Treball final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica

Universitat de Barcelona

Elixir: anàlisis i estudi del seu rendiment

Autor: Joaquim Salvadó Gubert

Director: Oscar Amoros Huguet

Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 21 de juny de 2020

Resum en català

Les aplicacions web destinades a servir un gran volum d'usuaris, implementen backends capaços de rebre i generar una gran quantitat de peticions HTTP. El rendiment d'aquest backend, no només depén del maquinari utilitzat, si no que també depèn molt del llenguatge de programació amb el que estigui implementat.

Al buscar comparatives de rendiment entre llenguatges de programació a internet no s'hi troba gran cosa i moltes comparacions acaben sent parcials.

Aquest treball investiga el rendiment de diversos llenguatges de programació, quan s'utilitzen per a crear un backend d'una aplicació web. El mateix backend s'ha implementat en Java, Python i Elixir, i s'han dissenyat tests de càrrega per mesurar la resposta dels backends. Amb els resultats dels tests hem pogut descobrir quin llenguatge de programació ha tingut una millor resposta i saber quina és la millor tecnologia, d'entre els 3 llenguatges escollits, per a la creació de backends.

Resum en anglès

Web applications designed to serve a large volume of users, implement backends capable of receiving and generating a large number of HTTP requests. The performance of this backend not only depends on the hardware used, but also a lot on the programming language with which it is implemented.

When looking for performance comparisons between programming languages on the Internet, you don't find much and many comparisons end up being partial.

This work investigates the performance of various programming languages, when used to create a backend of a web application. The same backend has been implemented in Java, Python and Elixir, and load tests have been designed to measure the response of the backends. With the test results we were able to find out which programming language had the best response and know which is the best technology, among the 3 languages chosen, for creating backends.

Resum en castellà

Las aplicaciones web destinadas a servir un gran volumen de usuarios, implementan backends capaces de recibir y generar una gran cantidad de peticiones HTTP. El rendimiento de este backend, no sólo depende del hardware utilizado, si no que también depende mucho del lenguaje de programación con el que esté implementado.

Al buscar comparativas de rendimiento entre lenguajes de programación en internet no se encuentra gran cosa y muchas comparaciones terminan siendo parciales.

Este trabajo investiga el rendimiento de varios lenguajes de programación, cuando se utilizan para crear un backend de una aplicación web. El mismo backend ha sido implementado en Java, Python y Elixir, y se han diseñado tests de carga para medir la respuesta de los backends. Con los resultados de los tests hemos podido descubrir qué lenguaje de programación ha tenido una mejor respuesta y saber cuál es la mejor tecnología, de entre los 3 lenguajes escogidos, para la creación de backends.

ÍNDEX

1. Introducció .... p. 6

1.1. Antecedents, motivacions i context .... p. 6

1.2. Objectius .... p. 7

1.3. Planificació .... p. 8

2. Desenvolupament .... p. 9

2.1. Descripció del projecte .... p. 9

2.2. Introducció a la implementació .... p. 9

2.2.1. Programació funcional .... p. 10

2.2.2. Programació concurrent .... p. 12

2.2.3. Model concurrent d’actors .... p. 13

2.2.4. Eines per al desenvolupament .... p. 15

2.2.4.1. Behaviors .... p. 15

2.2.4.2. Elixir Supervisor Tree .... p. 18

2.3. Implementació .... p. 19

2.3.1. Objectiu de la implementació .... p. 19

2.3.2. Creació de les aplicacions .... p. 19

2.3.2.1. Time API .... p. 20

2.3.2.2. Marketplace API .... p. 20

2.3.3. Entorns .... p. 28

2.3.4. Tests de càrrega .... p. 29

3. Resultats .... p. 30

3.1. Dur a terme els tests de càrrega .... p. 30

3.2. Time API en localhost .... p. 32

3.2.1. Resultats Java .... p. 35

3.2.2. Resultats Python .... p. 36

3.2.3. Resultats Elixir .... p. 37

3.2.4. Comentaris dels resultats .... p. 37

3.3. Time API en servidor .... p. 38

3.3.1. Resultats Java .... p. 39
3.3.2. Resultats Python .... p. 41
3.3.3. Resultats Elixir .... p. 42
3.3.4. Comentaris dels resultats .... p. 44
3.4. Marketplace API en servidor .... p. 44
3.4.1. Resultats Java .... p. 45
3.4.2. Resultats Python .... p. 48
3.4.3. Resultats Elixir .... p. 50
3.4.4. Comentaris dels resultats .... p. 51
3.5. Resultats obtinguts i objectius .... p. 55

4. Conclusions .... p. 56

5. Bibliografia .... p. 58

6. Apèndix .... p. 59

1. INTRODUCCIÓ

El món de la programació és molt gran i engloba moltes especialitats diferents, una d'aquestes especialitats és la programació web. La programació web està dividida en dues parts, el frontend i el backend. El frontend és la part que treballa com es visualitza i interactua amb la informació, en canvi, el backend treballa tota la lògica del sistema. Tant el frontend com el backend es poden crear fent servir diferents llenguatges de programació i fins i tot es poden arribar a combinar llenguatges, fent servir llenguatges específics per parts concretes del frontend o del backend que es vol crear.

Aquest treball es centra en el backend de les aplicacions web.

1.1 ANTECEDENTS, MOTIVACIONS I CONTEXT

Com a programador m'he trobat en situacions en les quals gent que no sap res de programació em pregunta quina diferència hi ha entre llenguatges de programació. És molt complicat donar una resposta correcte que puguin entendre, ja que no depèn només del llenguatge, sinó del tipus de problema que volem resoldre en cada ocasió. Si volem fer un motor gràfic es pot fer servir C++, en canvi si volem crear un servidor web es pot fer servir Node.js i si es vol tractar grans volums de dades es pot fer servir Python i les seves llibreries. La resposta a aquesta pregunta no és blanc o negre.

Buscar diferències entre llenguatges de programació a internet no ajuda, ja que les classificacions es fan en funció de les característiques internes dels llenguatges. Es pot saber si els llenguatges són orientats a objectes, si són imperatius, si són generics, etc. A més, moltes de les comparacions es fan en funció de la facilitat per fer servir el llenguatge o les preferències personals del autor.

Per això se'm va ocórrer escollir un problema concret, la programació encarada al web, i escollir diferents llenguatges de programació per a poder-los comparar resolent exactament el mateix problema: com són capaços de gestionar peticions

HTTP i en quin punt comencen a fallar, és a dir, es compararà l'eficiència dels llenguatges. Els llenguatges escollits per ser comparats són Elixir, Java i Python.

La resposta al problema de l'eficiència té més implicacions de les que semblen a primera vista. El rendiment d'un llenguatge de programació és molt important a l'hora d'escollir un llenguatge per a un projecte. Una empresa pot tenir moltes despeses en servidors pel fet que el llenguatge que van escollir no és eficient i no suporta la quantitat de peticions que reben. Un llenguatge de programació eficient vol dir que els servidors on s'executi el codi aguantaran més càrrega de peticions. Per tant caldrà menys servidors i hi haurà una despesa menor en electricitat, reduint els costos de l'empresa.

1.2 OBJECTIUS

L'objectiu d'aquest TFG és poder quantificar quan millor és un llenguatge de programació respecte a un altre. El llenguatge protagonista d'aquest estudi és Elixir, i el compararem amb Java i Python.

No tots els llenguatges de programació fan el mateix i tots tenen punts forts i fluixos depenent del problema que s'intenta solucionar. Per exemple, Python funciona molt bé per a treballar amb grans volums d'informació i Java és molt útil per a crear aplicacions per Android. Però en aquest TFG es posa a prova aquests llenguatges de programació en l'àmbit d'aplicacions web, més concretament, el backend d'aplicacions web. Una aplicació web és un programa que fa ús d'HTTP (Hypertext Transfer Protocol) i altres protocols per a guardar, processar i respondre a les peticions fetes per clients en una xarxa, com Internet.

Per a quantificar l'eficiència dels llenguatges de programació farem servir 2 aplicacions diferents, una simple i una més complexa, i les crearem en els 3 llenguatges de programació de forma que a nivell funcional sigui indiferent el llenguatge. Llavors realitzarem tests de càrrega per a veure el rendiment de les aplicacions i poder comparar els resultats.

1.4 PLANIFICACIÓ

El TFG estarà dividit en 2 parts, una part teòrica en la qual s'explica les característiques d'Elixir i els motius que fan que sigui un llenguatge molt eficient; i una part pràctica on es realitzaran el conjunt d'experiments per poder arribar a la solució a la pregunta plantejada: quan millor és Elixir respecte a Java i Python.

Es pot veure reflexat com s’ha dividit la feina del projecte i en quines parts s’ha estat treballant en el següent diagrama de Gantt:

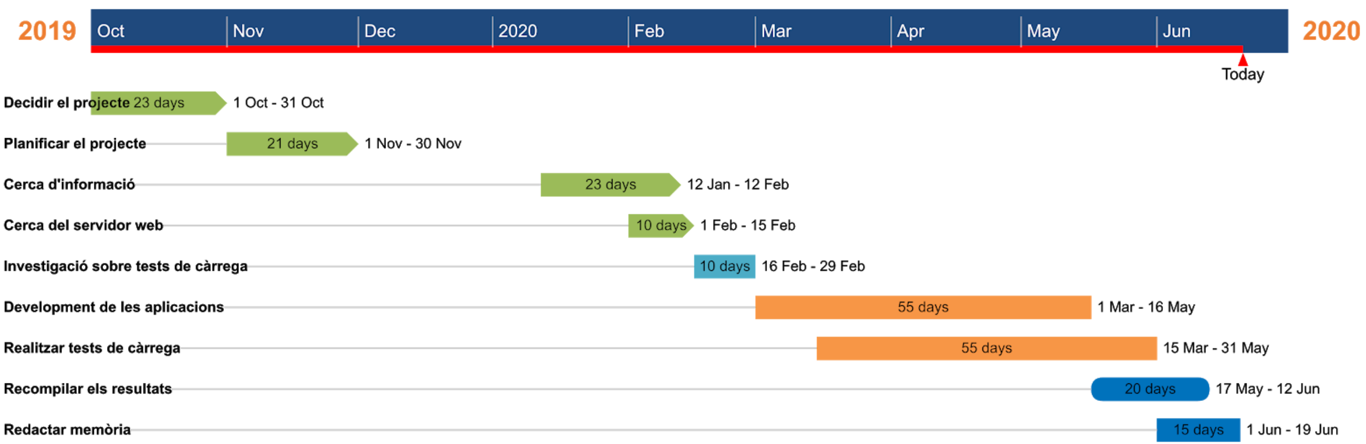


Figura 1: Planificació del projecte

2. DESENVOLUPAMENT

2.1 DESCRIPCIÓ DEL PROJECTE

El projecte consisteix en crear 2 aplicacions web, TimeAPI i MarketplaceAPI, en Java, Python i Elixir de forma que a nivell funcional les aplicacions fan exactament el mateix sense importar el llenguatge escollit. Un cop tenim les aplicacions creades, necessiten un entorn on executar-se. Els entorns escollits han estat 'localhost' i un servidor. Quan les aplicacions i entorn estan a punt, s'aplica un test de càrrega a les aplicacions. Els tests de càrrega simulen activitat d'usuaris i generen peticions HTTP als endpoints de l'aplicació. Un cop es tinguin els resultats dels tests de càrrega podrem comparar-los i saber el rendiment de cadascun dels llenguatges de programació.

2.2 INTRODUCCIÓ A LA IMPLEMENTACIÓ

Elixir és un llenguatge de programació funcional, concurrent i de propòsit general que s'executa a sobre la màquina virtual d'Erlang (BEAM). Elixir està escrit sobre Erlang i comparteix les mateixes abstraccions per desenvolupar aplicacions distribuïdes i tolerants a errors. Va ser creat per José Valim i la primera versió va aparèixer l'any 2011. El seu objectiu era crear un llenguatge que permetis una alta extensibilitat i productivitat en la màquina virtual d'Erlang mantinguent compatibilitat amb l'ecosistema d'Erlang.

Erlang és un llenguatge de programació concurrent i un sistema d'execució que inclou una màquina virtual i biblioteques. Va ser dissenyat en la companyia Ericsson per a realitzar aplicacions distribuïdes, tolerants a fallades i de funcionament ininterromput. Originalment, Erlang era un llenguatge propietari d'Ericsson, però va ser cedit com a programari de codi obert en 1998. Sense dubtes una cosa que fa molt especial d'Erlang és com maneja la concurrència. Erlang no maneja la concurrència amb fils com ens té acostumat C, C++ o Java. Erlang soluciona la programació concurrent mitjançant el model d'actors, que explicarem més endavant.

Elixir aprofita tots els avantatges que li proporciona el fet d'executar-se en la màquina virtual d'Erlang, com la concurrència i el sistema basat en actors; però és un llenguatge més modern, fet amb la idea de fer més fàcil interactuar amb la màquina virtual d'Erlang.

2.2.1 PROGRAMACIÓ FUNCIONAL

Que vol dir que Elixir sigui un llenguatge funcional? En informàtica, la programació funcional és un paradigma de programació declarativa basat en l'ús de veritables funcions matemàtiques. En aquest estil de programació les funcions són ciutadanes de primera classe, perquè les seves expressions poden ser assignades a variables com es faria amb qualsevol altre valor; a més de que poden crear-se funcions d'ordre superior [1].

La programació funcional té les seves arrels en el càlcul lambda [2], un sistema formal desenvolupat en els anys 1930 per a investigar la naturalesa de les funcions, la naturalesa de la computabilitat i la seva relació amb la recursió. Els llenguatges funcionals prioritzen l'ús de recursivitat i l'aplicació de funcions d'ordre superior per a resoldre problemes que en altres llenguatges es resoldrien mitjançant estructures de control (per exemple, cicles). Molts llenguatges de programació funcionals poden ser vistos com a elaboracions del càlcul lambda.

Alguns llenguatges funcionals també busquen eliminar els efectes secundaris; en contrast amb la programació imperativa, que emfatitza els canvis d'estat mitjançant la mutació de variables. Això significa que dues expressions sintàctiques idèntiques (crides a rutines, per exemple) poden produir significats diferents si depenen d'alguna cosa que ha mutat. En els llenguatges funcionals més estrictes, en contrast, el valor generat per una funció depèn exclusivament dels arguments que es passen a la funció. En eliminar els efectes secundaris es pot entendre i predir el comportament d'un programa molt més fàcilment. Aquesta és una de les principals motivacions per a utilitzar la programació funcional. L'anterior també pot ser aprofitat per a dissenyar estratègies d'avaluació que evitin repetir el còmput d'expressions abans vistes, estalviant temps d'execució.

Els programes escrits en un llenguatge funcional estan constituïts únicament per definicions de funcions, entenent aquestes no com a subprogrames clàssics d'un llenguatge imperatiu [3], sinó com a funcions purament matemàtiques, en les quals es verifiquen certes propietats com la transparència referencial (el significat d'una expressió depèn únicament del significat de les seves subexpressions), i per tant, la manca total d'efectes col·laterals.

Altres característiques pròpies d'aquests llenguatges són la no existència d'assignacions de variables i la falta de construccions estructurades com la seqüència o la iteració, el que obliga en la pràctica al fet que totes les repeticions d'instruccions es duguin a terme per mitjà de funcions recursives.

Existeixen dues grans categories de llenguatges funcionals: els funcionals purs i els híbrids. La diferència entre els dos consisteix en el fet que els llenguatges funcionals híbrids són menys dogmàtics que els purs, en admetre conceptes presos dels llenguatges imperatius, com les seqüències d'instruccions o l'assignació de variables. En contrast, els llenguatges funcionals purs tenen una major potència expressiva, conservant alhora la seva transparència referencial, cosa que no es compleix sempre amb un llenguatge funcional híbrid. Erlang i Elixir són llenguatges funcionals híbrids.

La raó per la qual els llenguatges funcionals són millors per al processament paral·lel es deu al fet que normalment eviten un estat mutable. L'estat mutable és l'arrel de tot el mal en el context del processament paral·lel; fan que sigui molt fàcil córrer en condicions de cursa (race conditions) quan es comparteixen recursos entre processos concurrents. La solució a les condicions de cursa comporta mecanismes de bloqueig i sincronització que provoquen una sobrecàrrega de temps d'execució, ja que els processos esperen que els recursos es comparteixin, i una major complexitat del disseny, ja que tots aquests conceptes solen ser profundament lligats a aplicacions concurrents.

La programació funcional no permet obtenir programes més ràpids, per regla general. El que fa és facilitar una programació simultània i paral·lela. Hi ha dues claus principals:

- Evitar un estat mutable tendeix a reduir el nombre de coses que poden funcionar malament en un programa i més encara en un programa concurrent.
- Evitar primitives de sincronització basades en memòria compartida i basada en bloqueig a favor de conceptes de nivell superior, això tendeix a simplificar la sincronització entre fils de codi.

La programació funcional promou un estil de programació que ajuda als desenvolupadors a escriure codi que sigui curt, concís i que es pugui mantenir.

2.2.2 PROGRAMACIÓ CONCURRENT

Que vol dir que Elixir sigui un llenguatge concurrent? En informàtica, concurrència és la capacitat de parts diferents o unitats d'un programa, algoritme, o problema per ser executat fora d'ordre o en ordre parcial, sense afectar el resultat final. Això permet una execució paral·lela de les unitats concurrents, els quals poden millorar significativament la velocitat global de l'execució en sistemes multiprocessador i multinucli. En termes tècnics, concurrència refereix a la propietat de descompondre's d'un programa, algoritme, o problema en components o unitats que siguin independents de l'ordre en què s'executen.

Un programa paral·lel és aquell que executa parts del seu codi seqüencial, en paral·lel. Per a poder fer això, utilitza una multiplicitat de maquinari computacional (per exemple, diversos nuclis de processador) per realitzar un càlcul més ràpidament. L'objectiu és arribar a la resposta més ràpidament, delegant diferents parts de la computació a diferents processadors que s'executen al mateix temps. Per contra, la programació és més complexa, ja que cal emprar tècniques d'estructuració de programes, amb les quals es defineix l'asincronisme i sincronisme de l'execució del codi. Conceptualment, els fils de control s'executen "al mateix temps"; és a dir, l'usuari veu els seus efectes entrellaçats. Si realment s'executen al mateix temps o no és un detall d'implementació; un programa concurrent es pot executar en un sol processador mitjançant execució entrellaçada o en diversos processadors físics.

Dues coses concurrents tenen contextos d'execució independents, però no sempre correran en paral·lel. Si tenim 2 tasques concurrents que necessiten l'ús de CPU i les executem amb un nucli CPU, no s'executaran en paral·lel. La concurrència no sempre significa que l'execució sigui més ràpida.

A Elixir, els processos són contextos separats (aïllament) i podem tenir centenars de milers de processos en una sola CPU. Si la nostra màquina té diversos nuclis, Elixir executarà processos en cadascun d'ells en paral·lel. Els processos permeten a Elixir fer ús de la concurrència i del paral·lelisme.

En la concurrència tradicional ens trobem amb dos problemes comuns: la incoherència de dades i el deadlock (punt mort). Per resoldre la incoherència de dades, podem aplicar mecanismes de bloqueig (mutex / semàfors) per garantir que només un fil a la vegada pugui modificar els estats dels objectes. Sobre el deadlock, succeeix quan l'estat bloquejat de dos fils A i B, depèn exclusivament l'un de l'altre. Només A desperta B i només B desperta a A. Per a que es doni el deadlock, en algun punt, els dos fils han de poder estar bloquejats simultàniament. En aquest punt, mai es despertaran. Aquests dos problemes fan que el multithreading sigui difícil d'implementar i de depurar.

Elixir soluciona els problemes de deadlock i incoherència de dades mitjançant el model d'actors.

2.2.3 MODEL CONCURRENT D'ACTORS

El model d'actors és un model de computació concurrent que va aparèixer l'any 1973. Defineix un seguit de normes sobre com les components d'un sistema han d'interactuar en un entorn computacional concurrent.

En el model d'actors tot pot ser un actor, i un actor és una unitat fonamental de computació que pot realitzar les següents accions:

- crear un nou actor
- enviar un missatge
- decidir com gestionar un missatge rebut

Les propietats d'un actor son:

- Missatge: és una unitat de comunicació entre els actors i hauria de ser immutable.
- Bústia: s'utilitza per emmagatzemar missatges, perquè els estats de l'actor no són modificats per altres actors, de manera que envien missatges a un altre actor.
- Referència d'actor: cada actor té una referència que és la forma d'identificar-se entre els actors, es comparteix lliurement entre els actors mitjançant el missatge.
- Dispatcher: és un component que assegura que els actors amb bústies no buides acabin sent gestionats per un fil específic i que aquests missatges es processin.

Els actors tenen un pes lleuger i és molt fàcil crear milions, ja que agafen menys recursos que els fils. Un actor té un estat privat propi i una bústia (similar a la cua de missatges). La bústia conté el missatge que ha rebut l'Actor d'altres persones, aquí el missatge significa estructures de dades senzilles immutables. Els missatges es processen per ordre FIFO [4]. Els actors estan aïllats per naturalesa i no comparteixen memòria. L'estat privat de l'actor només es pot canviar processant un missatge, només pot gestionar un missatge alhora i funcionen de manera asincrònica, vol dir que un actor mai no espera cap resposta d'un altre actor. Els actors interactuen entre ells només mitjançant missatges.

Per fer front als errors, Elixir proporciona supervisors que descriuen com reiniciar parts del sistema quan es produeixen errors, tornant a un estat inicial conegut que està garantit que funciona.

Utilitzant el model d'Actor, es poden crear sistemes que es curin de forma autònoma, és a dir, sistemes tolerants a fallades. El Primer Actor creat el podem anomenar com a mestre, i crea diversos altres actors, que estaran supervisats per el mestre. Si un actor mor o no respon, el seu supervisor pot retornar-lo a un estat estable.

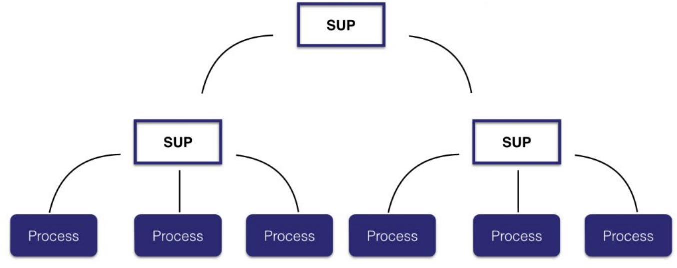


Figura 2: Diagrama de supervisió de processos

A Elixir cada un dels mòduls és una aplicació separada i individual. Després, cada una d'aquestes aplicacions té un procés principal que solen ser de tipus supervisor, encarregats d'iniciar i supervisar altres processos. Quan un procés supervisat mor, el supervisor el reinicia.

Escalabilitat

Tot el codi executat en Elixir funciona dins de fils lleus d'execució (anomenats processos) aïllats i que intercanvien informació mitjançant missatges.

A causa de la seva lleugeresa, no és rar tenir centenars de milers de processos executats simultàniament a la mateixa màquina. L'aïllament permet que els processos es puguin recollir de brossa (garbage collector [5]) de forma independent, reduint les pauses a tot el sistema i utilitzant tots els recursos de la màquina de la manera més eficaç possible (escala vertical [6]).

Els processos també poden comunicar-se amb altres processos en diferents màquines de la mateixa xarxa. Això proporciona els fonaments per a la distribució, permetent als desenvolupadors coordinar el treball en diversos nodes (escala horitzontal [7]).

2.2.4 EINES PER AL DESENVOLUPAMENT

2.2.4.1 BEHAVIORS

Els behaviors proporcionen una manera de:

- Definir un conjunt de funcions que han de ser implementades per un mòdul.
- Assegurar-nos que un mòdul implementi totes les funcions d'aquest conjunt.

Podem pensar en els behaviors com a interfícies en llenguatges orientats a objectes.

A Elixir tenim un behavior anomenat GenServer. Un GenServer és un procés com qualsevol altre procés d'Elixir i es pot utilitzar per mantenir l'estat, executar codi de manera asincrònica, etc. L'avantatge d'utilitzar un procés de servidor genèric (GenServer) implementat mitjançant aquest mòdul és que tindrà un conjunt estàndard de funcions d'interfície i inclourà funcionalitats per al seguiment i la informació d'errors. També cabrà en un arbre de supervisió, que s'explicarà més endavant.

Un GenServer s'encarrega de gestionar el model d'actors de forma senzilla. Un procés que faci ús de GenServer tindrà gestionada tota la feina d'enviar missatges de forma síncrona i asíncrona, rebre missatges i modificar l'estat intern del procés. La idea darrere el GenServer és senzilla. S'inicien processos que tenen un estat, llavors per a cada missatge que rebin realitzen una acció que pot modificar el seu estat i pot generar una resposta.

Supervisors

Software escrit en Elixir és tolerant als errors. Això vol dir que un sistema pot continuar funcionant tot i trobar-se amb errors que el desenvolupador no ha anticipat. El concepte clau per aconseguir que un sistema sigui tolerant als errors és la idea dels supervisors i de l'arbre de supervisió. Un supervisor és un procés el qual la seva feina és fer un seguiment d'altres processos (els seus fills) de forma que quan aquests processos moren, el supervisor n'és informat i pot realitzar l'acció apropiada al respecte.

Per entendre que fan els supervisors exemplificarem una situació en la qual tindrem codi no supervisat. Imaginem-nos que tenim un actor que està esperant missatges. Quan aquest actor rep un missatge agafa el contingut i el transforma de string a integer. El codi seria el següent:

```
def handle_call(message, _from, state) do
  i = String.to_integer(message)
  {:reply, i, state}
end
```

Figura 3: Bústia d'un actor

Un exemple d'execució correcte per part del procés seria quan el missatge amb el format "100". En aquest cas el procés llegiria l'string "100" i el convertiria a l'integer 100 i funcionaria bé. Però que passaria si el missatge fos "Joaquim". Aquest és un string que no es pot convertir a integer. El procés retornaria un error i moriria. Els supervisors estan per tornar a aixecar aquests processos que moren de forma automàtica. Degut a això, Elixir té la filosofia "let it crash", és a dir, deixar que el procés mori per després tornar-lo a aixecar. Això ens permet no haver de pensar en gestionar tots els possibles casos en què alguna cosa pot sortir malament. Si alguna cosa surt malament, es deixa morir al procés i es torna a aixecar.

Tots els supervisors tenen una llista de fills, aquests són els processos que supervisa. Un supervisor pot crear processos treballadors i processos supervisors. Hi ha diferents estratègies per reiniciar els processos un cop han mort, depenent del que el desenvolupador vulgui fer amb l'aplicació. Les estratègies bàsiques són:

- Un per un: si el procés mor, només aquest procés és reiniciat.
- Un per tots: si un procés mor, tots els processos del supervisor també moren. Després són tots reiniciats. Aquesta estratègia és útil quan els processos depenen els uns dels altres.
- La resta per un: si un procés mor, la resta de processos que han estat iniciats després d'aquest procés moren. Després el procés que ha mort i la resta de processos fills són reiniciats.
- El simple per un: aquesta estratègia és igual que "un per un" però en un context dinàmic. Els processos fills es creen en funció de la necessitat de crear-los, com per exemple, un gran nombre de peticions en un servei.

Workers

Els workers són els processos encarregats de fer la feina real, el que vols que la teva aplicació faci. No poden crear fills.

2.2.4.2 ELIXIR SUPERVISOR TREE

Elixir porta un control sobre tots els processos mitjançant el seu arbre de supervisió. Totes les aplicacions fetes amb Elixir tenen un supervisor de nivell superior que inicia els altres processos de l'aplicació. Aquests altres processos poden ser supervisors, que a la vegada tenen els seus propis fills. L'arbre de supervisió d'una aplicació té la següent forma:

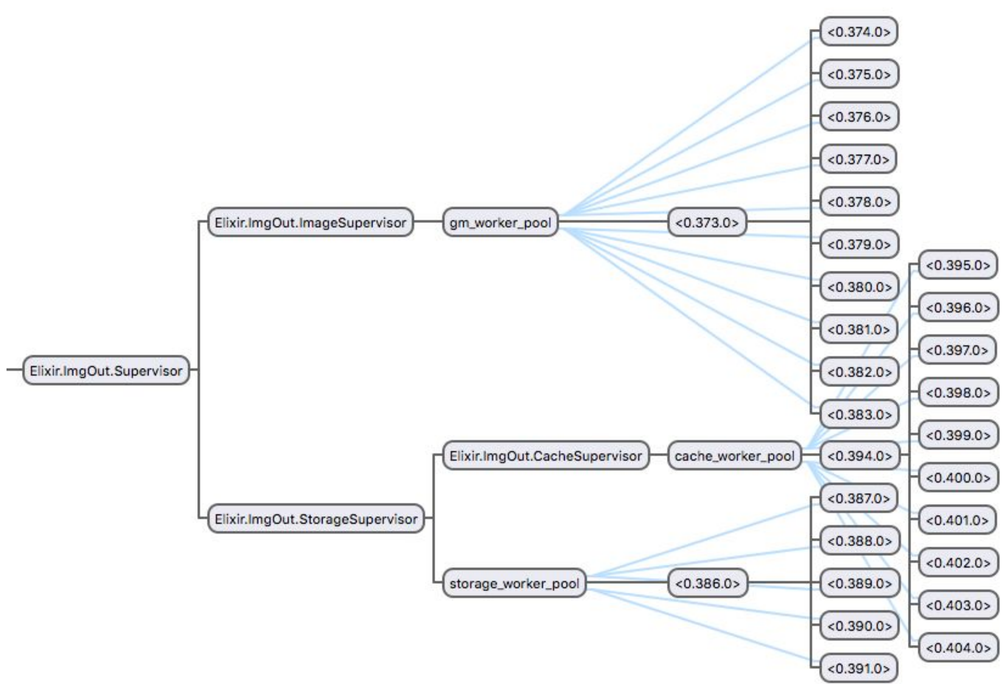


Figura 4: Arbre de supervisió de Elixir

És una eina molt poderosa. En aplicacions complexes és important tenir un control sobre tots els processos que es tenen iniciats i quin procés els supervisa.

2.3 IMPLEMENTACIÓ

2.3.1 OBJECTIU DE LA IMPLEMENTACIÓ

L'objectiu d'aquest apartat és crear les aplicacions que volem comparar i tot el material necessari per a poder executar-les en un servidor. Això inclou la creació de les aplicacions per cada llenguatge de programació i assegurar-se que les aplicacions tenen el comportament correcte. Llavors realitzarem tests de càrrega en aquestes aplicacions per veure el seu rendiment i poder comparar l'eficiència dels llenguatges.

2.3.2 CREACIÓ DE LES APLICACIONS

Per a comparar el rendiment de diferents llenguatges de programació s'han creat dues aplicacions diferents i s'ha repetit cada aplicació en els 3 llenguatges de programació escollits. La primera aplicació que s'ha creat ha estat molt senzilla i ha servit per a poder dur a terme totes les tasques que es pretenien realitzar. És a dir, s'ha fet servir l'aplicació senzilla per a testejar que tots els passos a realitzar fins a obtenir els resultats del test de càrrega es podien realitzar. Llavors, un cop sabem que podem fer tot el que volem fer, ho farem amb l'aplicació complexa.

Les aplicacions s'han creat en Python i Elixir, fent servir Visual Studio Code, i en Java, fent servir Eclipse.

En el transcurs de la memòria no s'aprofunditza molt en els detalls sobre com funciona el codi de cada aplicació en cada llenguatge. El fet de com s'ha creat les aplicacions té poca rellevància i no aporta cap informació útil per assolir els objectius del treball. El que hem de saber és que, a nivell de la lògica, les aplicacions fan exactament el mateix. Amb aquesta premisa podem comparar els llenguatges sabent que tenim les mateixes aplicacions en diferents llenguatges de programació.

2.3.2.1 TIME API

TimeAPI és una API que només té 1 endpoint [8] que retorna el temps exacte en el moment de la petició. És una aplicació senzilla que bàsicament serveix per comprovar si els objectius d'aquest treball es poden assolir.

El seu endpoint és:

- GET /time → retorna el temps en el moment de la petició

Els llenguatges amb els quals s'ha creat aquesta aplicació són:

Python (https://github.com/icosi/Python-Time-API)
Per a crear l'API en Python s'ha fet servir Django, un framework gratuït i open source que permet crear aplicacions web amb molta facilitat.

Java (https://github.com/icosi/Java-Time-API)
Per a crear l'API en Java s'ha fet servir Spark, un framework gratuït i open source que permet crear aplicacions web amb molta facilitat.

Elixir (https://github.com/icosi/Elixir-Time-API)
Per a crear l'API en Elixir no s'ha fet servir cap framework. S'ha creat l'API creant un projecte des de zero fent servir les eines que el llenguatge ofereix.

2.3.2.2 MARKETPLACE API

MarketplaceAPI és una API que serveix per comprar productes. Funciona de forma que hi han usuaris i productes. Cada usuari té la seva cistella de la compra i podrà afegir-hi els productes que vulgui. Un cop l'usuari tingui tots els productes que vol comprar, podrà 'pagar' la cistella. És una versió molt simplificada del que ve a ser una botiga online.

Creació de la base de dades
Hi han 4 models de dades diferents: Usuari, Cistella, Producte i CartProductRelation. Amb això n'hi ha prou per a crear les funcionalitats que es necessiten. Les relacions i paràmetres dels models són les següents:

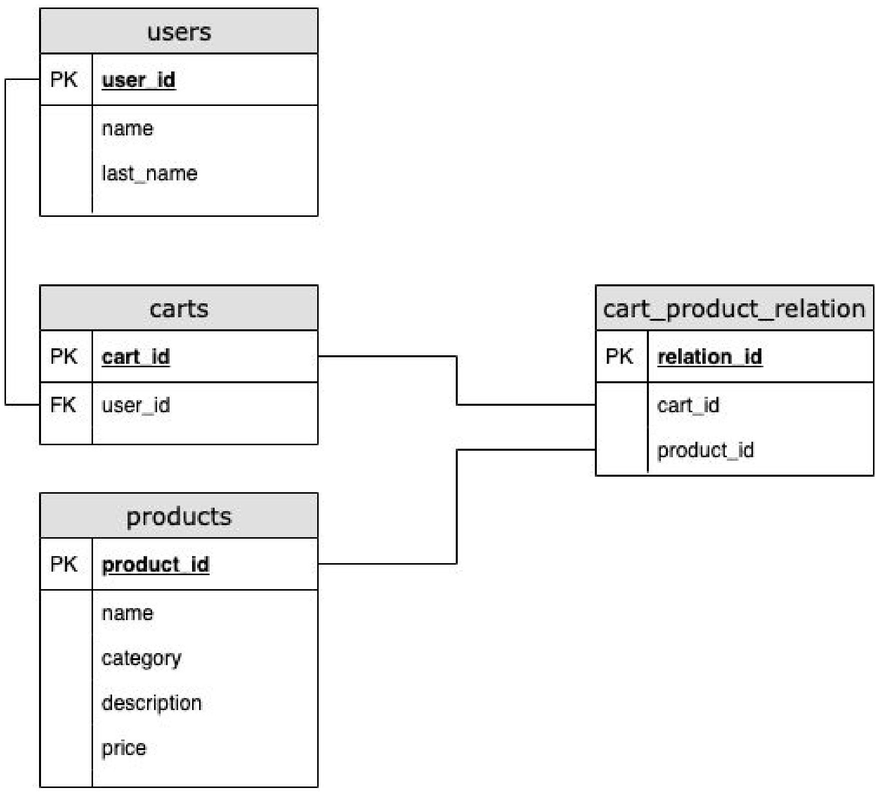


Figura 5: Model de dades fet servir per Marketplace API

Perquè les 3 aplicacions facin servir la mateixa base de dades s’ha decidit que la millor forma de fer-ho és crear-la a partir de docker. Docker és un projecte de codi obert que permet desplegar aplicacions dins de contenidors. El contenidor és l'entorn on viu i s'executa l'aplicació. La base de dades que s’ha fet servir estava dintre d'un contenidor de docker i les diferents aplicacions hi podien accedir per fer queries. La base dades és mysql 5.7. Per a facilitar l'ús de la base de dades s’ha fet servir l'eina docker-compose, que permet personalitzar la forma en la que es vol que els contenidors de docker es creïn. D'aquesta forma, només amb la comanda 'docker-compose up' es crea una nova base de dades exposada al port 4050 del sistema. En aquest punt les aplicacions estan llestes per connectar-se-hi. Després d'iniciar la base de dades amb docker-compose, s’han de crear les taules dintre de

la consola de mysql. L'arxiu *docker-compose* i les taules estan definides dintre del projecte Elixir Marketplace API. Els passos a seguir són els següents:

Es descarrega la imatge de mysql amb la comanda:

docker pull mysql:5.7

Es crea l'arxiu 'docker-compose' i es configura que es vol crear una instància de mysql que fa servir la imatge de *mysql:5.7*. Es defineix el nom de la base de dades, l'usuari, el password, el host i el port. Llavors ja es pot executar:

docker-compose up

Que arrenca la base de dades, en el cas d'aquest treball, en el port 4050. S'hi pot accedir amb la comanda:

mysql -h localhost -P 4050 --protocol=tcp -u root -pmarketplace

En la consola de mysql s'han de crear les taules de la base de dades. Per fer-ho s'executa:

```
CREATE TABLE products (
   product_id int NOT NULL AUTO_INCREMENT,
   name VARCHAR (36),
   category VARCHAR (36),
   description VARCHAR (36),
   price int,
   PRIMARY KEY (product_id)
);

CREATE TABLE users (
   user_id INT NOT NULL AUTO_INCREMENT,
   name VARCHAR (36),
   last_name VARCHAR (36),
   PRIMARY KEY (user_id)
);
```

```
CREATE TABLE carts (
    cart_id INT NOT NULL AUTO_INCREMENT,
    user_id INT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (user_id),
    PRIMARY KEY (cart_id)
);

CREATE TABLE cart_product_relation (
    relation_id INT NOT NULL AUTO_INCREMENT,
    cart_id INT NOT NULL,
    product_id INT NOT NULL,
    FOREIGN KEY (cart_id) REFERENCES carts (cart_id),
    FOREIGN KEY (product_id) REFERENCES products (product_id),
    PRIMARY KEY (relation_id)
);
```

Endpoints

L’API té els següents endpoints:

- **POST /user/create**

  Crea un nou usuari i l’afegeix a la base de dades. Els paràmetres d’entrada són el nom i cognom de l’usuari. La resposta és l’id d’usuari.

  ```
  body: {
          “name”: “quim”,
          “last_name”: “salvado”
  }

  response: {
          “user_id”: 1
  }
  ```

- **POST /user/cart**

  Retorna els productes que un usuari té a la seva cistella. El paràmetre

d’entrada és l’id de l’usuari i la resposta és un llistat de productes.

body: {
“user_id”: 1
}

response: [
{
"product_id": 1,
"name": "gambes de palamos",
"category": "alimentacio",
"description": "les millors gambes",
"price": 11
},
……..
]

- **POST /user/add_product**

  Afegeix un producte a una cistella. Els paràmetres d'entrada són l'id del usuari i l'id del producte. La resposta indica si el producte s'ha afegit correctament o no.

body: {
“user_id”: 1,
“product_id”: 1
}

response: {
“status”: “Cart updated”
}

- **POST /user/checkout**

  Calcula el valor total de la cistella d’un usuari i elimina els productes de la cistella. El paràmetre d’entrada és l’id de l'usuari i la resposta és el valor de la cistella.

body: {

“user_id”: 1

}

response: {

“total”: 100

}

- **GET /product/all**

  Retorna una llista amb tots els productes.

  response: [

  {

  "product_id": 1,

  "name": "gambes de palamos",

  "category": “alimentacio",

  "description": "les millors gambes",

  "price": 11

  },

  …….

  ]

- **POST /product/new**

  Afegeix un producte que els usuaris podran afegir a la cistella. Els paràmetres d’entrada són el nom, categoria, descripció i preu del producte. La resposta és el producte, si aquest s’ha creat correctament.

  body: {

  “name”: “gambes de palamos”,

  “category”: “alimentacio”,

  “description”: “les millors gambes”,

  “price”: 50

  }

response: {
"product": {
"product_id": 1,
"name": "gambes de palamos",
"category": "alimentacio",
"description": "les millors gambes",
"price": 50
}
}

- **POST /product/id**

  Retorna informació sobre un producte concret. El paràmetre d'entrada és l'id del producte i la resposta és la informació del producte.

  body: {
  "product_id": 1
  }

  response: {
  "product_id": 1,
  "name": "gambes de palamos",
  "category": "alimentacio",
  "description": "les millors gambes",
  "price": 50
  }

Els llenguatges amb els quals s'han creat aquesta aplicació són:

Python (https://github.com/icosi/Python-Marketplace-API)

Per a crear l'API en Python s'ha fet servir Django, un framework gratuït i open source que permet crear aplicacions web amb molta facilitat.

Els passos per a crear l'aplicació base són els següents:

Es crea el projecte base amb la següent comanda:

django-admin startproject Python_Marketplace_API

Es modifica la configuració del projecte perquè l'aplicació es conecti amb la base de dades que s'ha creat abans.

Creem els models de dades per poder interactuar amb les taules de la base de dades des del codi. Es crearan el models amb una comanda que escaneja la base de dades (les taules de la DB han d'estar creades), llegeix les taules que hi ha i crea els models en python a punt per fer-se servir:

python3 manage.py inspectdb > models.py

Llavors s'aplica la migració dels models:

python3 manage.py migrate

En aquest punt s'ha deixat l'aplicació connectada a la base de dades i ja es pot crear la lògica de l'API.

Java (https://github.com/icosi/Java-Marketplace-API)
Per a crear l'API en Java s'ha fet servir Spring, un framework gratuït i open source que permet crear aplicacion web amb molta facilitat. L'aplicació s'ha creat en Eclipse

Per a crear una aplicació Spring es visita la pàgina web: https://start.spring.io/ on es pot crear un projecte base a mida.

Es modifica la configuració del projecte per a que es connecti a la base de dades que s'ha creat i es creen els models de dades en java seguint la documentació de Spring. En aquest punt ja es pot començar a escriure la lògica de l'aplicació.

Elixir (https://github.com/icosi/Elixir-Marketplace-API)
Per a crear l'API en Elixir no s'ha fet servir cap framework. S'ha fet l'API creant un projecte des de zero fent servir les eines que el llenguatge ofereix.

Per a crear el projecte base s'executa:

mix new marketplace_api --sup

L’opció --sup indica que es vol que es generi una aplicació amb un arbre de supervisió.

S’afegeixen les dependències necessàries per a connectar-se a bases de dades i poder acceptar peticions http.

Es modifica la configuració del projecte per a connectar-se a la base de dades i es creen els models de dades en elixir. En aquest punt ja es pot crear la lògica de l’aplicació.

2.3.3 ENTORNS

S’han fet servir 2 entorns on executar les aplicacions. El primer és localhost, és a dir, el mateix ordinador on s’han desenvolupat les aplicacions. Les característiques de l’ordinador són:

Processador: 1,4 GHz Intel Core i5 de 4 nuclis
Memoria: 8 GB 2133 MHz LPDDR3

L'entorn de localhost s’ha fet servir bàsicament per comprovar que les aplicacions funcionen correctement i el test de càrrega es pot realitzar. Tot i que a l'executar les aplicacions en localhost ens donarà uns resultats que no són realistes a l'hora de comparar l'eficiència dels llenguatges de programació pel fet de executar-se en localhost.

El segon entorn és un servidor de Digitalocean, un proveïdor d’infraestructura en el núvol. El servidor que s’ha fet servir té les següents característiques:

Memory	vCPU’s	Transfer	SSD Disk	Price
1GB	1vCPU	1TB	25GB	$5/month

El més important de la informació del servidor de Digitalocean és el seu preu, ja que ens permet quantificar en dòlars el rendiment de les aplicacions.

Es pot connectar en aquest servidor mitjançant ssh, que permet obrir una consola dintre de la màquina.

2.3.4 TESTS DE CÀRREGA

Realitzar un test de càrrega és el procés de crear demanda en un sistema informàtic i mesurar la seva resposta.

L'eina que s'ha fet servir per a fer el test de càrrega s'anomena JMeter, una aplicació open source dissenyada per a analitzar i mesurar el rendiment de serveis com aplicacions web.

El que fa JMeter és generar peticions HTTP dirigides a un servidor on hi haurà l'aplicació executant-se. Per fer-ho, simula usuaris que criden als endpoints de l'aplicació. Per a poder crear múltiples usuaris i fer nombroses crides http concurrents fa ús de multi-threading, sempre que la maquina on s'executi JMeter ho permeti.

3. RESULTATS

3.1 DUR A TERME ELS TESTS DE CÀRREGA

Per a poder iniciar un nou load test s'ha de crear una configuració per a definir com volem realitzar el test de càrrega. Al conjunt de tota la configuració s'anomena Test Plan. La configuració en separa en dues parts: els usuaris i les crides http. Els paràmetres més importants a definir per als usuaris són el nombre d'usuaris que es vol simular i la quantitat de peticions que farà cada usuari. Per a la configuració de les crides http, tot dependrà de l'aplicació que s'estigui testejant, però bàsicament s'ha de definir com vols fer les crides http als endpoints de l'aplicació.

JMeter té diferents formes per visualitzar els resultats del test anomenades Listeners. En els tests realitzats es fan servir 2.

View Results in Table, que mostra el resultat de cada petició http de forma individual. De View Results in Table obtenim la següent informació:

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency
4	22:08:23.192	Thread Group 1-4	HTTP Request	60		173	120	60
5	22:08:23.192	Thread Group 1-7	HTTP Request	61		173	120	60
7	22:08:23.192	Thread Group 1-2	HTTP Request	60		173	120	60
12	22:08:23.192	Thread Group 1-1	HTTP Request	60		173	120	60
13	22:08:23.192	Thread Group 1-3	HTTP Request	60		173	120	60
6	22:08:23.200	Thread Group 1-8	HTTP Request	52		173	120	52
8	22:08:23.209	Thread Group 1-9	HTTP Request	44		173	120	43
9	22:08:23.219	Thread Group 1-10	HTTP Request	33		173	120	33
10	22:08:23.229	Thread Group 1-11	HTTP Request	23		173	120	23
2	22:08:23.239	Thread Group 1-12	HTTP Request	13		173	120	13
11	22:08:23.249	Thread Group 1-13	HTTP Request	3		173	120	3

Figura 6: Resultat del Listener View Results in Table

- **Sample:** nombre de la petició
- **Start time:** moment d'inici de la petició
- **Thread Name:** thread a què pertany la petició
- **Label:** etiqueta de la petició

- **Sample Time (ms):** temps que la petició ha tardat a rebre una resposta
- **Status:** status de la resposta
- **Bytes:** bytes de la resposta
- **Sent Bytes:** bytes de la petició
- **Latency:** temps que la petició ha tardat fins que ha començat a rebre una resposta

Summary Report mostra un sumari de totes les peticions que s'han fet per cada endpoint diferent de la configuració. De Summary Report obtenim la següent informació:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
GET - Time	115048	87	0	1508	133.49	0.78%	9813.0/sec	2626.46	1530.85	274.1
TOTAL	115048	87	0	1508	133.49	0.78%	9813.0/sec	2626.46	1530.85	274.1

Figura 7: Resultat del Listener Summary Report

- **Label**: nom de la petició
- **# Samples**: nombre total de peticions
- **Average**: temps mitjà transcorregut en mil·lisegons
- **Min**: temps mínim transcorregut en mil·lisegons
- **Max**: temps màxim transcorregut en mil·lisegons
- **Standard deviation**: quantifica la variació del conjunt de dades
- **Errors %**: Percentatge d'errors
- **Throughput**: nombre de peticions per segon
- **Received KB/sec:** informació que el sistema rep en kilobytes per segon
- **Sent KB/sec:** informació que el sistema envia en kilobytes per segon
- **Average Bytes:** pes mitjà de les peticions en bytes

JMeter guardarà en arxius l'informació que aquests dos Listeners generin perquè puguem tractar les dades amb més facilitat i podem crear gràfics.

3.2 TIME API EN LOCALHOST

En aquest apartat es posa a prova l'API del temps executant-se en localhost.

Per a testejar l'API de temps en localhost s'ha fet servir la següent configuració de jMeter:

- Configuració d'usuaris:
  - Usuaris: 100
  - Peticions per usuari: 100
  - Peticions totals 10.000
- Configuració de crides http:
  - Endpoint 1 (l'únic de l'aplicació):
    - Protocol: http
    - IP: localhost
    - Port:
      - Elixir: 4000
      - Java: 4567
      - Python: 8000
    - Mètode: GET
    - Path: /time

L'objectiu és realitzar a les 3 aplicacions un load test fent servir la mateixa configuració. D'aquesta forma les aplicacions hauran estat testejades amb exactament la mateixa quantitat de càrrega.

En executar l'aplicació i realitzar els tests de càrrega s'obté un fitxer .csv (valor separat per coma) amb tota la informació específica de cada petició realitzada en el test:

	timeStamp	elapsed	label	responseCode	responseMessage
0	1587919249474	108	HTTP Request	200	OK
1	1587919249481	111	HTTP Request	200	OK
2	1587919249506	93	HTTP Request	200	OK
3	1587919249493	112	HTTP Request	200	OK
4	1587919249582	43	HTTP Request	200	OK
...	...	...	...	...	...
9995	1587919278264	1581	HTTP Request	200	OK
9996	1587919279845	54	HTTP Request	200	OK
9997	1587919279899	64	HTTP Request	200	OK
9998	1587919279963	57	HTTP Request	200	OK
9999	1587919280020	47	HTTP Request	200	OK

10000 rows × 17 columns

Figura 8: Resultats del test realitzat a Time API

En aquest punt, un cop realitzats cadascun dels tests de càrrega s'ha fet servir Jupyter per a tractar els resultats. Jupyter és una aplicació web de codi obert que permet crear i compartir documents que contenen codi que s'executa. És fet servir per a tractar i visualitzar dades, big data, modelatge estadístic i molt més.

En el nostre cas es fa servir Jupyter per a visualitzar la informació recaptada pel test de càrrega. S'ha generat un arxiu de 10.000 línies amb informació de cada crida http per cada test, però la informació que ens interessa és el temps que ha trigat cada crida en ser executada.

El resum del test de càrrega de l’API de temps executada en localhost és:

Llenguatge	Average [ms]	Min [ms]	Max [ms]	Throughput	Temps
Java	0	0	4	9970.1/sec	1 sec
Elixir	6	0	83	6821.3/sec	1 sec
Python	105	10	265	514.9/sec	19 sec

- **Llenguatge**: llenguatge de programació fet servir per al test
- **Average**: temps mitjà de les peticions en tot el test
- **Min**: temps mínim del conjunt de peticions del test
- **Max**: temps màxim del conjunt de peticions del test
- **Throughput**: nombre de peticions per segon que ha absorbit l’aplicació
- **Temps**: temps que ha tardat el test a executar-se per complet

Anem a veure amb més profunditat els resultats de cada un dels tests. Es mostren els boxplots en funció del temps que ha tardat cada petició a executar-se per complet en mil·lisegons.

3.2.1 RESULTATS JAVA

Boxplot del resultat del test a l'API feta en Java:

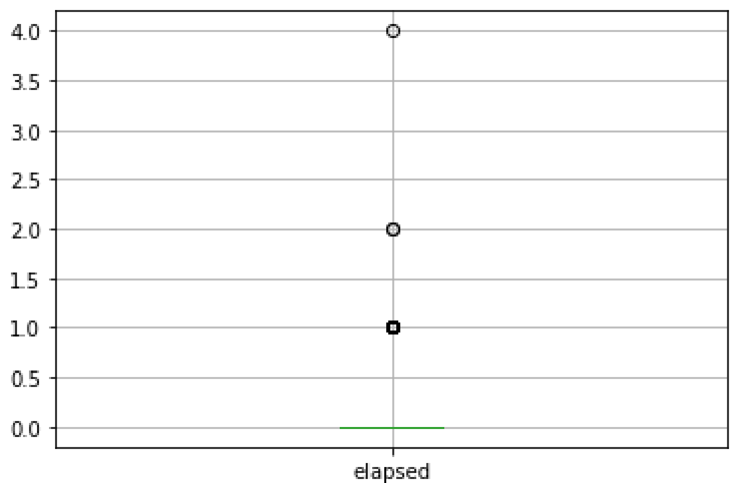


Figura 9: Boxplot dels resultats obtinguts de Time API en Java

Es veu com Java ha aconseguit un rendiment perfecte. El temps mitjà que ha tardat a respondre cada petició és de 0 segons, amb alguna que un altra petició tardant fins a un màxim de 4 mil·lisegons.

3.2.2 RESULTATS PYTHON

Boxplot del resultat del test a l'API feta en Python:

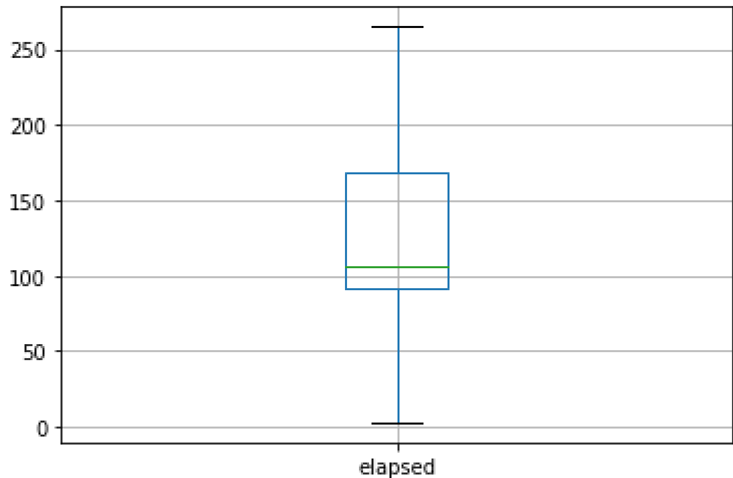


Figura 10: Boxplot dels resultats obtinguts de Time API en Python

Python ha tret els pitjors resultats de les 3 aplicacions. El seu temps mitjà de resposta ha estat 105 mil·lisegons, nombre molt elevat tenint en compte que s'està executant en localhost. Això vol dir que el llenguatge no gestiona molt bé un gran nombre de peticions a la vegada.

3.2.3 RESULTATS ELIXIR

Boxplot del resultat del test a l'API feta en Elixir:

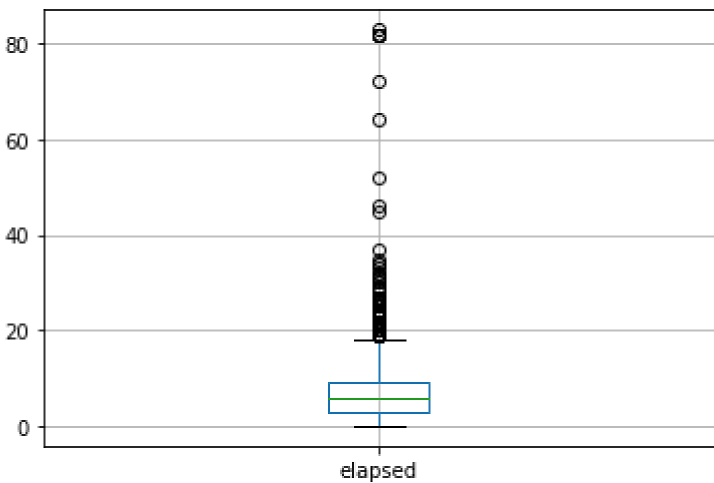


Figura 11: Boxplot dels resultats obtinguts de Time API en Elixir

Elixir ha aconseguit un bon resultat, una mitjana de 6 mil·lisegons per petició. Però no ha estat tan constant com Java, per això podem veure com una part de les peticions va des dels 18 mil·lisegons als 83 mil·lisegons.

3.2.4 COMENTARIS DELS RESULTATS

Comparant el rendiment de les diferents aplicacions podem veure que Java és el llenguatge que millor se n'ha sortit amb un temps de resposta increïblement baix

Es pot veure que els resultats no reflecteixen de forma realista com de bé rendeixen els llenguatges de programació. L'entorn on s'han executat les aplicacions ha estat la màquina en local, degut a això les peticions mai han sortit fora de la màquina per el que el temps de resposta sempre és més baix del que realment seria en un entorn de producció real. Pero el fet de haver executat tots els tests de forma correcte i recopilar els resultats ja fan d'aquesta prova un èxit.

3.3 TIME API EN SERVIDOR

En aquest apartat es posa a prova l’API del temps executant-se en un servidor real.

Per a testejar l’API de temps en el servidor es fa servir la mateixa configuració que s’ha fet servir en localhost, només canviant l’IP:

- Configuració d’usuaris:
    - Usuaris: 100
    - Peticions per usuari: 100
    - Peticions totals 10.000
- Configuració de crides http:
    - Endpoint 1 (l'únic de l'aplicació):
        - Protocol: http
        - IP: 134.209.254.62
        - Port:
            - Elixir: 4000
            - Java: 4567
            - Python: 8000
        - Mètode: GET
        - Path: /time

Com a resultat d’aquests tests també s’obtè uns arxius .csv amb tots els resultats de cadascuna de les peticions realitzades.

Els resultats obtinguts del load test són els següents:

Llenguatge	Average	Min	Max	Throughput	Temps
Java	363	45	12661	250/sec	40 sec
Elixir	169	42	5194	333.33/sec	30 sec
Python	873	22	4200	62.5/sec	160 sec

Anem a veure amb més profunditat els resultats de cada un dels tests.

3.3.1 RESULTATS JAVA

Visualitzem el boxplot de les dades obtingudes a partir de totes les peticions http que s'han fet:

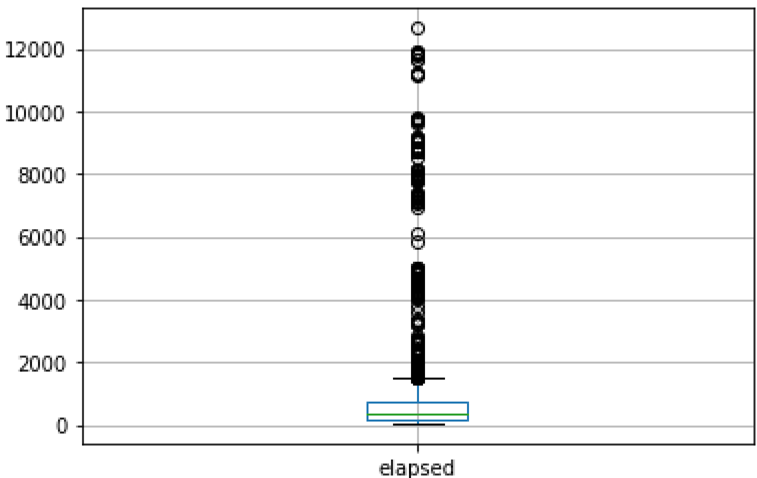


Figura 12: Boxplot dels resultats obtinguts de Time API en Java

El temps mitjà és de 363 mil·lisegons.

Es pot veure que no es visualitza la informació del tot bé, pel que anem a generar l'histograma de les dades. Al visualitzar l'histograma de totes les peticions fetes a l'aplicació de Java, es veu que la gran majoria de peticions es troben per sota dels 2.000 mil·lisegons de resposta.

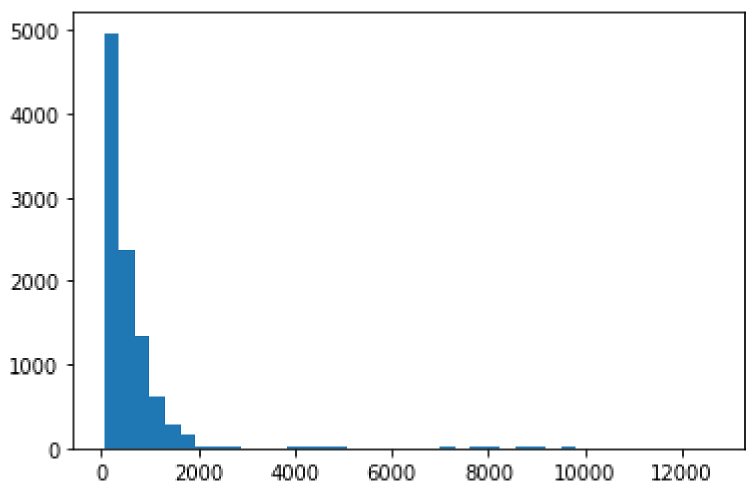


Figura 13: Histograma dels resultats obtinguts de Time API en Java

De les 10.000 mostres diferents, 9.773 es troben per sota dels 2.000 mil·lisegons de resposta. Per culpa de això él que es farà serà eliminar les mostres superiors als 2.000 mil·lisegons per a mostrar un boxplot que es visualitza millor:

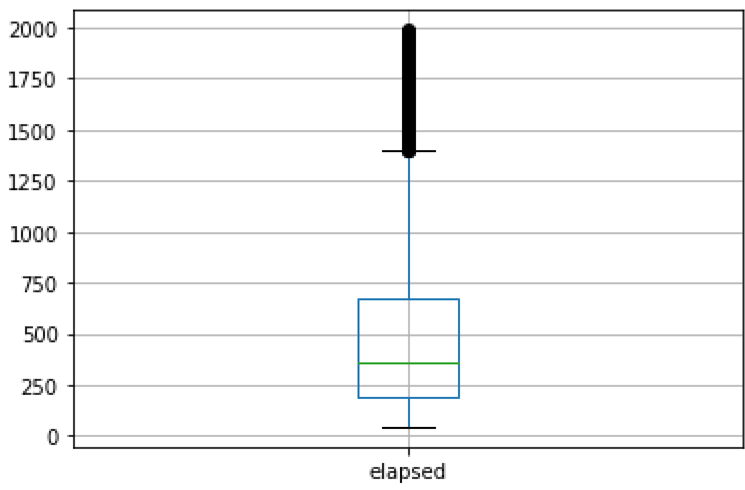


Figura 14: Boxplot dels resultats filtrats obtinguts de Time API en Java

El temps mitjà de les crides és 355 mil·lisegons.

No s’ha guanyat molt, però d’aquesta forma es pot veure els límits del boxplot.

3.3.2 RESULTATS PYTHON

Les dades del load test de Python són una mica estranyes. Per començar, no hi ha 10.000 peticions, sinó que n'hi ha 14.735. El motiu d'aquest augment és perquè hi ha peticions que han generat el codi d'error 301 Moved Permanently. Segons la Wikipedia, aquest codi d'error indica que el host ha estat capaç de comunicar-se amb el servidor però que el recurs sol·licitat ha estat mogut a un altra direcció permanentment. Però tot i aquesta anomalia, s’ha pogut filtrar les dades correctes de les incorrectes a Jupyter i es pot continuar amb l'anàlisi de resultats. El boxplot de les peticions ens mostra el següent:

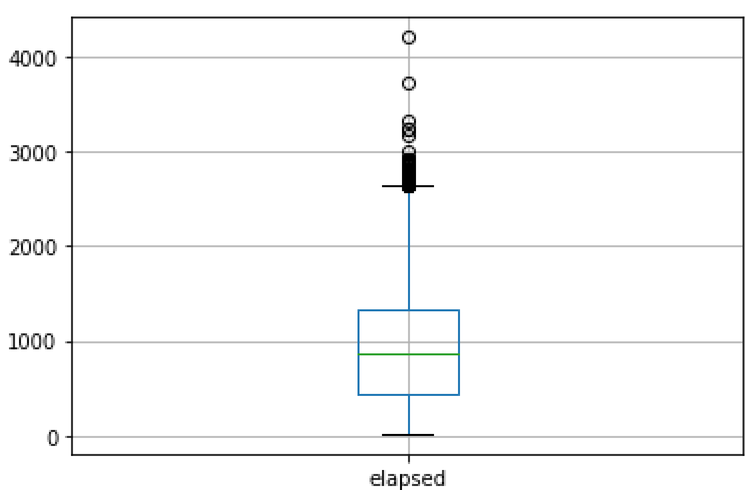


Figura 15: Boxplot dels resultats obtinguts de Time API en Python

El temps mitjà és de 873 mil·lisegons.

Visualitzem l’histograma:

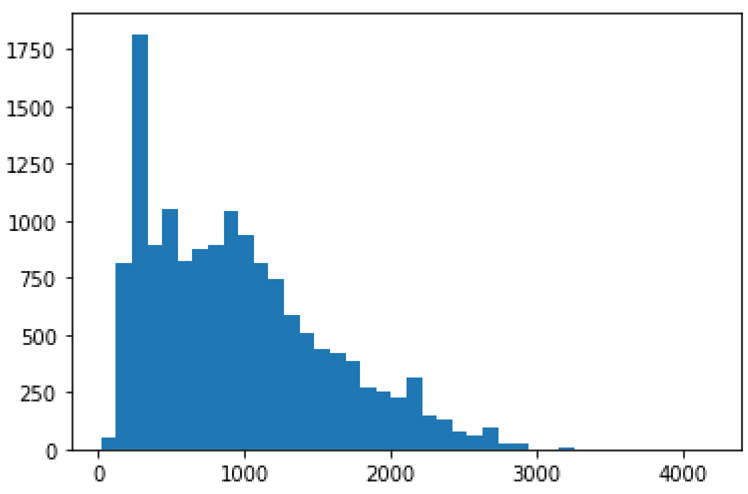


Figura 16: Histograma dels resultats obtinguts de Time API en Python

Podem veure que els valors estan més distribuïts que en l'histograma de Java. Això vol dir que Python no es capaç de gestionar de forma eficient una gran quantitat de peticions.

3.3.3 RESULTATS ELIXIR

Boxplot de les dades:

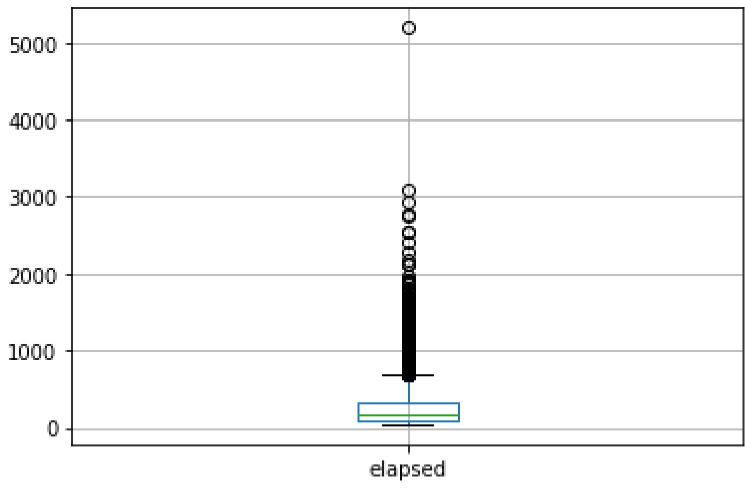


Figura 17: Boxplot dels resultats obtinguts de Time API en Elixir

Les dades son dificils de visualitzar de forma que s'entengui. Visualitzem l'histograma:

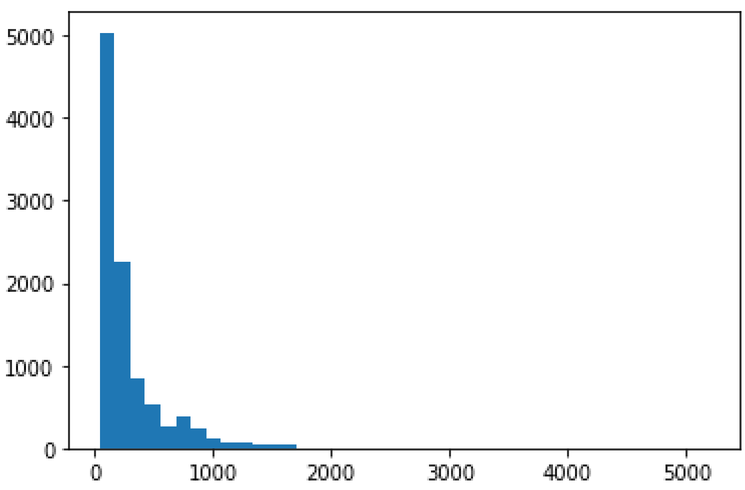


Figura 18: Histograma dels resultats obtinguts de Time API en Elixir

Veiem que només hi ha un total de 13 mostres que estiguin per sobre de 2.000 mil·lisegons. Les retirem i tornem a calcular el boxplot:

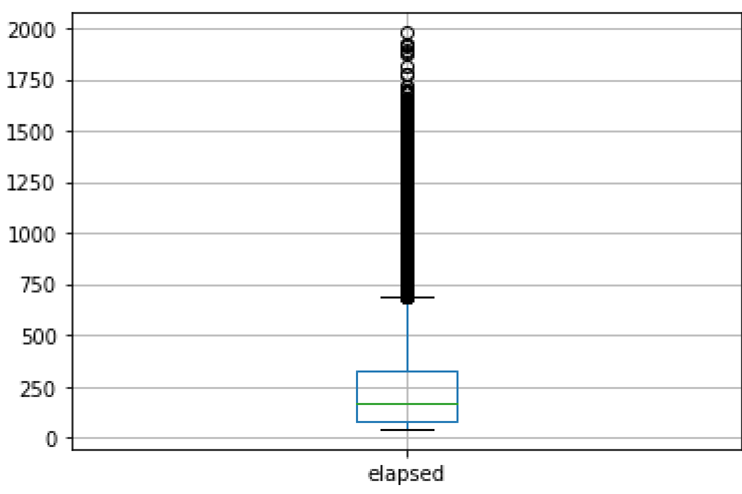


Figura 19: Boxplot dels resultats obtinguts de Time API en Elixir

La mitjana son de 169 mil·lisegons. A més, es pot veure que les peticions, en general, no passen dels 350 mil·lisegons.

3.3.4 COMENTARIS DELS RESULTATS

Es pot veure com els resultats que s'han obtingut en l'API de temps executant-se en el servidor tenen uns temps de resposta més realistes. Els resultats que s'han obtingut ja deixen veure que Elixir té més bon rendiment que Java i Python, però tot hi això, tota la lògica d'aquesta API es basa en dir la hora i només té un endpoint. No es pot decidir quin llenguatge és millor basant-se únicament en els resultats anteriors.

3.4 MARKETPLACE API EN SERVIDOR

Aquest és el test de càrrega més important, ja que executa una aplicació complexa amb múltiples endpoints i una lògica darrere que realitza lectures i escriptures a una base de dades.

Degut al nombre d'endpoints de l'API la configuració de jMeter ha estat la següent:

- Configuració d'usuaris:
    - Usuaris: 20
    - Peticions per usuari: 50
    - Peticions totals 1.000 * nº d'endpoints = 7.000
- Configuració de crides http:
    - Per a tots els endpoints:
        - Protocol: http
        - IP: 134.209.254.62
        - Port:
            - Elixir: 4000
            - Java: 4567
            - Python: 8000
    - Endpoint 1:

        - Mètode: POST
        - Path: /user/create
    - Endpoint 2:
        - Mètode: POST
        - Path: /user/cart
    - Endpoint 3:
        - Mètode: POST
        - Path: /user/add_product
    - Endpoint 4:
        - Mètode: POST
        - Path: /user/checkout
    - Endpoint 5:
        - Mètode: POST
        - Path: /product/new
    - Endpoint 6:
        - Mètode: POST
        - Path: /product/id
    - Endpoint 7:
        - Mètode: GET
        - Path: /product/all

A l'executar l'aplicació i realitzar els tests de càrrega s'obté un fitxer .csv per a cada endpoint amb tota la informació específica de cada petició realitzada en el test.

Per a veure els resultats de tots els endpoints d'una forma clara, es mostra els resultats dels tests en forma de boxplot.

3.4.1 RESULTATS JAVA

Al realitzar el test de càrrega a java l'aplicació s'ha vist sobrecarregada, degut a això s'ha modificat la configuració del test de càrrega per a simular 20 usuaris que fan 50 peticions cadascun. El temps que ha trigat en donar resposta a totes les 7.000

peticions ha estat de 6:14 minuts, això vol dir que ha suportat una càrrega mitja de 18,7 peticions per segon.

En el següent diagrama es pot observar com es distribueix el temps de resposta de les peticions (en mil·lisegons) per cada endpoint:

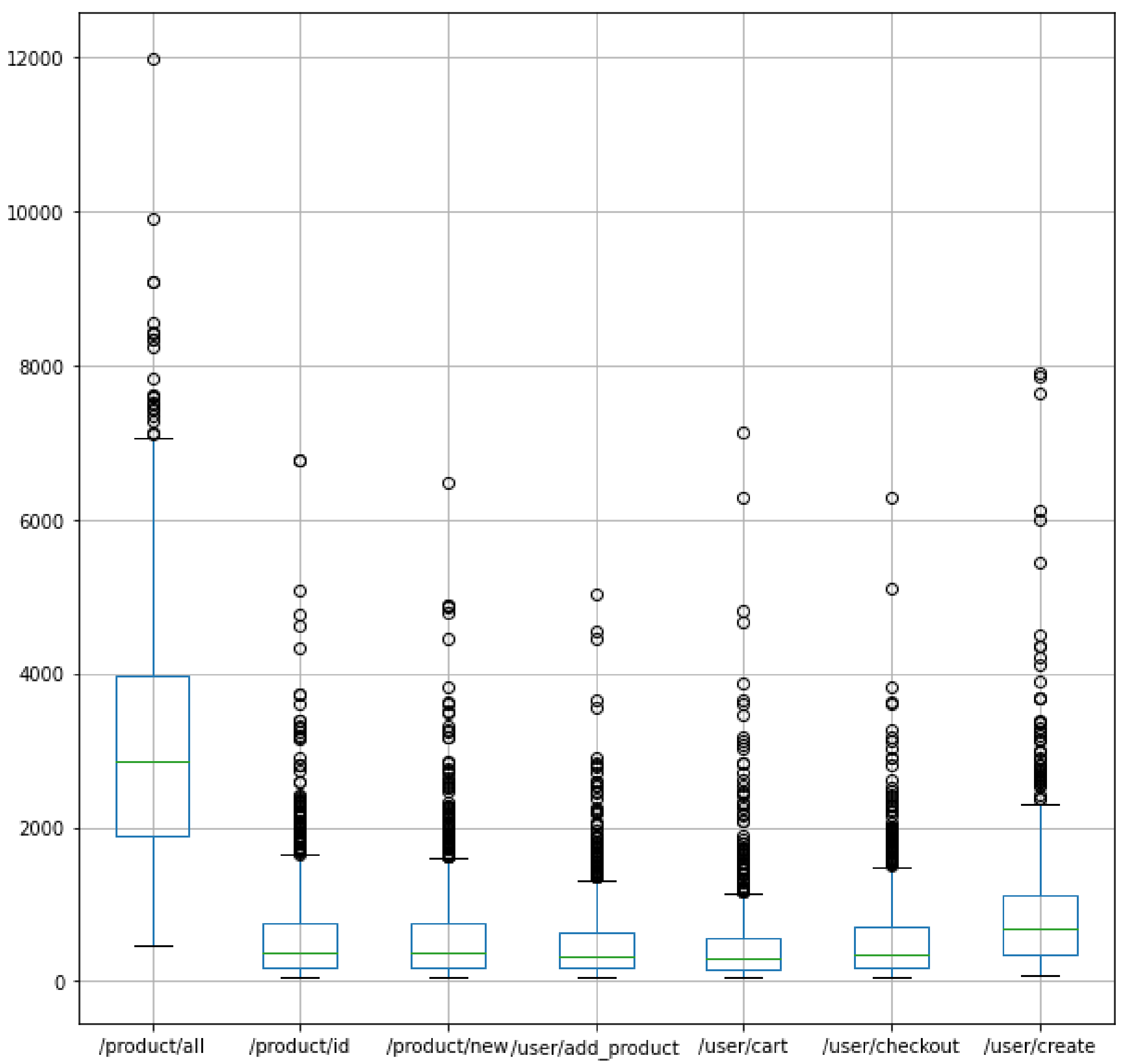


Figura 20: Boxplot dels resultats obtinguts de Marketplace API en Java

El temps mitjà dels endpoints de /product és:

	product/all	product/id	product/new
Temps mitjà [ms]	2859	384	367,5

El temps mitjà dels endpoints de /user és:

	user/create	user/cart	user/add_product	user/checkout
Temps mitjà [ms]	680	308,5	325,5	359,5

El temps mitjà que l'aplicació tarda a respondre una petició entre tots els endpoints és de 754,8 mil·lisegons.

3.4.2 RESULTATS PYTHON

El temps que ha trigat a donar resposta a totes les 7.000 peticions ha estat de 5:40 minuts, això vol dir que ha suportat una càrrega mitja de 20,5 peticions per segon.

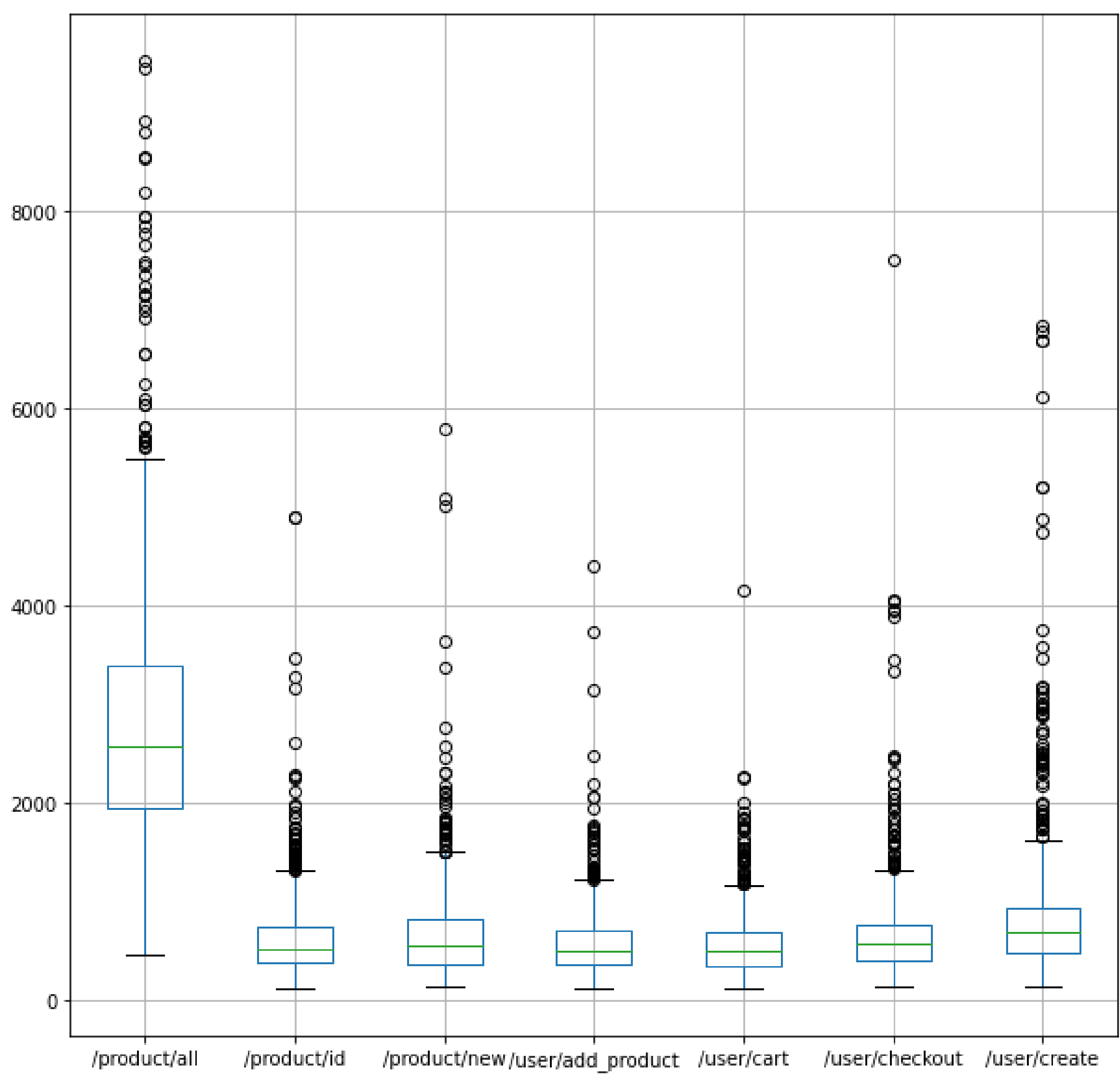


Figura 21: Boxplot dels resultats obtinguts de Marketplace API en Python

El temps mitjà dels endpoints de /product és:

	product/all	product/id	product/new
Temps mitjà [ms]	2562	521	552,5

El temps mitjà dels endpoints de /user és:

	user/create	user/cart	user/add_product	user/checkout
Temps mitjà [ms]	680,5	485,5	501	567,5

El temps mitjà que l’aplicació tarda en respondre una petició entre tots els endpoints és de 838,5 mil·lisegons.

3.4.3 RESULTATS ELIXIR

El temps que ha trigat a donar resposta a totes les 7.000 peticions ha estat de 2:20 minuts, això vol dir que ha suportat una càrrega mitja de 50 peticions per segon.

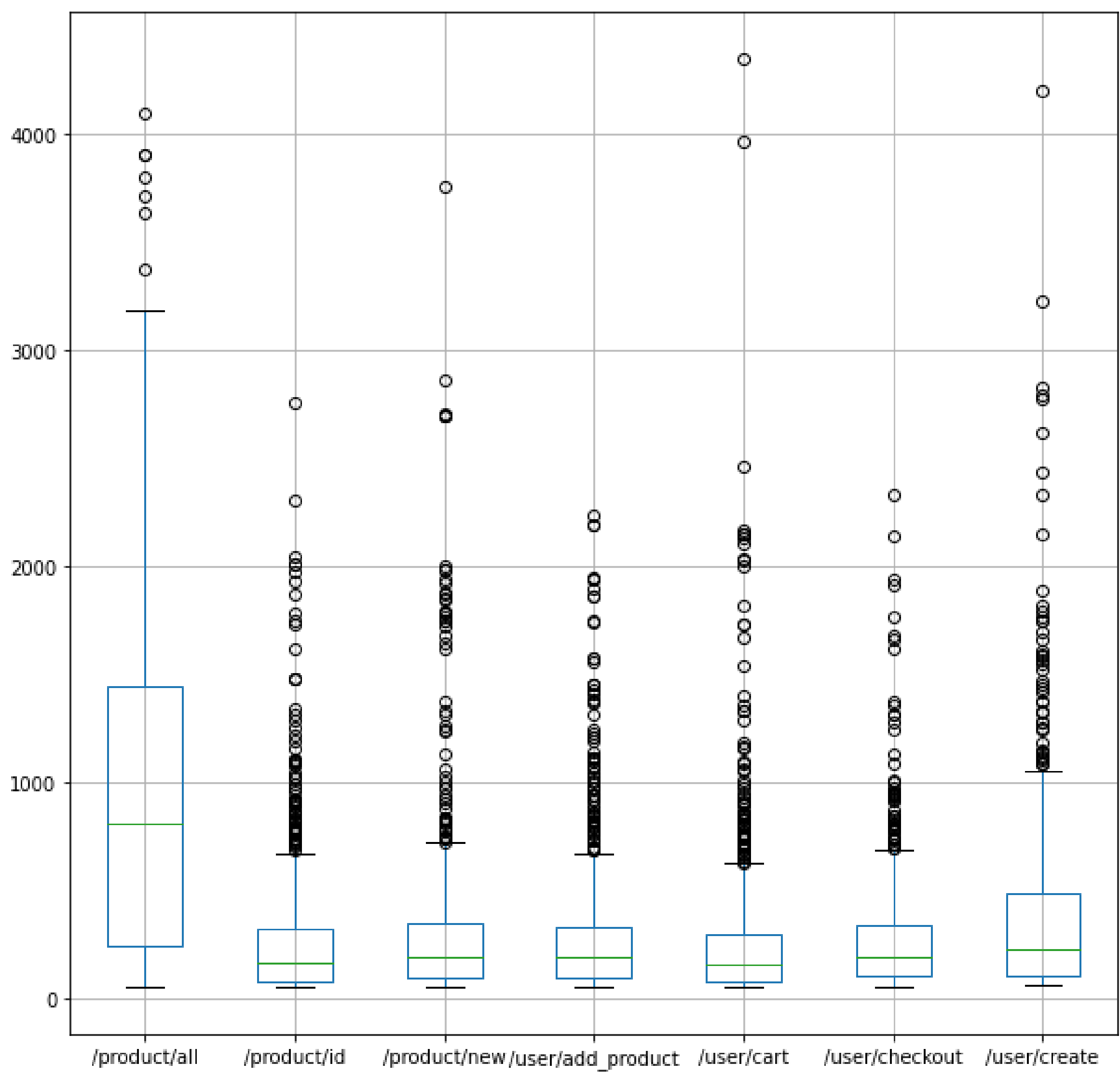


Figura 22: Boxplot dels resultats obtinguts de Marketplace API en Elixir

El temps mitjà dels endpoints de /product és:

	product/all	product/id	product/new
Temps mitjà [ms]	804,5	166	186

El temps mitjà dels endpoints de /user és:

	user/create	user/cart	user/add_product	user/checkout
Temps mitjà [ms]	229	158	192,5	194

El temps mitjà que l'aplicació tarda a respondre una petició entre tots els endpoints és de 275,7 mil·lisegons.

3.4.4 COMENTARIS DELS RESULTATS

Podem observar com mentre que tots els endpoints tenen un temps mitjà similar dintre del mateix llenguatge, /product/all i /user/create tarden més que la resta. Això és degut al nombre de queries a la base de dades que es realitzen. /product/all ha d'agafar tots els productes existents en la base de dades, cosa que pot ser costos quan hi ha un gran volum d'elements. Per l'altre costat, /user/create no només crea l'usuari, sinó que també crea la cistella i fa la relació entre usuari i cistella. Aquest és el motiu per el qual triga una mica més que la resta d'endpoints.

Anem a comparar els resultats de cada endpoint en funció del llenguatge de programació fet servir:

- Endpoint 1: /user/create

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	680	72	11.976
Elixir	229	55	4199
Python	680,5	123	6824

- Endpoint 2: /user/cart

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	308,5	54	7131
Elixir	158	50	4351
Python	485,5	109	4147

- Endpoint 3: /user/add_product

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	325,5	59	5039
Elixir	192,5	52	2233
Python	501	111	4405

- Endpoint 4: /user/checkout

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	359,5	64	6293
Elixir	194	54	2334
Python	567,5	136	7498

- Endpoint 5: /product/new

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	367,5	57	6465
Elixir	186	52	3757
Python	552,5	134	5794

- Endpoint 6: /product/id

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	384	59	6771
Elixir	166	49	2755
Python	521	116	4883

- Endpoint 7: /product/all

Llenguatge	Average [ms]	Min [ms]	Max [ms]
Java	2859	478	11974
Elixir	804,5	54	4096
Python	2562	453	9518

Es pot veure com Elixir és el clar guanyador en rapidesa en tots els endpoints

Podem crear una taula agrupant la informació que és més important per arribar a una conclusió:

Llenguatge	Temps de test	Peticions/segon	Temps mitjà per petició
Java	6:14 minuts	18,7	754,8 milisegons
Elixir	2:20 minuts	50	275,7 milisegons
Python	5:40 minuts	20,5	838,5 milisegons

El test consta de 7.000 peticions distribuïdes de forma que es realitzen 1.000 peticions per endpoint. El servidor on les aplicacions s'han executat consta de 1 CPU, 1 GB de RAM i costa 5 dolars al mes.

En el test realitzat, segons la quantitat de peticions per segon que l'aplicació és capaç de respondre, Elixir demostra ser 2,6 cops més eficient que Java i 2,4 cops més eficient que Python. Segons el temps mitjà que tarda en respondre una petició, Elixir és 2,7 cops més ràpid que Java i 3 cops més ràpid que Python.

Això vol dir que Java i Python necessiten 3 servidors com el que hem fet servir per obtenir el mateix rendiment que Elixir li treu a 1 servidor.

Veiem com hi ha un resultat que a primera vista no sembla molt intuïtiu. Java ha tardat més temps a realitzar el test que Python, però el temps mitjà per petició és més baix en Java. Això passa perquè el temps mitjà per petició és des que s'envia fins que es respon la petició, mentre que el temps total de test té en compte des que es comença a enviar la primera petició fins que es rep la resposta de l'última. Cada llenguatge té la seva forma interna de gestionar les peticions que hi arriben. Java sembla que pot executar les peticions de forma individual més ràpid que Python, però Python pot gestionar més peticions a la vegada.

3.5 RESULTATS OBTINGUTS I OBJECTIUS

L'objectiu del treball ha estat comparar el rendiment d'Elixir amb altres llenguatges de programació. A la vegada s'ha explorat les característiques internes d'Elixir per entendre els motius que el fan tant potent. Amb les dades obtingudes a través dels tests s'ha pogut arribar a una conclusió al respecte. En l'àmbit de la programació web, Elixir és més eficient que Java i Python i s'ha pogut demostrar amb xifres obtingudes a través de proves reals.

4. CONCLUSIONS

Aquest estudi no es centra a obtenir la millor programació possible, per això no s'ha aprofundit molt en com s'ha fet la lògica de les aplicacions creades. Aquest estudi s'ha centrat en el rendiment d'aplicacions escrites en diferents llenguatges de programació executant-se en un entorn real de producció. Per tal de realitzar l'estudi s’han fet servir les següents tecnologies:

- Visual studio code
- Eclipse
- Java
- Python
- Elixir
- Docker
- Mysql
- jMeter
- Postman
- Git
- Ssh
- Digitalocean
- Jupyter

Aquestes tecnologies s’han fet servir per a crear, guardar, executar, desplegar, provar i testejar aplicacions, a més de per visualitzar els resultats obtinguts dels tests realitzats.

S’ha pogut comprovar com Elixir és més eficient que Python i Java pel fet que Elixir és un llenguatge paral·lel, concurrent i funcional, que fa servir el model d’actors per a comunicar-se.

Realitzar aquest projecte ha servit per veure tot el cicle de producció del software. Ha estat a petita escala, pero s’ha realitzat la feina d’un desenvolupador, per a desenvolupar les aplicacions; d’un QA (quality assurance), per a testejar el codi

desenvolupat; de sysops, per tal de poder fer ús d'un entorn on executar les aplicacions; i, molt per sobre, data analyst, per transformar els resultats dels tests en informació útil.

Aquest tipus de estudi és molt rellevant per a qualsevol CTO (Chief Technology Officer) que ha d'escollir el llenguatge de programació en el que desenvolupar un projecte. Lo bo que tenen Java i Python és que és fàcil trobar desenvolupadors, i això pot semblar bo a primera vista. Pero si es realitza tot un projecte i un cop posat en un entorn de producció es veu que el cost de manteniment és molt elevat, el llenguatge acaba siguent un coll de botella. Ser conscient de quin és el millor llenguatge de programació per assolir els objectius de l'empresa forma part de les aptituds que un bon CTO ha de tenir.

5. BIBLIOGRAFIA

Bibliografia citada:

- [1] https://en.wikipedia.org/wiki/Higher-order_function
- [2] https://en.wikipedia.org/wiki/Lambda_calculus
- [3] https://en.wikipedia.org/wiki/Imperative_programming
- [4] https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)
- [5] https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
- [6] https://searchcio.techtarget.com/definition/vertical-scalability
- [7] https://searchcio.techtarget.com/definition/horizontal-scalability
- [8] https://rapidapi.com/blog/api-glossary/endpoint/

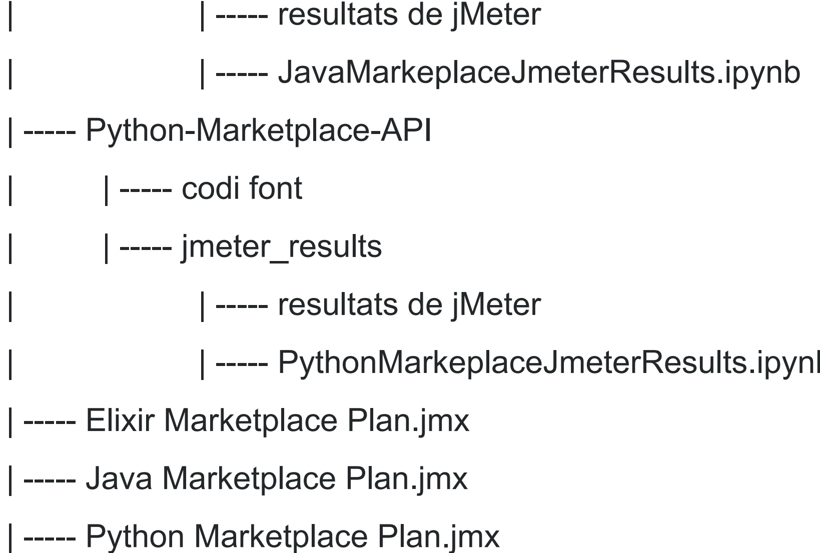
Bibliografia consultada:

- https://elixir-lang.org/
- https://en.wikipedia.org/wiki/Elixir_(programming_language)
- https://en.wikipedia.org/wiki/Functional_programming
- https://en.wikipedia.org/wiki/Concurrent_computing
- https://en.wikipedia.org/wiki/Erlang_(programming_language)
- https://en.wikipedia.org/wiki/Actor_model
- https://medium.com/@nguyenthanhquang.cse/introduction-to-actor-model-2c82d25d1b83
- https://jmeter.apache.org/usermanual/index.html
- https://docs.djangoproject.com/en/3.0/
- https://docs.spring.io/spring/docs/current/spring-framework-reference/
- https://spark.apache.org/docs/2.4.6/

6. APÈNDIX

Estructura de la carpeta de codi font:

```
Codi font
      | ----- TimeAPI
      |       | ----- Elixir-Time-API
      |       |       | ----- codi font
      |       |       | ----- jmeter_results
      |       |               | ----- resultats de jMeter
      |       | ----- Java-Time-API
      |       |       | ----- codi font
      |       |       | ----- jmeter_results
      |       |               | ----- resultats de jMeter
      |       | ----- Python-Time-API
      |       |       | ----- codi font
      |       |       | ----- jmeter_results
      |       |               | ----- resultats de jMeter
      |       | ----- Elixir Time Plan.jmx
      |       | ----- Java Time Plan.jmx
      |       | ----- Python Time Plan.jmx
      |       | ----- TFG.ipynb
      |
      | ----- MarketplaceAPI
              | ----- Elixir-Marketplace-API
              |       | ----- codi font
              |       | ----- jmeter_results
              |               | ----- resultats de jMeter
              |               | ----- ElixirMarkeplaceJmeterResults.ipynb
              | ----- Java-Marketplace-API
              |       | ----- codi font
              |       | ----- jmeter_results
```



```
|                    | ----- resultats de jMeter
|                    | ----- JavaMarkeplaceJmeterResults.ipynb
| ----- Python-Marketplace-API
|          | ----- codi font
|          | ----- jmeter_results
|                    | ----- resultats de jMeter
|                    | ----- PythonMarkeplaceJmeterResults.ipynb
| ----- Elixir Marketplace Plan.jmx
| ----- Java Marketplace Plan.jmx
| ----- Python Marketplace Plan.jmx
```

Els ‘resultats de jMeter’ són 1 o més arxius .csv que contenen els resultats dels tests realitzat a l’API de la carpeta on es troben.

Els arxius .jmx són les configuracions de jMeter per cada un dels tests fets a les API’s.

Els arxius .ipynb són els notebooks de Python on s’ha creat la visualització de boxplots i histogrames dels resultats dels tests fets a les API’s.

Els ‘codi font’ són les carpetes on es guarda el codi font fet servir per a crear les API’s.

Els dos ‘codi font’ fets en Java inclouen un fitxer .jar. Aquest fitxer és l’executable creat a partir del codi font fet servir per executar les API’s.