incorporar redis al sistema, estem incorporant una capa de middleware que executa la funcionalitat de brooker de missatges entre serveis, i estem fent que sigui completament reactiu mitjançant les funcions de callback.

Les funcions de callback són funcions que es poden passar per paràmetres a altres funcions on no tenen la necessitat d'estar declarades i s'executen a

mode reactiu, és a dir, com a resposta a algun event. En el nostre cas funciona de la següent manera; un servei es connecta a redis, a l'hora de subscriure's a un canal passa per paràmetre també la funció de callback, la funció de redis s'executa normalment i quan rep un missatge executa la funció rebuda per paràmetre en el moment de la subscripció. Redis no té en cap moment cap d'aquestes funcions declarades i cada instancia de redis declarada per cada un del serveis que en fan ús pot tenir una funció diferent assignada com a funció de resposta



2. Esquema del funcionomanet de les publicacions i subscripcions executant una 17nstan de callback.

D’aquesta manera el codi és 100% reactiu, ja que quan es rep un missatge s’executa la funció directament en comptes d’anar iterant a través de les connexions.

3.2. FLASK

Abans d'aquest projecte hi havia una interfície d'usuari, però era només per a ús funcional i no estava pensada per a gaires canvis o ampliacions. Aquesta interfície estava feta amb un servidor flask i el propi servidor era qui oferia el codi html.

Flask és un micro servidor web fet en python que es caracteritza per que no depèn de cap llibreria i no té feta la capa d'abstracció de base de dades, en poques paraules, no pren cap decisió. Això fa que sigui simple, extensible i a l'hora, costa de implementar

La idea era continuar implementant flask, però fer-ho dins del docker i que el servei que ofereixi sigui només d'intermediari, necessitant llavors, a més, un servidor frontend. A més el nou flask s'ha de connectar amb redis (abans ho feia amb el broadcaster) i ha d'estar preparat per poder transmetre les dades al servidor de frontend que hi hagi.

Així doncs el nou servidor flask fa la feina de servidor de sockets subscript a redis i comunica un servidor frontend (encara no existeix) i la part de backend (webhook i mongodb) a través de subscripcions a redis.

Una vegada està encès, flask funciona responent a dos tipus de endpoints, els que tracten la lògica del frontend i els que no. Tot el que tracta la lògica de frontend està relacionat amb socketio i estarà més detallat en el següent apartat. Pel que fa la resta de endpoints, es poden fer servir per a diferents necessitats; com oferir pàgines web com si fos un servidor web o fer accessos a la base de dades (per aquest projecte només s'ha fet servir socketio i una web per a la demostració).

A part dels endpoints el servidor també té funcions que s'utilitzen a mode de callback per a la subscripció a canals de redis explicat anteriorment.

3.3. SOCKETIO

Hi ha un sistema que gestiona els commits a github (webhook i mongodb), un sistema de comunicació (redis) i un servidor backend (flask) que tracta aquesta informació; però encara falta la part més visual, el frontend. En aquest tfg no s'ha tractat la part de frontend però si que s'ha tingut en compte la comunicació entre el frontend i el backend, i per aquesta comunicació s'ha triat l'opció de la llibreria socketio.

Per defecte javascript ja ve amb la funcionalitat de comunicació per sockets utilitzant una tecnologia anomenada websocket. Amb aquesta tecnologia ja es podria fer

comunicació entre client i servidor, però convé fer servir una llibreria (la qual s'ha d'importar) de javascript anomenada socketio ja que facilita molt les coses i dona robustesa a la comunicació.

La comunicació amb websockets fent servir la llibreria socketio funciona de la següent manera:

Un socket té com a informació (entre d'altres) dos camps, l'esdeveniment i les dades. D'esdeveniments n'hi ha de dos tipus, els estàndard (connexió, desconnexió, etc...) i els declarats, els quals poden ser qualsevol string. Les dades són strings.
El servidor té una sèrie d'esdeveniments declarats enfocats a socketio (amb una declaració especial) als quals respon depenent del esdeveniment que tingui assignat el socket rebut cridant la funció vinculada al esdeveniment. A més, per comunicar-se amb els clients, el servidor també pot enviar dades sense la necessitat de que sigui a mode de resposta d'esdeveniment. Aquesta ultima possibilitat es fa servir en les funcions de callback ja que no es respon a un esdeveniment de socketio (sinó de redis) però s'envien dades a través de sockets.

La pàgina web, té una sèrie de funcions javascript que estableixen la

connexió i, com al servidor, responen a esdeveniment identificats per un string, quan succeeix algun esdeveniment s'executa la funció que toca i en aquest cas executa codi javascript que mostra les dades rebudes a través del esdeveniment en components html (tot i que és podria fer qualsevol cosa que es vulgui fer amb javascript). Igual que al servidor, el client també pot enviar dades de forma autònoma fent servir les funcions javascript adients.

3.4. DOCKER

Una part molt important d'aquest projecte és la d'executar codi dels usuaris per extreure'n conclusions. Aquest codi pot no estar ben fet de moltes maneres diferents, cosa que pot comprometre la estabilitat del sistema; això fa que es tingui la necessitat d'executar el codi en un entorn segur i, que en cas de mal funcionament, no afecti a la resta del sistema. Una de les possibilitats seria fer servir una màquina virtual, tenir un sistema operatiu amb tot instal·lat dins i provar el codi dins de la màquina, però aquesta opció es costosa tant d'implementar com de mantenir.
Després hi ha varies opcions de software de virtualització que no sigui tant costós, com una màquina virtual, però entre les diferents opcions la que al final s'ha acabat triant per diferents motius (implantació, ús, continuïtat, ...) a estat docker.

Docker és un sistema de màquines virtuals que opera a nivell de sistema operatiu sense treure recursos al sistema. Aquest sistema executa codi en paquets anomenats contenidors; cada contenidor està aïllat de la resta i té la seva pròpia configuració i les seves pròpies llibreries. Tots els contenidors funcionen sota el mateix sistema operatiu i són més lleugers que una maquina virtual convencional. Cada contenidor està creat a partir d'una imatge a les que s'accedeix a partir de repositoris públics.

A més docker t'ofereix un avantatge que fa que sigui molt atractiu, i és que docker és aliè al sistema operatiu, és a dir, tot sistema de dockers configurat que funcioni en un sistema operatiu pot ser migrat a un altre sistema operatiu (amb docker instal·lat) i seguirà funcionant exactament igual. Això facilita molt tant el desenvolupament com la implementació, però requereix que estigui ben configurat.

A l'hora de configurar cada contenidor, és possible que sigui necessari instal·lar dins del docker algunes dependències a nivell de software. Totes aquestes dependències han de venir indicades en un arxiu especificant nom

i versió; després, a l'executar docker, totes les dependències especificades que no estiguin instal·lades s'instal·laran dins del docker. Aquesta instal·lació és aliena al sistema operatiu i no es podrà fer servir fora del docker.

Docker té dos arxius de configuració, docker-compose i dockerfile. Docker-compose s'encarrega de gestionar tots el contenidors del sistema assignant paràmetres importants com ara els ports, les dependències, on es guarden les dades en local entre moltes altres. Dockerfile, per altre banda, s'encarrega de la configuració de cada contenidor individualment, on es poden especificar coses com quines són les comandes per executar-lo, a quin nivell de carpetes correspon on es guarden les seves dades.

Per aquest projecte s'ha fet un contenidor de docker per a cada servei (webhook, mongodb, redis, flask i servidor web), més uns contenidors temporals on s'executa el codi. Cada un d'aquests contenidors està aïllat de la resta i en un principi no poden comunicar-se, però docker permet que els contenidors es comuniquin tant entre ells com a l'exterior si se'ls configura adequadament; aquesta part es pot veure més elaborada en el següent capítol.

4. DISSENY

L'aplicació amb tots els canvis consta de les següents parts:

-Servidor webhook que gestiona els commit a partir de l'api de github.

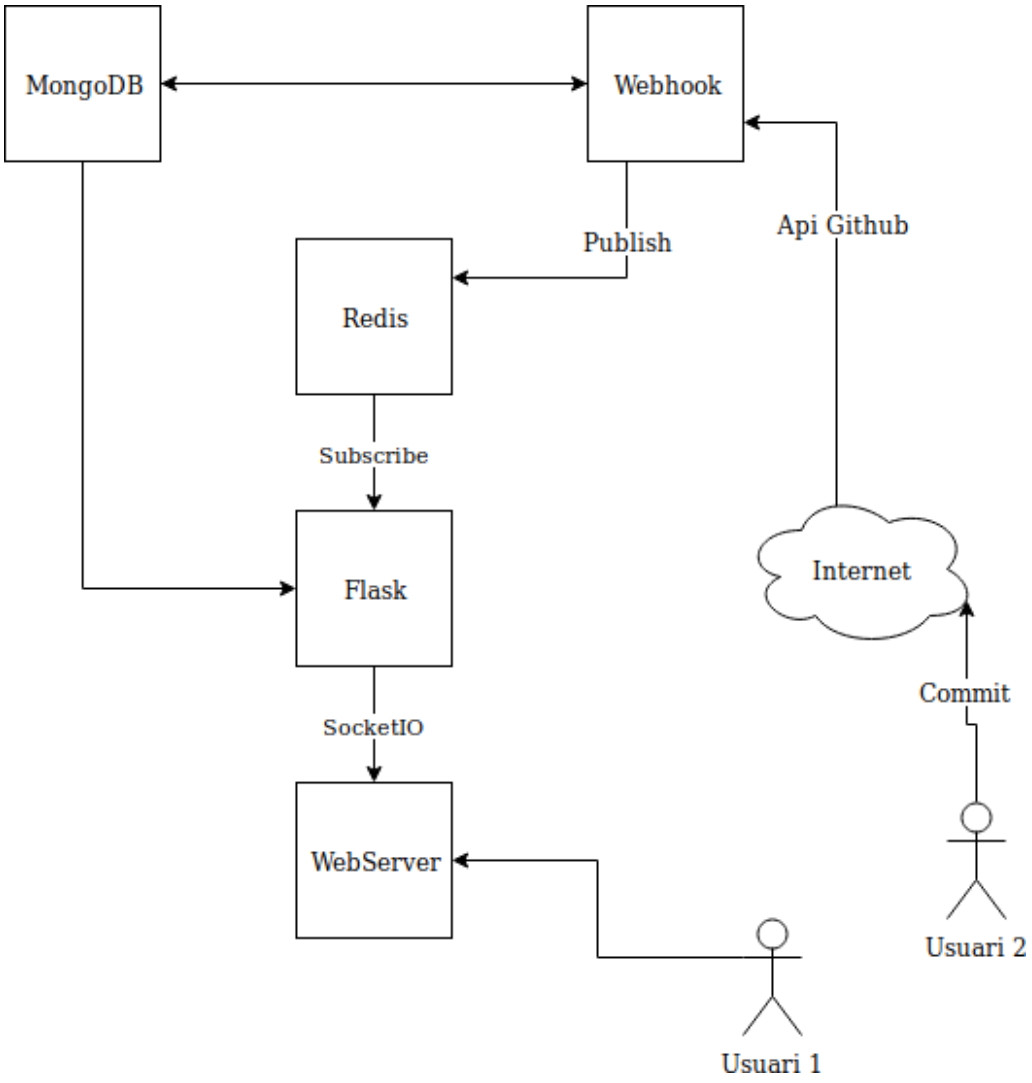
-Servidor mongodb on es guarden les dades.

-Servidor redis que fa de intermediari entre els servidors.

-Servidor flask que gestiona la informació i fa de servidor de sockets per a la interfici d'usuari.

-Servidor html.

Un cop totes les tecnologies han estat introduïdes, l'estructura de la aplicació ha passat a ser aquesta:



3.Esquema de GradeMe amb els canvis efectuats.

On abans hi havia un servei de broadcast, ara hi ha un servidor de redis que en comptes de redistribuir les dades funciona amb un sistema de subscripció i publicació. Ara hi ha un servidor flask que funciona com a servidor de sockets i no fa de servidor web; aquest servidor s'encarrega de subscriure's a redis i enviar la informació rebuda per socketio a les pàgines web. I per últim, un servidor de web on es troba la interfície d'usuari fent peticions al servidor via websockets.

Durant el projecte s'han hagut de prendre certes decisions basades en com i quines tecnologies introduir.

El projecte inicial estava implementat dins de docker, i es va prendre la decisió de que tots els futurs servidors que s'afegissin també estiguessin dins de docker amb els seus contenidors corresponents en comptes de deixar-los fora ja que l'avantatja de la portabilitat i aïllament de docker romandria reduïda si la meitat del sistema està fora del docker.

Hi havia la necessitat d'afegir un broker que substituís a un broadcaster. La llista per triar era bastant llarga, però l'opció triada, la més clara, més coneguda, que s'estudia a la facultat i que a més és codi obert , va ser redis.

A l'hora de triar un servidor de sockets que recollís les dades de redis i les pogués gestionar, l'opció escollida ha sigut flask, seguint la recomanació de Guillem Pascual, ja que la dimensió del projecte no era massa gran i flask és prou potent per satisfer les necessitats que existeixen obtenint a canvi tots els avantatges de simplicitat i versatilitat que ofereix.

Flask també pot actuar com a web server; de fet, per realitzar proves s'ha fet servir també com a tal. Però per tenir el codi més organitzat i per a que en un futur es pugui triar un web server que agradi més, com ara angular, es va triar la opció de no fer servir el mateix servidor per a dues tasques clarament

diferenciades. També estava

l'opció de fer servir django (que a més s'ensenya a la facultat), però django és molt més restrictiu i pren decisions relacionades amb temes com la base de dades, per això es va descartar.

Durant la fase de reconeixement del projecte, es va comentar la necessitat de comunicar frontend amb el servidor fent servir sockets. Javascript ja permet fer servir comunicació amb sockets amb un conjunt de funcions natives anomenades websocket. Amb websocket ja s'hagués pogut fer tot el que es necessitava, però existeix una llibreria javascript no nativa anomenada socketio que facilita molt més l'ús de la comunicació entre sockets, i que a més, fa tasques que garanteixen la comunicació en entorns on websocket hagués suposat molta feina.

Durant la realització d'aquest projecte, la major part de la feina ha estat aprendre a implementar i implementar tot el sistema de dockers amb els seus servidors configurats adientment per a que es puguin comunicar i que tot funcioni correctament. Cadascun dels servidors té les seves complexitats i cadascun requereix uns coneixements diferents. Per començar a implementar sense haver fet cap canvi es va seguir un readme proporcionat per Guillem Pascual amb el que es va poder encendre la aplicació, però sense que funcionés el servidor de webhook.

Abans de començar a fer les implementacions, s'ha de fer una ullada als arxius de configuració del docker ja que demana alguns paràmetres que s'han de preveure i poder no es tenen. Aquest paràmetres són la clau i l'organització de github.

Amb tot el software base necessari instal·lat al sistema la primera prova és iniciar tot i aquí es pot executar o bé dins del docker o bé fora del docker. Executar fora del docker, tot i no ser com s'hauria d'executar en un principi, ofereix certs avantatges per al desenvolupament, temps d'espera reduït i interfície gràfica.

Quan l'aplicació s'executa des de docker, s'han d'executar tots els serveis dins dels seus contenidors alhora, i cada vegada que es fa un canvi en el codi d'alguna part de l'aplicació, s'ha de tancar i tornar a engegar tots els contenidors de docker; a més, en

molts casos, depenent del que s'hagi modificat, tancar i tornar a obrir-ho tot no és suficient per veure els canvis, s'han d'esborrar els contenidors de zero i tornar a instal·lar el software dins del contenidor cosa que fa el procés d'execució des de dins del docker molt més lent. Aquest últim cas no succeïa sempre però prou freqüentment per que fos un inconvenient desenvolupar des de dins del docker.

Per tornar a engegar tot des de zero, primer es paraven tots els processos docker, després s'esborraven totes les imatges des de dins de docker, i per últim s'eliminava l'estructura de contenidors per finalment tornar a instal·lar i executar tot el codi.

Sudo docker rm $(docker
ps -a -q) sudo docker rmi
$(docker images -q)

docker-
compose
down sudo
./build.sh
sudo ./run.sh

Per altre banda, Guillem Pascual tenia una interfície de testeig de commits la qual era força útil ja que es podia veure el resultat d'un commit sense haver de fer-ne un. Això és especialment pràctic ja que evita haver de fer canvis en els ports de la xarxa on s'estigui provant i evita haver de fer realment un commit. Aquesta interfície gràfica només es podia executar desde fora del docker fent més interesant aquesta opció a mode de desenvolupament.

Tot i això, executar amb docker seguia tenint l'avantatge de no haver de tenir res instal·lat al sistema (servidors); i jo, personalment, vaig fer una barreja executant alguns serveis dins de docker i d'altres fora durant el desenvolupament d'aquest treball de fi de grau.

A l'hora de configurar docker s'han de comprovar dos arxius; docker-compose i dockerfile. Pel que fa a docker-compose és important veure quins contenidors depenen d'altres i aixecar-los en l'ordre correcte o no funcionarà. A més, també s'ha de configurar la connexió entre els diferents contenidors indicant-li al docker a quina xarxa pertanyen. La xarxa del docker no és com una xarxa local, és més aviat una

xarxa de docker on cada membre de la xarxa és un contenidor. Els contenidors de docker poden comunicar-se fora de docker però a docker compose s'especifica com ho fan des de dins del docker amb els camps "links" i networks". Pel que fa als ports de cada un dels serveis, s'han deixat els que tenien per defecte ja que de no ser així, pot donar problemes.

Cada servei que s'executa des de codi de l'aplicacio (webhook i flask) ha de tenir el seu propi dockerfile, en aquest dockerfile és pot especificar des d'on s'obre, amb quins paràmetres s'obre, on es guarda la informació i moltes més coses. El funcionament de python fa que a l'hora d'importar arxius python d'altres carpetes sigui bastant estricte; si s'executa un arxiu des de una carpeta a un nivell alt,

després no es pot tornar enrere i pujar per un altre carpeta. Davant d'aquest problema la solució més factible és que tots els programes python executables s'executin des de l'arrel de l'aplicació.

Quan es fa referencia als ports i als hosts als que es connecta un servei, s'ha d'anar amb compte amb si estan o no dins de docker. Normalment, quan es fa una connexió en local a un mateix, es fa dient que quan es connecta a "localhost", sent localhost sempre un mateix. Quan s'executa amb un servei

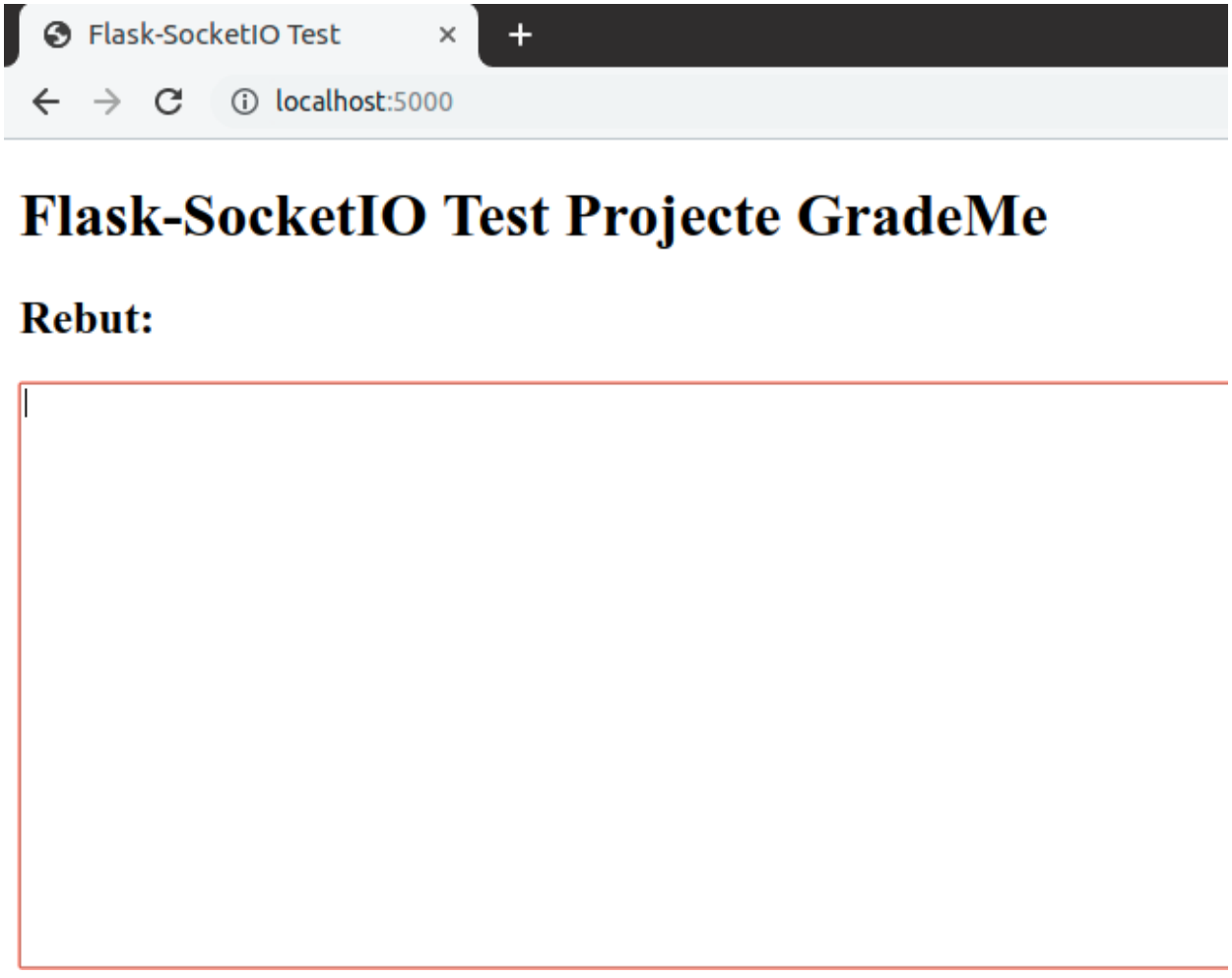
des de dins del docker la expressió “localhost” no serveix, i s’ha d’especificar el nom del contenidor al qual es vol fer la connexió. Aquesta situació però, no passa si s’executa des de fora del docker, on si és pot establir una connexió amb qualsevol servei de dins d’un docker amb “localhost” com a direcció del servidor.

Executar un servei des de dins del docker també comporta altres diferències del comportament per defecte que té quan s’executa des de fora, i és que per defecte, molts serveis s’executen amb els ports oberts; però dins del docker això no és així, i s’ha d’especificar que vols que aquest servei estigui obert a connexions provinents de fora (encara que sigui des de local).

5. PROVES

Al final calia comprovar que amb els canvis i la Refacció s'obtingués informació en una pagina web a partir d'una simulació de commit.

S'ha creat una pàgina web per a fer proves i visualitzar els resultats. Aquesta pàgina web escolta del servidor flask i mostra dins d'un quadre de text la informació que li envien.



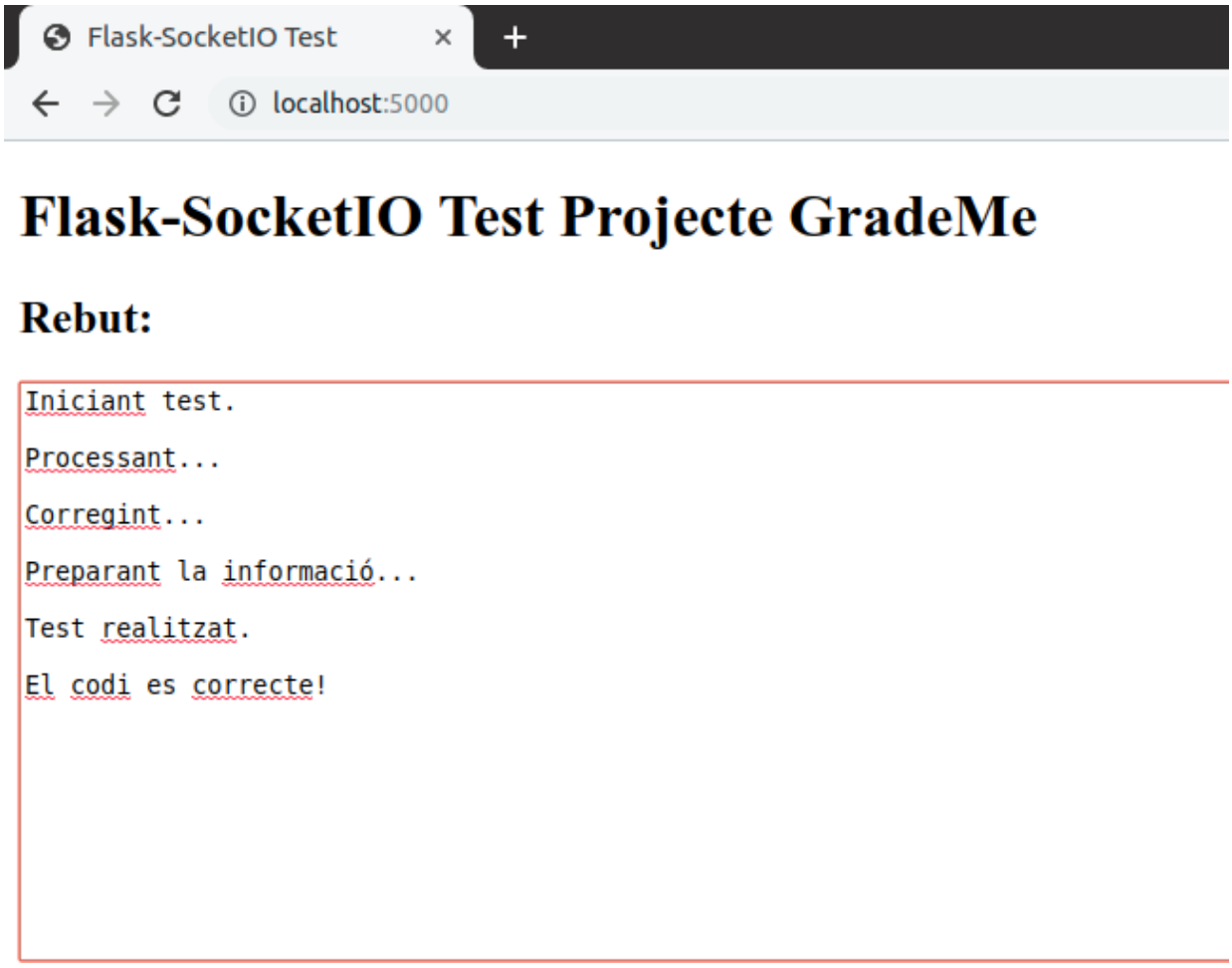
4. Pàgina de prova abans d'executar un test

Es van fer varies proves executant tot l'entorn dins de docker i fent simulacions de commit. Quan webhook rep un commit, s'executen una sèrie de passos; cada un d'aquests passos envia també informació a un canal de redis indicant

l'estat en que es troba el procés de correcció.

Flask rep cadascun d'aquest missatges ja que està subscrit al mateix canal al qual webhook està publicant i, de forma reactiva, s'executa la funció de callback cada vegada que la informació canvia.

A la funció de callback es tracta la informació rebuda i s'envia per socketio als clients. Aquesta funió s'executa cada vegada que es rep un missatge i la informació tractada i enviada serà la que aparegui per la pagina web.



6. Pàgina de proba després d'executar el test

Al final es pot veure per la pàgina la informació tractada provinent del servidor, aquesta es la informació que ha rebut la pàgina web, a través de la seva connexió amb el servidor mitjançant socketio i s'ha tractat amb codi javascript

per posar-la dins d'una caixa de text html. Els missatges no es reben de forma instantània, ja que el procés de correcció de codi triga una mica a fer les coses (especialment l'últim pas).

Quan s'ha acabat el test, al servidor de bases de dades mongoDB s'ha afegit una copia de les dades.

```
mongo    | 2019-06-27T16:14:01.264+0000 I NETWORK  [conn34] end connection 172.1
9.0.1:57036 (2 connections now open)
mongo    | 2019-06-27T16:14:01.274+0000 I NETWORK  [listener] connection accepte
d from 172.19.0.1:57060 #37 (3 connections now open)
mongo    | 2019-06-27T16:14:01.275+0000 I NETWORK  [conn37] received client meta
data from 172.19.0.1:57060 conn: { driver: { name: "PyMongo", version: "3.4.0" }
, os: { type: "Linux", name: "Ubuntu 19.04 disco", architecture: "x86_64", versi
on: "5.0.0-19-generic" }, platform: "CPython 3.7.3.final.0" }
mongo    | 2019-06-27T16:14:34.255+0000 I NETWORK  [listener] connection accepte
d from 172.19.0.1:57088 #38 (4 connections now open)
mongo    | 2019-06-27T16:14:34.255+0000 I NETWORK  [conn36] end connection 172.1
9.0.1:57048 (3 connections now open)
mongo    | 2019-06-27T16:14:34.255+0000 I NETWORK  [conn38] received client meta
data from 172.19.0.1:57088 conn: { driver: { name: "PyMongo", version: "3.4.0" }
, os: { type: "Linux", name: "Ubuntu 19.04 disco", architecture: "x86_64", versi
on: "5.0.0-19-generic" }, platform: "CPython 3.7.3.final.0" }
mongo    | 2019-06-27T16:14:34.256+0000 I NETWORK  [listener] connection accepte
d from 172.19.0.1:57092 #39 (4 connections now open)
mongo    | 2019-06-27T16:14:34.256+0000 I NETWORK  [conn39] received client meta
data from 172.19.0.1:57092 conn: { driver: { name: "PyMongo", version: "3.4.0" }
, os: { type: "Linux", name: "Ubuntu 19.04 disco", architecture: "x86_64", versi
on: "5.0.0-19-generic" }, platform: "CPython 3.7.3.final.0" }
```

7. Informació del servidor de base de dades durant les execucions de diferents tests.

6. LINIES FUTURES:

GradeMe ha estat desenvolupat per Guillem Pascual, aquest treball de fi de grau ha estat una ampliació a la seva feina, però no ha estat suficient per acabar-lo i encara queden per fer diferents tasques.

Crear un servidor de frontend dockeritzat que es comuniqui amb el existent flask fent servir socketio; ja hi ha una pàgina d'exemple per a la comunicació frontend-flask.

La funcionalitat de la pàgina web hauria de ser mostrar els resultats dels test (tant els que s'estan executant com els antics a mode d'historial) per part dels alumnes. Per part dels professors cal fer una interfície de creació de tests i una interfície que permeti veure les dades dels tests realitzats pels alumnes de les seves assignatures. Seria convenient que el sistema d'accés es faci a partir de l'api de github ja que tothom hauria de tenir compte creat i així la gestió de comptes seria independent de la universitat.

Gestionar múltiples connexions. Al servidor flask hi ha una lògica de connexions amb socketio i subscripcions a redis, però no tracta enviar a cada client la informació que li correspon (envia tot a tots)

Recuperar les dades de mongodb i enviar-les als clients. Flask de moment només obté informació de la subscripció a redis, seria convenient afegir una connexió amb el servidor de bases de dades per obtenir informació específica i enviar-li a l'usuari final corresponent.

Personalment recomano que si algú continua aquest projecte i s'ha llegit aquesta memòria abans de fer res ho consulti amb la Universitat de Barcelona que és qui està més al corrent de les coses.

7. CONCLUSIONS

Al final de projecte s'ha aconseguit complir tots els objectius assumits al començament. S'ha fet una refactoriózacio de l'aplicació, afegint els serveis necessaris i fent que es puguin comunicar i s'ha deixat l'aplicació preparada per a que futures intervencions es facin de forma més senzilla i puguin centrar-se en altres parts on no hi ha gaire contingut.

A l'hora d'introduir redis s'ha pogut comprovar ho bé que interactua amb python i la quantitat elevada de documentació a la xarxa fa que sigui molt fàcil, ja no només de fer servir sinó d'instal·lar o introduir dins d'un contenidor docker.

Pel que fa flask, tot i ser un servidor propi el qual en un principi no té cap ús concret assignat i se l'hi ha hagut d'afegir totes les funcionalitats i les interaccions amb altres serveis no ha suposat cap problema a l'hora d'integrar-se amb la resta de l'aplicació.

Socketio i flask s'entenen molt bé i hi ha molta documentació a la xarxa, cosa que ha fet que sigui una bona decisió fer-los servir alhora.

Afegir cada servei dins de docker fent que l'aplicació funcioni 100% dins de docker és molt fiable i fa que l'aplicació estigui més ordenada i sigui més fàcil de desenvolupar i implementar en diferents entorns.

Cada uns dels canvis realitzats ha complert les exigències que es demanaven i s'ha realitzat tot el projecte dins del temps estimat.

8. GLOSARI

Backend: És la part del software que gestiona la capa d'accés a les dades.

Broadcaster: Sistema d'enviament de dades en el qual les dades s'envien a tots els participants sense excepció.

Commit: Modificació que es fa a un arxiu o conjunt d'arxius d'un repositori.

Docker: Sistema de màquines virtuals molt lleuger que executa software específic en paquets anomenats contenidors.

Endpoints: Extrems inici o final en el procés d'una comunicació.

Frontend: És la part del software que gestiona la capa de presentació de les dades a l'usuari final.

Github: Organització americana que proveeix un servei d'allotjament de software en desenvolupament.

Javascript: És un llenguatge de programació compilat i d'alt nivell utilitzat majoritàriament en pàgines web.

Maquina virtual: Simulació d'un sistema informàtic.

Middleware: En aplicacions distribuïdes, middleware és un software que permet la comunicació entre les diferents parts i s'encarrega de gestionar les dades.

Python: És un llenguatge de programació interpretat d'alt nivell de propòsit general.

Refactor: Reestructuració del codi per millorar el seu rendiment sense canviar

el comportament extern.

Repositori: Sistema informàtic on s'emmagatzema informació per a ús compartit entre diferents usuaris.

Sockets: Un socket de xarxa és un endpoint a l'hora d'enviar o rebre dades a nivell de software.

9. BIBLIOGRAFIA

https://redis.io/documentation

https://tech.webinterpret.com/redis-notifications-python/

https://hub.docker.com/_/redis/

https://socket.io/docs/

https://docs.docker.com/compose/gettingstarted/

https://docs.docker.com/engine/reference/builder/

http://flask.pocoo.org/docs/1.0/