Definició, arquitectura i desplegament de l'aplicació SplitIt

Pol Gràcia Espelt

Resum - SplitIt és una aplicació de gestió de despeses grupals i personals que neix d'una idea en forma de start up. S'ha creat una arquitectura i un model de dades actual, potent i escalable capaç de suportar les demandes futures que pugui precisar l'aplicació per al seu correcte funcionament i donar una bona experiència al usuari final.

Paraules Clau - Front-end, Back-end, Base de Dades, Docker, Cloud, AWS.

Abstract - SplitIt is a group and personal expense management application that was born from an idea in the form of a start-up. A current, powerful, and scalable data architecture and model has been created capable of withstanding future demands that the application may require for its proper operation and a good user experience.

Keywords - Front end, Back end, Data Base, Docker, Cloud, AWS.

1. Introducció

En moltes ocasions es dóna la situació de trobar-se en un bar o sopant amb amics i a l'hora de pagar, no es pot dividir la compta, es crea una situació incòmoda on els que han consumit més o menys no volen dividir equitativament el tiquet, i doncs, ho ha de pagar un i reclamar els diners, sense mencionar que si la compta és llarga, dividir-la un a un és una tasca complexa i pesada.

Així doncs, neix SplitIt, una aplicació que té l'objectiu de donar solució a aquestes situacions i molt més. SplitIt és una aplicació que mitjançant un sistema d'intel·ligència artificial de lectura de tiquets i amb funcionalitat de xarxa social, permet dividir de manera còmoda i ràpida qualsevol tiquet o despesa entre els integrants d'un grup d'amics de manera no equitativa.

Fig 1: Funcionament SplitIt

Formalment, SplitIt crea un contracte social o acord entre uns individus o consumidors, dividint i adjudicant elements o productes a cada usuari, i formalitzat l'acord, fent-ne un seguiment i assegurant el compliment i pagament d'aquest deute.

L'aplicació utilitza una interfície d'usuari *'user friendly'* que permet a l'usuari de manera intuïtiva navegar per l'aplicació i fer anar les seves funcionalitats.

A més a més, SplitIt pot gestionar despeses personals no només grupals. Pots introduir les teves pròpies despeses en forma de tiquet o manualment, i automàticament, gràcies a l'ajuda d'intel·ligència artificial, agruparà les despeses en diferents grups, per veure-les i accedir-hi de manera més clara i senzilla, per tenir-ne un millor control. Els clients gaudiran d'un espai on podran veure les seves despeses en funció de diferents categories; despeses totals, despeses agrupades cronològicament, despeses agrupades per àmbit, etc.

1.1 Diagrama de flux funcional de l'aplicació

Per poder entendre què aporta SplitIt en un cas d'ús de negoci necessitem saber que és capaç de fer l'aplicació, per tant, el primer que s'explicarà seran les característiques funcionals de les quals disposa.

En l'apèndix podem veure els diagrames de flux de les funcionalitats de l'aplicació.

S'accedeix a la app mitjançant un url web des de qualsevol dispositiu. Un cop s'accedeix, s'introdueixen les credencials d'usuari o es registra. Es veurà la pestanya de Home, des d'allà es poden mirar les dades personals, carregar un tiquet o demanar recomanacions basades en les dades del usuari.

Dins l'aplicació es poden crear amistats cercant per nom d'usuari i agregant-los establint una relació. Quan es té un amic, es pot crear un grup, aquesta és la unitat amb la qual es divideixen les despeses posteriorment i se'n poden crear tants com es vulguin: amb amics, amb familiars, amb companys de viatges, etc. En funció del context es tindrà un grup o un altre per repartir les comptes.

A continuació només s'ha de consumir algun producte: ja sigui en un supermercat, un restaurant o una factura i es demana el tiquet.

Mitjançant un telèfon o un ordinador es pren o es puja una foto del tiquet i es processa mitjançant un algorisme d'intel·ligència artificial que analitza tots els elements d'aquest: A partir

d'aquí s'haurà de seleccionar entre quin grup es vol dividir el compte per saber amb quins usuaris dividir el tiquet (tot i que no és necessari que tots hagin consumit!).
A continuació es visualitza una nova pàgina on es troben els elements del tiquet desglossats. Per cada element del tiquet, es selecciona quina o quines persones l'han consumit i es divideix el seu preu entre aquestes. També s'ha de seleccionar qui ha pagat el tiquet, que al final és a qui se li deuen els diners.
Finalment, un cop s'hagin repartit tots els elements es podrà veure un desglossament del que ha consumit cadascú i el que pertoca pagar. Tots els membres del grup que hagin consumit, independentment, podran veure des de la seva aplicació quins són els elements que se'ls ha assignat i a qui han de pagar l'import corresponent.
Quan hagin pagat i el pagador hagi rebut els diners es podrà confirmar la transacció com a completada. Actualment, s'haurà de fer de manera manual, ja que no hi ha accés als detalls de transaccions bancàries dels usuaris tot i que es vol implementar en un futur.
En la figura 2 es pot observar la pestanya Home de l'app, a la qual s'accedeix quan s'ha introduït de manera correcta les credencials o s'ha registrat un nou usuari, com es pot veure hi ha una barra de navegació que permet moure's per a les diferents funcionalitats principals de l'aplicació, **Home, Profile, Tickets, Friends, Groups, Statistics**.

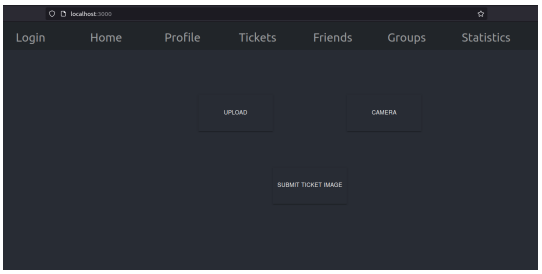


Fig 2: SplitIt pàgina Home

1.2 Funcionalitats de l'aplicació

Registre: Els nous usuaris han de crear un usuari, introduint les seves dades correctament en una pestanya de registre, per tal de poder accedir a l'aplicació.
***Login*:** Per tal de poder accedir a l'aplicació és necessari introduir les credencials d'usuari (nom d'usuari i contrasenya) i que aquestes coincideixin amb les d'un usuari registrat a la base de dades.
Visualització de perfil: Els usuaris poden veure i modificar les seves dades d'usuari.
Creació d'amistats: Els usuaris poden buscar i afegir a altres usuaris registrats en l'aplicació per poder interactuar amb ells posteriorment.
Creació de grups: Els usuaris poden crear grups d'usuaris amb les seves amistats.
Pujada d'imatges de tiquets: Els usuaris poden pujar imatges de tiquets a l'aplicació des del dispositiu o poden fer una foto d'un tiquet que pren l'aplicació automàticament.
Assignació de grup: L'aplicació permet a l'usuari que ha pujat un tiquet triar de quin grup es tracta la despesa per poder dividir-lo entre aquests usuaris.
Lectura dels elements dels tiquets: L'aplicació pot processar imatges de tiquets per tal de detectar elements com els productes, els preus, el nombre d'elements el lloc i hora de compra, etc.
Assignació d'elements del tiquet a usuaris: L'aplicació permet a l'usuari que ha pujat el tiquet assignar quin element del tiquet correspon a cada usuari. La manera és assignar un element del tiquet a tots els usuaris que l'han consumit iterativament per tots els elements del tiquet.
Visualització de despeses grupals individualment: Tots els usuaris d'un grup poden veure quins elements se'ls han assignat en la divisió d'un tiquet concret.
Gestió de comptes personals: L'aplicació permet fer diferents anàlisis sobre les despeses d'un usuari: per àmbit, cronològicament, per grup d'amics, etc.

1.3 Objectius globals

- Creació d'una aplicació web capaç de realitzar totes les funcionalitats descrites
- Creació d'una arquitectura actual, coherent, escalable i tolerant d'errors.
- Creació d'un model d'intel·ligència artificial capaç d'analitzar tiquets.
- Creació d'un programa d'anàlisi i explotació de dades generades per l'aplicació.

1.4 Objectius específics

- Definició dels components de l'aplicació
- Disseny model de dades.
- Definició de l'arquitectura.
- Anàlisi i solució dels colls d'ampolla.
- Desplegament al cloud.

1.5 Metodologia

Es treballa mitjançant una metodologia àgil per sprints. S'utilitza GitHub com a eina de control de versions i punt comú per compartir codi, l'eina de tasques del Microsoft Teams per gestionar les tasques per fer, fetes i els sprints. A part de la metodologia àgil, s'ha emprat també una eina per realitzar diagrames de Gantt per a planificar el projecte.

S'han fet dos diagrames; un comú per a tots els integrants del grup i un altre individual. (Veure a l'apèndix).

2. Aplicació web

En l'enginyeria del software es coneix una aplicació web[0] com aquelles eines amb les quals els usuaris poden fer ús accedint a un servidor web a través d'un navegador i de la xarxa.

Són populars a causa de la pràctica del navegador web com a client lleuger, són independents al sistema operatiu i és fàcil actualitzar i mantenir aplicacions sense distribuir i instal·lar programari a milers d'usuaris potencials.

Una aplicació web pot contenir elements que permeten una comunicació activa entre l'usuari i la informació i els serveis de l'aplicació. Això permet que l'usuari accedeixi a les dades de manera interactiva, ja que la pàgina respon a les accions de l'usuari.

Tot i que hi ha moltes variacions, generalment una aplicació web està estructurada com una aplicació de 3 capes.

En la seva forma més comuna, el navegador ofereix la primera capa, interpretant el codi. En la segona capa hi ha el servidor que ofereix aquest codi i tota la informació. Finalment, una tercera capa que és una o diverses bases de dades on es recopilen les dades.

2.1 Arquitectura en 3 capes i definició dels components

L'aplicació SplitIt, seguint l'arquitectura d'aplicacions web modernes, es construeix sobre 3 elements troncals sobre els quals es desenvoluparan les seves funcionalitats i requeriments. Es tracta d'una arquitectura en blocs.

Aquests 3 blocs són: un front-end on l'usuari interactua amb les opcions i els serveis que s'ofereixen, un back-end on es processen els requeriments de l'usuari i intermèdia la comunicació i transferència de dades entre les diferents capes de l'aplicació i una base de dades on s'estructuren i s'emmagatzemen les dades generades per als usuaris.

Per a l'elecció dels *frameworks* per cada capa s'han triat aquells que compleixen uns requisits d'actualitat i centralitat en el món tecnològic sobre els que ens interessa construir el projecte.

Per al **front-end** de l'aplicació s'utilitza el llenguatge ReactJS [1]. React és una biblioteca de Java Script de codi obert basada en components i estats. Ajuda a crear interfícies d'usuari interactives de forma relativament senzilla i es pot renderitzar fent ús de Node.js.

Per al **back-end** es fa servir el micromòdul de Python Flask[2]. Flask és un web framework minimalista que permet crear aplicacions web de manera senzilla i el que és més important per al nostre projecte, existeix una llibreria anomenada flask-sqlalchemy[3] que s'utilitza per connectar l'aplicació amb la base de Dades eficientment mitjançant ORM (Object-Relational Mapper).

Per a la **base de dades**, tot i haver considerat bases de dades no-relacionals com MongoDB, s'ha conclòs que per l'estructura de dades que volem emmagatzemar, és adequat emprar una Base de Dades Relacional com Postgres[4], ja que és *open source*, àmpliament documentada i representa les dades que fem servir.

3. Model de dades

El model de dades segueix una estructura relacional que busca minimitzar el nombre d'interaccions i representar de manera òptima les dades que s'emmagatzemen:

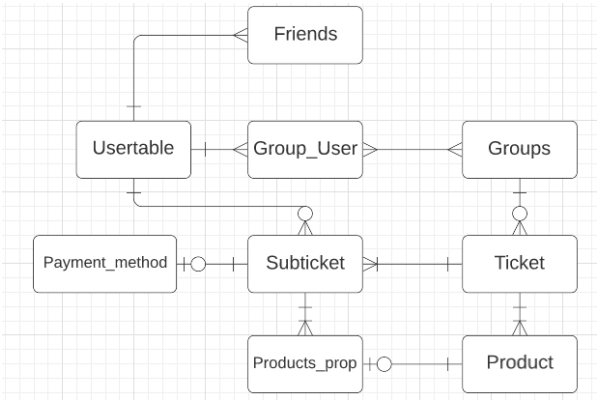


Fig 3. Esquema taules base de dades SplitIt (només taules), veure al apèndix esquema ER complert.

La Base de Dades de l'aplicació està formada per 9 taules, relacionades entre elles per diferents atributs.

Amb aquest disseny podrem fer totes les relacions entre productes i tiquets, usuaris i despeses, amics i grups que necessitem per a fer posteriors requests o anàlisis de dades.

Cal mencionar que les taules Friends, Products_prop i Group_user es definirien sense clau primària en un esquema relacional estàndard, però en el nostre cas utilitzarem ORMs per a establir la connexió a la BD i requereixen la definició d'una clau primària a totes les taules.

Usertable: Taula que conté la informació de l'usuari. La seva clau primària és el nickname de l'usuari.

Friends: Taula que conté la informació sobre les relacions entre els usuaris. La seva clau primària és friendship_id, té dos claus externes que relacionen amb dos usuaris respectivament.

Groups: Taula que conté la informació dels grups. La seva clau primària és el group_id.
Group_user: Taula que conté la informació sobre les relacions entre grups i usuaris. La seva clau primària és el groupuser_id i té dos claus externes, group_id que referència a la taula groups i nickname que referencia la taula usertable.
Ticket: Taula que conté la informació sobre un tiquet consumit per un grup. La seva clau primària és el ticket_id i té una clau externa group_id que la relaciona amb la taula Groups.
Product: Taula que conté la informació sobre els productes d'un tiquet. La seva clau primària és product_id i té dos claus externes, ticket_id que referència a Ticket i paid_by_id que relaciona amb un nickname de la usertable.
SubTicket: Taula que conté la informació sobre la part d'un tiquet assignat a un usuari. La seva clau primària és subticket_id. Té dos claus externes, nickname_user que referència un usuari de usertable i ticket_id que referencia un tiquet.
Payment_method: Taula que conté la informació sobre el pagament d'un subticket. La seva clau primària és payment_id i referencia el payment_id d'un subtiquet.
Products_prop: Taula que conté la informació sobre els productes consumits per un usuari. La seva clau primària és products_prop_id i té dos claus externes, product_id que referència un producte i subtiquet_id que referència un subtiquet.
En el model de dades cal destacar la importància de l'element que permet estructurar la divisió dels tiquets: La taula 'subticket' permet representar una despesa a un usuari només com la part que li pertoca pagar. Aquesta estructura és entregada al usuari com el seu deute.

4. Definició de l'arquitectura

Des d'una primera instància és prioritari que l'arquitectura de l'aplicació pugui escalar en funció de les necessitats i nombre d'usuaris, que sigui tolerant a errors i que permeti triar entre un ventall de possibilitats de servidors web on poder-la desplegar, independentment a les seves funcionalitats i característiques tècniques. És a dir, que en cap cas ens sentim limitats per una arquitectura rudimentària o poc eficient que ens pugui passar factura en moments crítics de màxim ús.
A més a més, es vol implementar la capacitat de tenir una aplicació que funcioni independentment al hardware, sistema operatiu i navegador que s'utilitzi i que pugui funcionar sense instal·lar forçosament per cada ús tots els paquets necessaris per a la seva execució.
És per aquests motius que després d'un complex estudi de serveis que puguin assolir aquests objectius s'ha decidit muntar l'aplicació en Docker.
Docker[5] és una plataforma de programari que permet crear, provar i implementar aplicacions ràpidament. Docker empaqueta programari en unitats estandarditzades anomenades contenidors que inclouen tot el necessari perquè el programari s'executi, incloses biblioteques, eines de sistema, codi i temps d'execució. Amb Docker, es pot implementar i ajustar l'escala d'aplicacions ràpidament en qualsevol entorn amb la certesa de saber que el vostre codi s'executarà.
S'ha estudiat com a contrapartida de Docker muntar l'aplicació en Linux containers (LXC), funcionen de manera molt similarment a Docker, però s'ha vist que en el mercat hi ha una adopció genèrica més gran del triat, així com la facilitat per importar certes llibreries de Python que s'utilitzaran per al desenvolupament d'aquesta.
Docker implementa una línia d'independència entre l'entorn on s'executa: sistema operatiu, hardware, fitxers, software… I l'execució de l'aplicació en el contenidor. Veure esquema a l'apèndix.
Docker inclou tot el necessari perquè el programari s'executi, sent així independent de la màquina on s'executi i el programari instal·lat. D'aquesta a més a més de satisfer els objectius d'arquitectura permet que es pugui treballar paral·lelament en el front-end, el back-end i la base de dades sense crear conflictes de compatibilitat entre els membres i optimitzant l'espai usat.

4.1 Arquitectura de Docker en blocs

De la mateixa manera que es defineix l'arquitectura en blocs de l'aplicació, és necessari crear un component dockeritzat per cadascuna de les capes, que funcioni en conjunció amb les altres.
Els contenidors que executaran cadascuna de les capes són:
Front end container

- **Docker React.js** que permet la interacció de l'usuari amb l'aplicació, donant-li així a l'aplicació part de la funcionalitat desitjada.

Back End container

- **Docker Python** amb la llibreria Flask, en forma de restful api.

Database container

- **Docker PostgreSQL** que rebrà les queries des de el back-end per poder escriure o llegir informació segons es requereixi.

En l'apartat de dockerització de l'aplicació es detallen els components de cada capa.

Finalment, s'integrarà el model d'intel·ligència artificial en l'arquitectura, la qual es comunicarà directament amb el back-end (Docker de Python Flask).

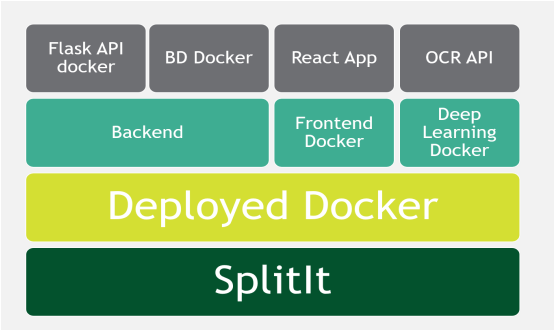


Fig 4. Arquitectura en Docker

4.2 Creació dels components Docker (dockerització)

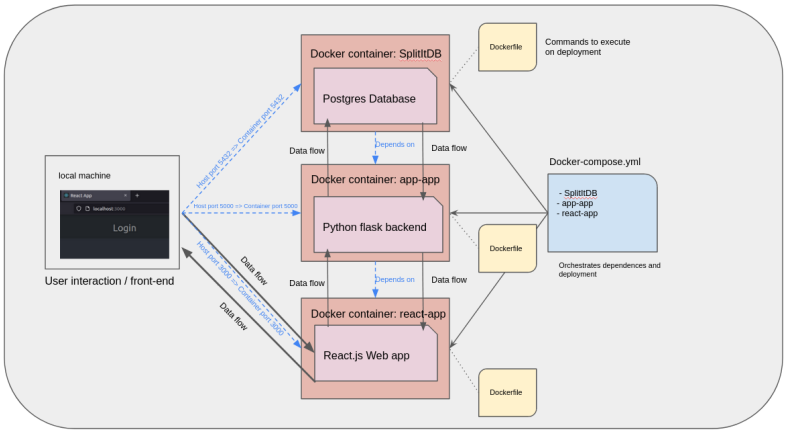


Fig 5. Esquema de dockerització

Per desplegar un contenidor són necessaris: un fitxer Docker-compose.yml, que orquestra quins contenidors s'han d'inicialitzar, de quina manera, en quin ordre i defineix variables globals, i un fitxer DockerFile que indica quines comandes s'han d'executar abans de desplegar el contenidor, com descarregar mòduls o configurar directoris.

En aquest cas s'ha organitzat el directori de treball en 3 subdirectoris: Base de dades, api i src. Cadascun d'aquests subdirectoris conté un Dockerfile amb les instruccions a realitzar per cada component independent.

En el directori root trobem un únic fitxer docker-compose.yml on s'indiquen els contenidors a desplegar i la localització relativa dels DockerFiles: en primer lloc, un postgres Docker que funcionarà en el port 5432, un flask Docker que funcionarà en el port 5000, depèn del Docker de postgres, i un React Docker que funcionarà en el port 3000, depèn del Docker de flask. És molt important mappejar correctament els ports entre màquina i contenidors per poder posteriorment dur a terme la connexió entre ells.

Es van trobar nombrosos problemes de mapejats de ports i el desplegament va ser un procés molt lent, ja que la primera vegada que s'executa el docker-compose s'han d'instal·lar totes les dependències i les imatges ocupen 5 GB en total.

Més en concret els Docker files executen les següents comandes quan despleguem els contenidors:

Base de dades: s'executa un fitxer SQL que crea les taules en la base de dades.

Api: S'executa el fitxer app.py on es defineix l'app i l'inicialitza en el mateix port que el Docker. Crea la connexió amb les taules de la base de dades amb objectes ORM i opcionalment també insereix mock data a la base de dades.

Src: s'instal·len tots els paquets de react i s'inicialitza el servei npm a la url localhost:3000.

4.3 Establiment de la connexió dels components de l'aplicació

Una vegada s'han definit els components independents de l'aplicació desplegats en contenidors, s'ha de configurar dins del codi de l'aplicació la comunicació entre ells.

El back-end i el front-end es comuniquen mitjançant rutes. Per establir la connexió és necessari definir en el proxy del front-end la ruta del contenidor del back-end. A continuació, una funcionalitat des del front-end crida a la funció 'fetch' que rep com a paràmetres una ruta i un mètode, al seu torn, en el back-end hem de tenir una funció implementada per aquella mateixa ruta que s'executarà quan es cridi des del front-end.

Per connectar el contenidor back-end amb el contenidor Base de dades, hem utilitzat el mòdul de flask 'flask-sqlalchemy' que permet connectar els components mitjançant ORMs[7]. Un ORM (Object-Relational Mapper) és un toolkit que ens ajuda a treballar amb les taules de la base de dades com si fossin objectes, de manera que cada taula és mapeja amb una classe i cada columna amb un atribut d'aquesta classe. A més, també ens permet mapejar les relacions entre taules com a relacions entre objectes. Una de les característiques que fan més atractives als ORMs és que pots canviar la teva base de dades gairebé sense modificar codi. De manera que pots programar la teva aplicació amb un ORM sense pensar que la base de dades és, per exemple, PostgreSQL o MySQL.

Definim el url on el Docker s'està executant en format de connexió del sql alchemy: 'postgresql+psycopg2://postgres:FDP@SplitItDB

:5432/postgres'. A partir d'aquest, ja podem definir les taules de la base de dades com a objectes de Python i fer-hi queries, inserir dades o modificar dades de manera còmoda sense haver d'executar directament codi sql.
Així doncs, per exemple, quan un usuari vol entrar a l'aplicació des de la pestanya 'login' del front-end, insereix el nom d'usuari i contrasenya en els objectes de React que prenen un input, aquest nom d'usuari i contrasenya s'envia amb el fetch i per la ruta '/login' al back-end que al seu temps, fa una query a la base de dades i comprova si el nom d'usuari existeix, si no existeix, retorna al front-end un missatge comunicant que l'usuari no està registrat, si existeix, comprova si hi ha un usuari amb aquest nom d'usuari i contrasenya a la base de dades, si no hi és, retorna un missatge comunicant que la contrasenya és incorrecta i si hi és, retorna un valor True i l'estat del front-end passa a ser la pantalla 'Home'.

5. Anàlisi de colls d'ampolla

Per analitzar amb coherència els colls d'ampolla de l'aplicació s'ha de distingir, en primer lloc, els dos punts d'execució d'aquesta: el *client side* i el *server side*.
El *client side* es refereix a la part de l'aplicació que es mostra a l'usuari i amb la que interactua directament. Això inclou el que veu l'usuari: text, imatges, navegació i qualsevol acció que es realitzi dins l'aplicació.
Pel que fa a l'execució, el codi d'aquest *side* en l'aplicació SplitIt, es correspon al codi de React, la navegació dins de l'aplicació, les funcions o interaccions escrites en JavaScript i la renderització. Aquestes es processen en el dispositiu de l'usuari i no en el servidor on es troba la web. Potenciar aquestes execucions resulta en una disminució de càrrega de còmput sobre el servidor.
El *server side* es refereix a la part de l'aplicació que s'executa en el servidor i no en el *client side*. Inclou totes les interaccions entre les diferents capes de l'aplicació (front-end, back-end i base de dades) transparents a l'usuari i l'execució de les funcions del back-end i queries a la base de dades.
Els colls d'ampolla del *client side* sobretot van lligats a un codi escrit ineficientment que porta a un mal ús del processador i la memòria.
Així doncs, ens centrarem en les ineficiències que podem trobar en el *server side,* ja que el *client side* s'executarà de manera més o menys eficient en funció del dispositiu on s'executi, tot i que el codi és eficient i els renderitzats i funcions no són complexes i s'haurien d'executar fluidament.
S'han d'analitzar 4 àrees que són sovint focus d'ineficiència en el *server side*: Veure a l'apèndix esquema de colls d'ampolla d'una aplicació web.

5.1 Funcions d'I/O (queries a la base de dades)

Les crides d'input/output que constitueixen una interacció entre el front-end, el back-end i la base de dades són sens dubte focus d'ineficiència.
Les bases de dades estan dissenyades per suportar grans volums d'operacions I/O i quan el nombre d'usuaris és petit rarament són un problema. De totes maneres, poden generar problemes si s'hi accedeix constantment per tasques trivials. En aquest cas, si tenim molts usuaris pot resultar en queries lentes i una mala experiència.
És per això que s'han d'utilitzar uns principis per als accessos a la BD:

- No executar queries dins de loops en el codi.
- Utilitzar operacions 'bulk' que carreguin moltes dades sempre que sigui possible.
- Considerar moure a memòria dades que s'accedeixen sovint.
- Planejar amb antelació queries intensives (com d'anàlisis de dades).

5.2 Ús de CPU

La CPU és la base del sistema, s'encarrega de fer anar la base de dades, processar-la i gestionar els requests dels usuaris. Quan tenim diversos clients interactuant simultàniament amb l'aplicació, el servidor assigna les tasques a diferents processos de CPU que s'executen en paral·lel o en una cua en funció del hardware disponible.
Les tasques es desenvoluparan sense dificultat sempre que el nombre de sol·licituds de clients estigui sota control i la càrrega combinada del processador estigui dins dels límits de la potència de la CPU disponible en el servidor web. Però si s'experimenta un trànsit intens inesperat, aquest nombre de requests poden superar la capacitat del servidor i el rendiment de l'aplicació es veurà afectat negativament.

5.3 Ús de memòria

La memòria en les aplicacions web es refereix als recursos de RAM del servidor. La mida de la memòria determina el nombre de processos que poden ser gestionats per la CPU i els que poden guardar-se en cua.
Una RAM insuficient o amb poca capacitat pot provocar que el procesador no rebi les dades prou ràpides i, per tant, alentint l'aplicació.

5.4 Ús de xarxa

Les condicions de xarxa també tenen un paper molt important en el rendiment de l'aplicació

web. La quantitat i la velocitat de dades que es transfereixen entre clients i servidor varien en funció de l'amplada de banda dels components intermediaris de la xarxa.
Quan el volum de transferència de dades és superior a l'amplada de banda de la xarxa, conegut com a 'hot spot', provoca que la transferència sigui lenta i que l'aplicació es congeli i no respongui, afectant l'usuari final.

5.5 Ús SplitIt

S'han realitzat diferents proves de rendiment sobre l'aplicació Split It per analitzar el consum de recursos d'aquesta. Els resultats a continuació es mostren per una instància i un usuari.
L'ús de cpu mitjà i tràfic de xarxa mig entre front-end, back-end i base de dades per les funcionalitats de l'aplicació es pot veure a la taula següent. Veure amb més detall a l'apèndix. Comentar que el tràfic de xarxa és variable en diferents funcionalitats en funció de diversos paràmetres com de nombre d'amics, tamany de la imatge del tiquet…

Mètode	tràfic mitja/request	ús cpu mitjà*
Get (login, profile, friends, group)	200B	1.30%
Post (register, add friend/group)	300B	2.70%
Upload ticket	Depèn imatge	13%
Statistics (mètode Get complex)	661B	26%

Taula 1. Ús de recursos per funcionalitat SplitIt

El temps d'execució de totes les funcionalitats és inferior a 500ms. Tal i com es pot apreciar pel tràfic de xarxa en la taula 1, els canvis en memòria tant en front end com en back end són molt petits al llarg del ús i convergeixen. Per aquests motius es consideren com constants o molt poc variants. Una instància de l'aplicació consumeix 1 GB en memòria com a base per funcionar i té el següent consum de memòria mitja per usuari en un ús extens i complert de les funcionalitats de l'aplicació. Veure gràfiques detallades al apèndix:

*Processador local Intel® Core™ i7-8550U 8ª generació, 4 cores

Capa	Memòria mitja / usuari
Front-end	60 MB
Back-end	140 MB

Taula 2. Ús de memòria SplitIt

El flux d'execució de l'aplicació en cap cas es estàtic i els valors mostrats poden variar i escalar en funció del nombre d'usuaris.
Per motius tècnics no s'ha pogut provar la diferència d'ús per més d'un usuari en una instància.

6. Solucions colls d'ampolla

6.1 Optimització de funcions d'I/O (queries a la base de dades)

La base de dades de l'aplicació web SplitIt és un postgres que s'executa en un Docker i es comunica amb el usuari a través del back-end.
La interacció es gestiona des del back-end mitjançant sql-alchemy, un mòdul que permet tractar les taules de la base de dades com objectes de Python i les seves columnes com atributs, facilitant l'accés a les dades (pel que fa a queries). A més a més s'utilitza flask-sql-alchemy que permet interactuar amb la base de dades més eficientment i en forma de restful api.
Concretament, dóna un avantatge estratègic de cara a les insercions de dades a la base de dades (mètodes 'post') gràcies a una funcionalitat en el mòdul de flask: ens permet la possibilitat d'interactuar amb la base de dades una única vegada, quan l'usuari tanca la sessió de l'aplicació.
És a dir, totes les dades que es vulguin guardar de l'usuari a la base de dades en el seu transcurs d'interaccions amb l'aplicació s'aniran guardant en memòria del client i s'insereixen com una operació 'bulk' quan finalitza la sessió amb un commit.
Això és possible gràcies al fet que sql alchemy fa un 'snapshot' de totes les queries d'inserció de dades i les dades a inserir en memòria i només s'executen quan es fa un 'commit()' i l'usuari tanca la connexió.
Les operacions de 'get' per la seva banda, és a dir, aquelles operacions en forma d'una query de tipus 'select' que pren dades de la base de dades sí que requereix una interacció directa cada vegada que es crida. En l'aplicació SplitIt, però, hem optimitzat aquestes crides fent que es guardin en memòria una vegada ja s'han executat (sempre que les dades a guardar siguin poques).
Per exemple, quan un usuari entra a la pestanya 'Profile' per veure el seu perfil en l'aplicació, es

crida un mètode 'get' del back-end que recupera de la base de dades el nom d'usuari, el nickname, la data de registre i els diners totals que s'ha gastat i ho mostra al front-end. Aquestes dades s'emmagatzemen en la taula 'user' de la base de dades i, per tant, aquesta operació suposa una interacció.

Per tal de minimitzar les interaccions amb la BD i en ser dades que ocupen poca memòria es guardaran com una variable en React, en el client, fent que si l'usuari vol tornar a accedir al seu perfil més endavant en la mateixa sessió no interactua amb la base de dades sinó que es carregaran de memòria.

Aquests són menors mètodes d'optimització d'I/O que ajuden al correcte flux de l'aplicació però no garanteixen que en un moment de màxima càrrega no es col·lapsi la base de dades, és per això que potenciarem també més capacitat de CPU i memòria a continuació.

6.2 Optimització d'ús de CPU, memòria i xarxa

Els problemes derivats d'una sobrecàrrega de CPU, memòria o xarxa són en la seva majoria solventables per l'augment de capacitat computacional i de recursos en qualsevol dels camps que representi coll d'ampolla.

7. Desplegament

Per definir la capacitat del nostre servidor tenim dues opcions d'infraestructura on desplegar l'aplicació:

Arquitectura on premise: Es tracta de tenir un o diversos servidors físics en les instal·lacions de l'empresa amb un equip d'IT encarregat del seu manteniment, seguretat i correcte funcionament.

Arquitectura cloud: Es tracta de contractar serveis a un 3r per poder utilitzar recursos computacionals de hardware, software o serveis que són propietat el proveïdor de cloud en qüestió. La infraestructura es gestiona en la xarxa i la responsabilitat del manteniment recau en el proveïdor.

7.1 Arquitectura cloud vs. on premise:

Ambdues solucions pretenen reduir els costos organitzatius i les càrregues de manteniment mitjançant una infraestructura informàtica àgil, tot i que s'ha demostrat que el cloud és més rendible i fàcil per a petites empreses i start ups com és el nostre cas, en part degut a la manca de necessitat inicial d'inversió per a comprar servidors i mantenir-los.

Les principals diferències són:

7.1.2 Hosting

En l'arquitectura on-premise, la infraestructura es troba en les instal·lacions de l'empresa. Això es tradueix directament en la necessitat de personal de manteniment per a la gestió del servidor.

En l'arquitectura cloud la infraestructura es manté a través d'internet. El proveïdor de serveis al núvol és qui s'encarrega de la gestió del servidor, incloent-hi escalabilitat, tolerància d'errors, replicació i manteniment.

7.1.3 Escalabilitat

En l'arquitectura cloud, en qualsevol moment es pot definir el nivell d'escalabilitat de l'aplicació i quins recursos volem utilitzar; podem definir uns serveis mínims i uns paràmetres d'escalabilitat: números de CPU, mida de memòria… a afegir en la infraestructura en cas de necessitat o punts d'alta concurrència. En l'arquitectura on-premise la capacitat computacional i de xarxa ve limitada per la infraestructura disponible i per l'amplada de banda contractat. Per escalar la infraestructura és necessari adquirir més CPU, memòria… I instal·lar-los en els servidors de l'aplicació.

7.1.4 Estalvi d'energia

Tenint en compte que els servidors consumeixen una quantitat molt elevada d'energia elèctrica, tenir un desplegament on-premise significa també muntar un sistema de refrigeració i fer-se càrrec de les factures. A part al no ser escalable, tot i no estar fent servir tots els recursos computacionals, pel fet de tenir-los disponibles i estar oberts ja consumeixen.

En canvi, els serveis en el cloud estan òptimament organitzats i muntats perquè consumeixin la menor energia possible (amb complexos sistemes de refrigeració i flux d'aire), i tot i que el càrrec d'electricitat ve inclòs en el preu del servei, el fet de tenir una arquitectura escalable fa que en la majoria dels casos no s'haurà de pagar electricitat com si es tinguessin tots els servidors oberts constantment.

7.1.5 Seguretat i control d'accés

En l'arquitectura on-premise és necessari definir i desplegar sistemes de seguretat, tant de xarxa com de software per assegurar la fiabilitat del sistema.

Els proveïdors cloud posen a la disposició dels usuaris nombrosos sistemes de control i seguretat per defecte, fàcils d'emprar que minimitzen la falta de seguretat.

8. Cloud

Per els diferents avantatges que aporta la infraestructura cloud respecte a l'arquitectura on-premise, s'ha pres la decisió de desplegar l'aplicació en aquest tipus d'arquitectura.

Tal com s'ha comentat, molts dels colls d'ampolla de la nostra aplicació són solventables amb més recursos computacionals, i això ho podem

explotar al cloud molt més fàcilment que on premise.
També cal mencionar que com a start up, això ens dóna avantatges claus econòmiques claus com no necessitar capital inicial per comprar servidors i contractar tècnics per a muntar-los i desplegar-hi l'aplicació.

8.1 Proveïdors cloud

Els proveïdors cloud són totes aquestes empreses que ofereixen components de computació sota demanda com Infrastructure as a Service (IaaS), Platfrom as a Service (PaaS) o Software as a Service (SaaS). En el nostre cas ens centrarem en la infraestructura com a servei, és a dir, una màquina escalable on desplegar la nostra aplicació tot i que també donarem ús a altres serveis inclosos com grups de seguretat i accés o software de testing i monitoratge.

8.2 Amazon Web Services

Amazon Web Services (AWS) [10] és el proveïdor cloud més complet i àmpliament utilitzat per empreses, que ofereix més de 200 serveis amb totes les funcionalitats dels centres de dades a escala mundial.
AWS ofereix les tecnologies d'infraestructura com la computació, l'emmagatzematge i les bases de dades que es requereixen per a la nostra aplicació. A més a més, essent el proveïdor cloud més gran també consta de la comunitat més gran, amb ple de documentació escrita i documental per aprendre a utilitzar amb profunditat tots els serveis de la plataforma.
Amazon web services serà el proveïdor cloud on desplegarem l'aplicació SplitIt pels motius exposats i per l'experiència prèvia amb aquests serveis.

9. Desplegament al cloud

Per desplegar l'aplicació dockeritzada i funcional en local al cloud s'ha de realitzar per un procés de migració al cloud.
En primer lloc, s'ha d'analitzar quins dels serveis oferits per Amazon Web Services són els que més s'adeqüen a l'arquitectura desenvolupada i ja funcional en local[11]. Els principals serveis que ofereix el proveïdor per a desplegar aplicacions dockeritzades són:
AMAZON EC2[12]: Amazon Elastic Compute Cloud és un servei basat en web que proporciona una potència informàtica escalable a AWS. Les empreses poden triar entre més de 500 instàncies (servidors) basades en processadors, emmagatzematge, memòria, xarxes, sistema operatiu i models de preus. Es poden fer servir per desplegar-hi Docker o qualsevol mena d'aplicació.
AMAZON ECS[13]: Amazon Elastic Container Service és un servei d'orquestració de contenidors totalment gestionat que ajuda a desplegar, gestionar i escalar aplicacions en contenidors fàcilment. S'integra profundament amb la resta de la plataforma AWS per proporcionar una solució segura i fàcil d'utilitzar per executar càrregues de treball de contenidors al núvol.
Permet desplegar aplicacions senzillament, ja que es pot connectar amb l'orquestrador de docker-compose, usat pel desplegament en local.

9.1 ECS vs. EC2

EC2 és un servei informàtic que permet que les aplicacions s'executin a AWS, mentre que ECS és un servei AWS que s'utilitza principalment per orquestrar contenidors Docker.
En essència, Amazon ECS és una agrupació lògica o un clúster d'instàncies EC2 que actuen com a hosts de Docker. ECS necessita instàncies EC2 per córrer. Són productes complementaris.
Per executar ECS a EC2, els usuaris necessiten un Amazon ECS Container Agent que s'executi en aquesta instància. S'executa un agent de contenidor a cada instància EC2 que està en funcionament dins d'un clúster d'Amazon ECS. L'agent és necessari per enviar informació sobre les tasques que s'executen i els recursos que s'utilitzen. Aquest agent pot començar i finalitzar les tasques segons sigui necessari en funció de les sol·licituds rebudes d'Amazon ECS.
A nivell d'escalabilitat, ECS afegeix clústers en lloc d'escalar recursos informàtics com EC2, és a dir, afegir més emmagatzematge, xarxes, memòria i càlcul.
És per aquests motius i per l'arquitectura base de l'aplicació que s'ha optat per desplegar-la en un servei d'Amazon ECS.

9.2 Desplegar SplitIt

Per pujar l'aplicació al cloud s'aprofita que l'aplicació es troba desplegada en una arquitectura dockeritzada, estructurada amb el fitxer de docker-compose que orquestra el desplegament dels diferents contenidors i els respectius dockerfiles per determinar com posar en funcionament cadascun. Això permet poder pujar l'app a l'unísson i més fàcilment que si fossin components independents a pujar i connectar entre diferents instàncies d'AWS en funció de la seva funcionalitat.
Esquema de desplegament a AWS:

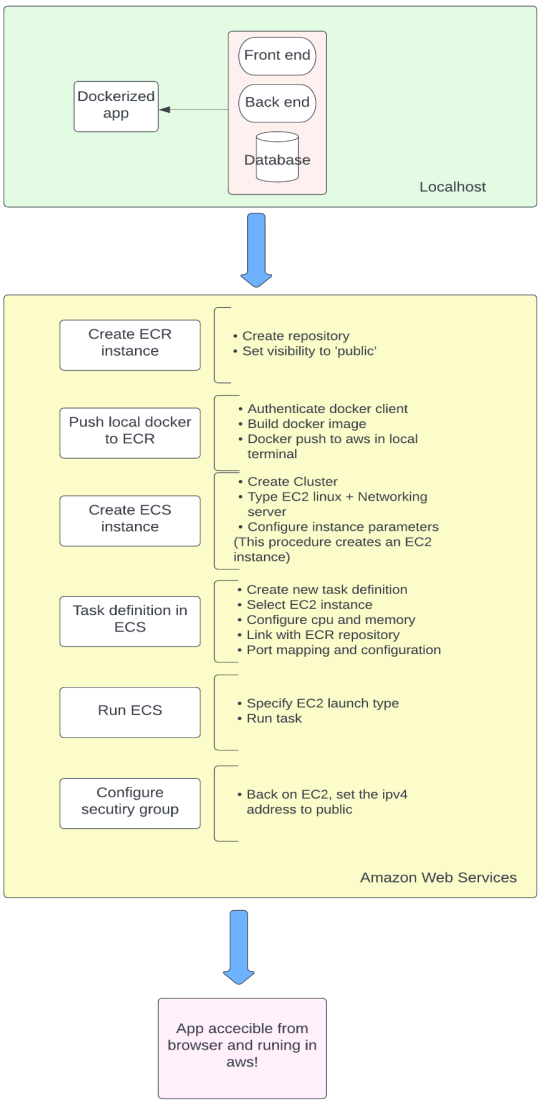


Fig 6. Ordre desplegament en AWS

El primer pas per desplegar l'aplicació al cloud és crear una instància d'Amazon ECR.

Amazon Elastic Container Registry [14] (Amazon ECR) emmagatzema, gestiona i desplega imatges de Docker. És a dir, ECR és un repositori que guarda tot el codi empaquetat com una imatge de Docker i ECS corre i utilitza el codi per fer anar les aplicacions sobre instàncies d'EC2. S'utilitza per pujar les imatges de local a AWS i viceversa. S'aprofita que l'aplicació es troba desplegada en una arquitectura dockeritzada, estructurada amb el fitxer de docker-compose que orquestra el desplegament dels diferents contenidors i els respectius dockerfiles per determinar com posar en funcionament cadascun. Això permet poder pujar l'app al unison amb la comanda de docker 'manifest' que permet orquestrar la pujada de diferents imatges, en aquest cas la imatge del back-end, front-end i BD.

Una vegada es pugen els fitxers locals a Amazon ECR via comandes específiques de Docker i AWS-cli, es crea una instància d'Amazon ECS. Per aquesta, configurarem una màquina EC2 de Linux amb un servidor de xarxa. Aquest procés internament crea un agent ECS que orquestra les imatges i un clúster EC2 on s'executaran els dockers.

A continuació es configuren les tasques i capacitat de les màquines EC2 on s'executaran les imatges. Es configura un ús on-demand i una màquina de tipus t2.xlarge. També es configuren els mappings de ports, i s'associa l'agent ECS amb el repositori d'ECR on estan emmagatzemades les imatges. Durant aquest procés es genera una adreça d'internet on podem connectar-nos per accedir a l'aplicació en el navegador.

Finalment i abans de poder connectar-nos serà necessari modificar el grup de seguretat de la instància i obrir-la com a pública. D'aquesta manera qualsevol persona podrà accedir a l'aplicació des del navegador.

9.3 Escalabilitat

Amazon ECS publica mètriques de CloudWatch amb la CPU mitjana i l'ús de memòria del servei. Es poden utilitzar aquestes i altres mètriques de CloudWatch per escalar el servei (afegir més tasques) per fer front a l'alta demanda en les hores punta o executar menys tasques per reduir costos durant els períodes de baixa utilització.

El proveïdor ofereix diferents formes d'escalabilitat automàtica[15] en funció de diferents paràmetres que són fàcilment configurables des de la consola. Concretament, es pot definir l'escalabilitat de cada instància de docker independentment mitjançant la comanda *describe-scalable-targets*. Això permet poder escalar, per exemple, el back-end sense necessitat d'escalar la base de dades.

9.4 Pressupost

Per calcular l'estimació de cost de desplegament i manteniment de l'aplicació en Amazon web services s'utilitza la calculadora de preus d'Amazon en la regió eu-central-1. Els detalls de les instàncies a desplegar, la configuració i el seu cost es desglossa en la taula següent. Cal mencionar que com s'ha explicat anteriorment les instàncies Amazon ECS són un servei

complementari amb Amazon EC2 que no té un cost adicional més que el de la mateixa màquina on es despleguen els contenidors.

Servei	Configuració	€/1 mes
ECR	10 GB + 1 GB de tràfic	0.60 €
EC2	Tipus t2.xlarge amb 4 vCPUs, 16GB RAM, fins a 5GB de xarxa, 30GB ssd, utilització on - demand amb un 70% utilització/mes (aprox)	128 €
Total	-	128.6€

Taula 3. Pressupost dels serveis a AWS

Aquesta instància, idealment, serviria per és la mínima per poder fer anar l'aplicació fluidament amb uns 75 usuaris simultàniament. S'ha de tenir en compte que també es configuren paràmetres d'escalabilitat que poden afectar en el preu estimat gràcies a les opcions de l'agent d'ECS. En un futur, es planteja utilitzar instàncies d'EC2 més potents i amb GPU integrada per poder utilitzar models més potents en el reconeixement de tiquets. També es vol mencionar que per finançar el projecte Amazon web services ofereix fins a 100.000$ en crèdits d'ús per a start-ups[16].

10. Conclusió

S'han assolit els objectius de crear una arquitectura per l'aplicació independent a les funcionalitats i característiques tècniques de la mateixa, escalable en funció de les necessitats i nombre d'usuaris, tolerant a errors, segura i desplegable al cloud. També s'ha creat un model de dades eficient que concorda amb les necessitats funcionals d'SplitIt i permet un flux de dades entre usuari i aplicació fluid.

Referències

[0] https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web Definició d'aplicació web.
[1] https://es.reactjs.org/ Web oficial del llenguatge react.
[2] https://flask.palletsprojects.com/en/2.1.x/ Web oficial de flask.
[3] https://flask-sqlalchemy.palletsprojects.com/en/2.x/ Documentació oficial del mòdul flask-sql.
[4] https://www.postgresql.org/ Web oficiaal de postgres.
[5] https://docs.docker.com/ Web oficial de docker.
[6] https://aws.amazon.com/es/docker/ Definció de docker segons aws.
[7] https://www.sqlalchemy.org/ Web oficial de sql alchemy.
[8] https://scoutapm.com/blog/performance-bottlenecks Focus d'ineficiències a la aplicació web.
[9] https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/ Desenvolupament del client side i el server side.
[10] https://aws.amazon.com/what-is-aws/?nc1=f_cc Web oficial d'Amazon Web Services.
[11] https://aws.amazon.com/containers/?nc1=f_cc Serveis de desplegament de contenidors a AWS.
[12] https://aws.amazon.com/ec2/ Amazon EC2 information main page.
[13] https://aws.amazon.com/ecs/ Amazon ECS information main page.
[14] https://aws.amazon.com/ecr/ Amazon ECR information main page.
[15] https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html Amazon escalabilitat en ECS.
[16] https://aws.amazon.com/activate/ Programa de financiació aws per start-ups

Apèndix

1. Diagrames de flux de les funcionalitats de l'aplicació

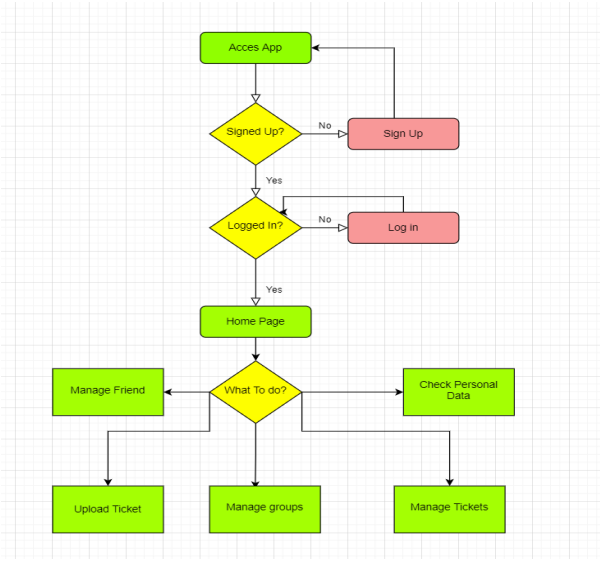


Diagrama de flux funcional de l'aplicació

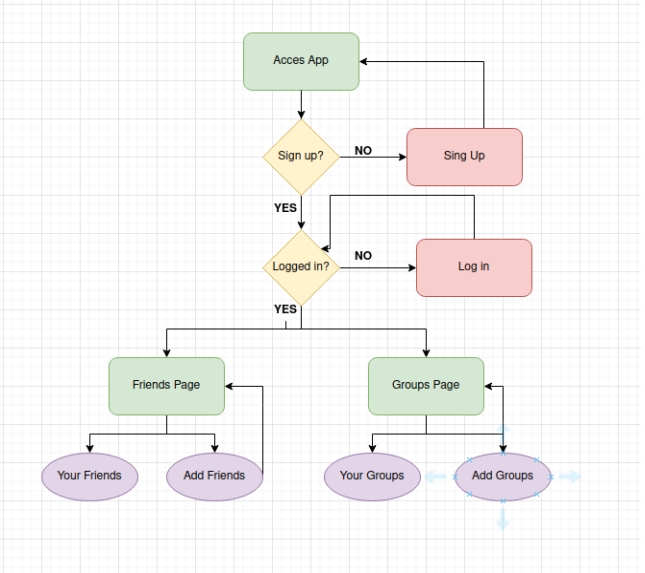


Diagrama de flux de pujar el ticket a l'aplicació

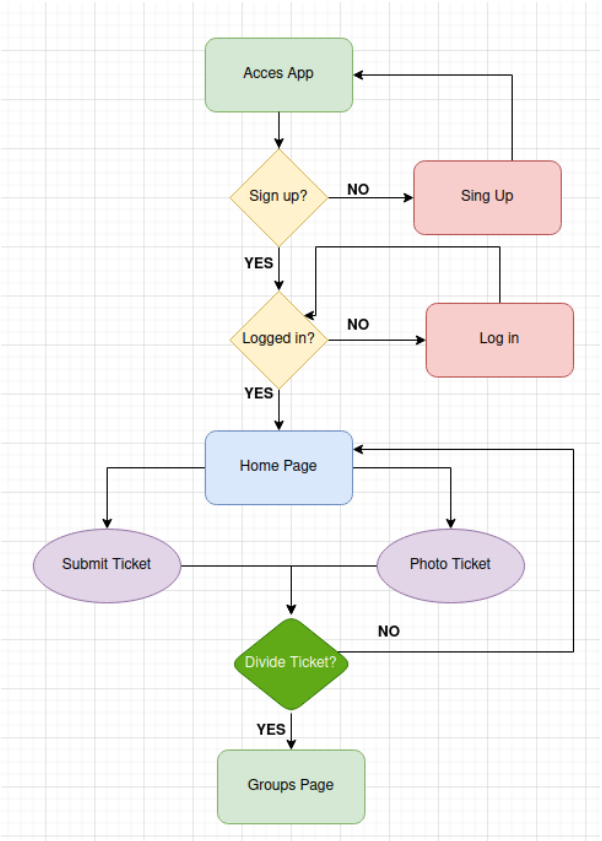


Diagrama de flux registre a l'aplicació

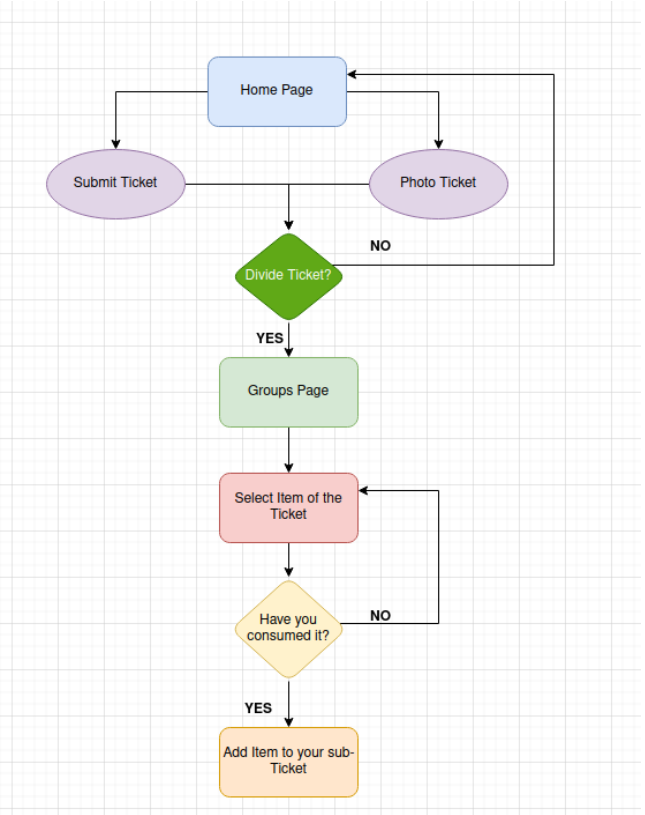


Diagrama de flux de mostra de tickets personals

2. Diagrames de Gannt per a la planificiació

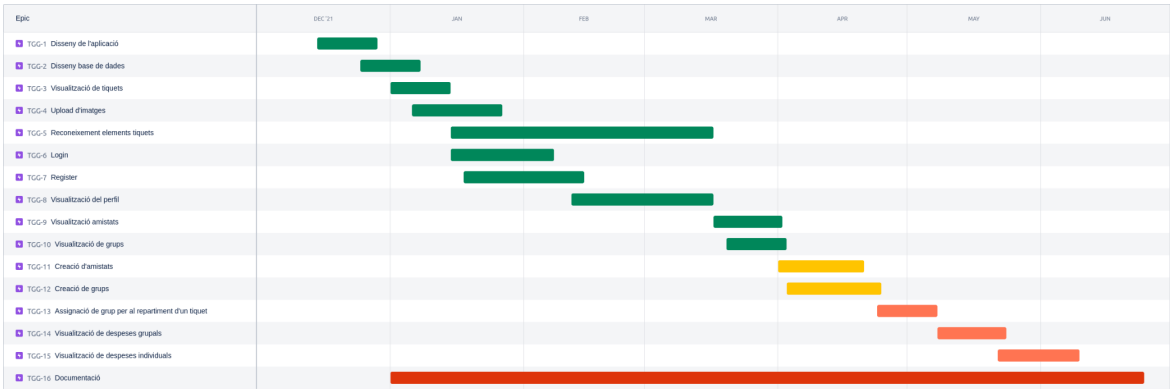


Diagrama de Gannt grupal

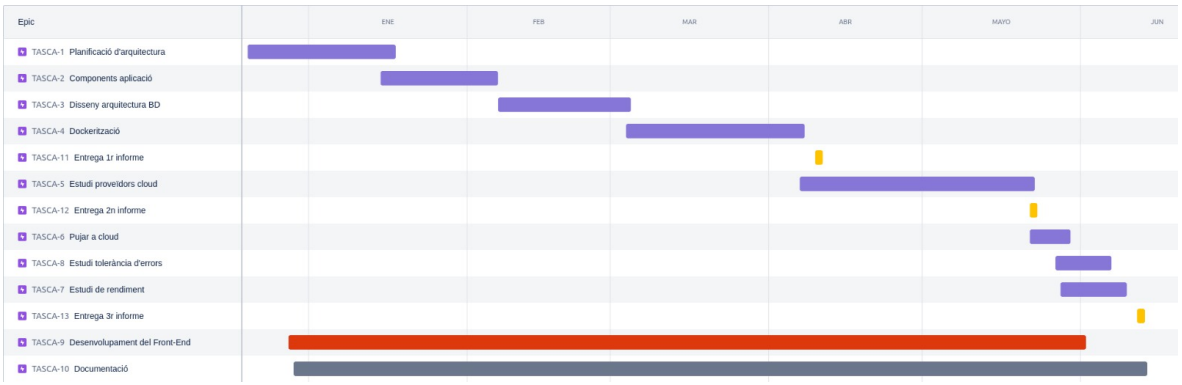
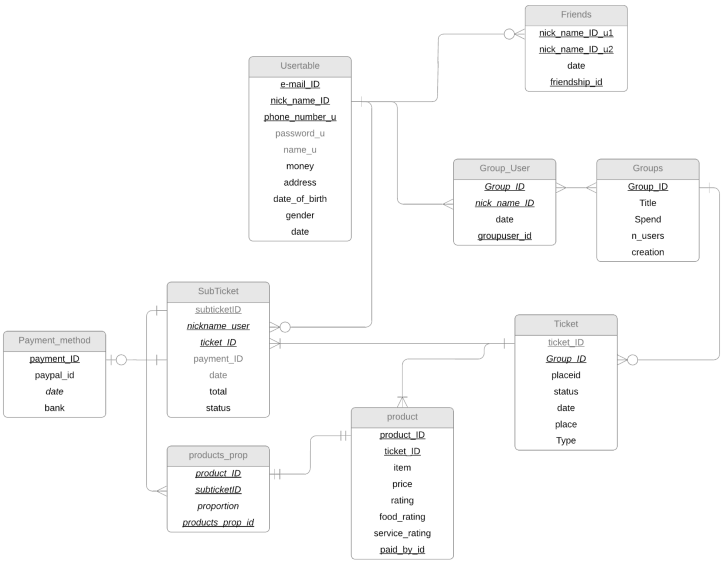
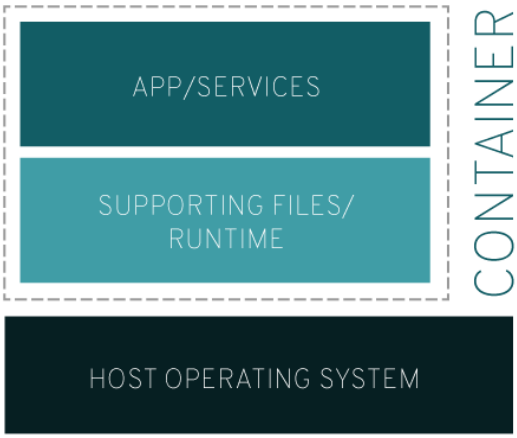


Diagrama de Gannt individual

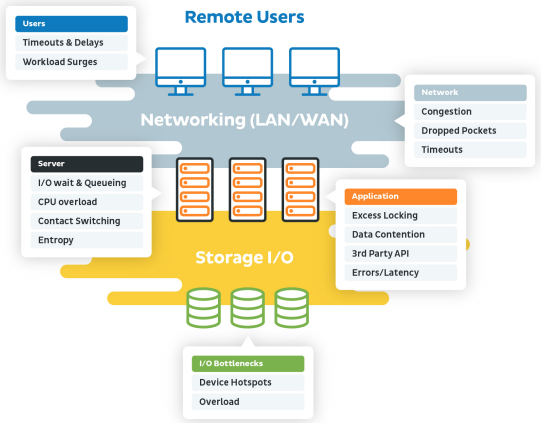
3. Model de dades



4. Esquema funcionament execució de docker

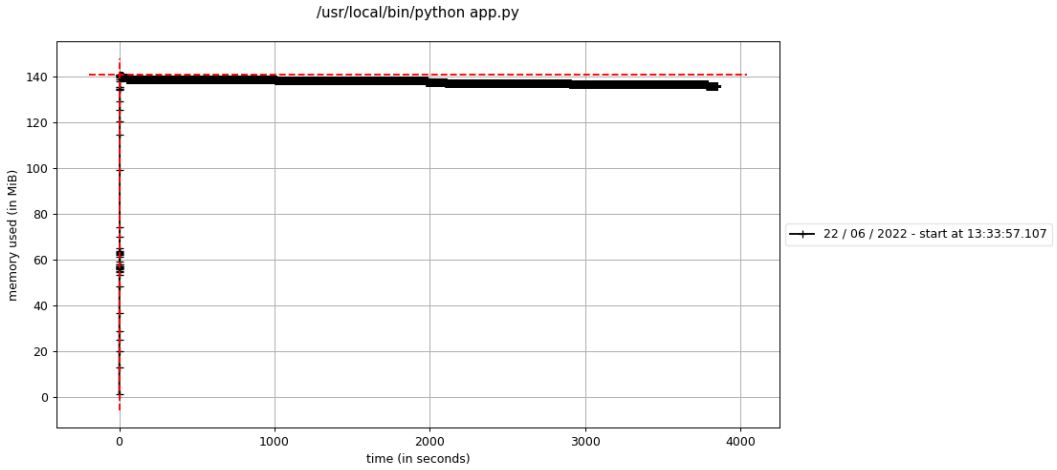


5. Esquema de colls d'ampolla d'una aplicació web



6. Back-end memory usage

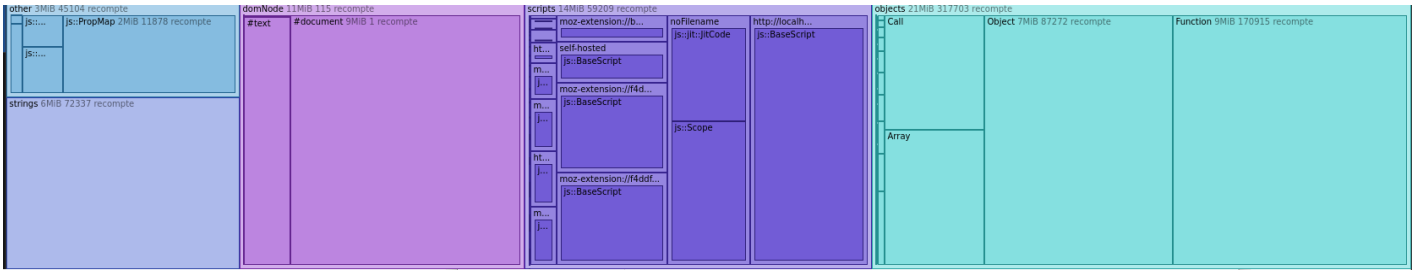
(en ús continuat de les funcionalitats de l'aplicació per 1 usuari)



7. Front-end memory usage

(en ús continuat de les funcionalitats de l'aplicació per 1 usuari)

Total de 60.05 MB



8. Trafic de xarxa per funcionalitat

Estat	Mètode	Domini	Fitxer	Iniciador	Tipus	Transferits	Mida
200	GET	localhost:3000	login?nickname=polgraciae@gmail.com&password=splitit	fetch	json	241 B	30 B
500	POST	localhost:3000	upload	fetch	html	2,77 KB	12,75 KB
200	GET	localhost:3000	profile?email=polgraciae@gmail.com	fetch	json	317 B	105 B
200	GET	localhost:3000	friends?email=polgraciae@gmail.com	fetch	json	401 B	189 B
200	GET	localhost:3000	profile?email=polgraciae@gmail.com	fetch	json	317 B	105 B
200	GET	localhost:3000	groups?email=polgraciae@gmail.com	fetch	json	306 B	95 B
200	GET	localhost:3000	statistics?email=polgraciae@gmail.com	fetch	json	661 B	449 B