Projecte Final de Grau

DESENVOLUPAMENT I IMPLEMENTACIÓ D'UNA BOTIGA ONLINE

16 de juny de 2019

Autor: Jordi Alvaro Arqués

Universitat Oberta de Catalunya

Consultor: Albert Grau Perisé
Professor: Santi Caballe Llobet

Títol del treball	Desenvolupament i implementació d'una botiga online
Nom de l'autor	Jordi Alvaro Arqués
Nom del consultor/a	Albert Grau Perisé
Nom del PRA	Santi Caballe Llobet
Data de lliurament	06/2019
Titulació o programa	Graud d'Enginyeria Informàtica
Àrea del Treball Final	Java EE
Idioma del treball	Català
Paraules clau	microservices, Spring, ReactJS

Resum del treball
Amb la generalització dels dispositius mòbils, ordinadors i l'Internet, ha aparegut un ventall d'aplicacions que ens faciliten algunes tasques de la nostra vida diària. En aquest sentit, s'han creat moltes empreses que es dediquen exclusivament a la venta online de productes. L'objectiu principal d'aquest projecte és aprendre i familiaritzar-se amb les noves tecnologies i tècniques que s'usen per a la implementació de sistemes al núvol. Per aconseguir-ho, s'ha dissenyat l'arquitectura d'una botiga online seguint una estructura de microserveis mantenint una separació de responsabilitats i els principis SOLID. Per al desenvolupament del *frontend* s'ha escollit les llibreries ReactJS i Bootstrap 4. També, s'ha utilitzat la llibreria Redux com a complement de ReactJS per a mantenir un estat global. Per al *backend*, s'ha preferit utilitzar Java 8 com a base i el *framework* Spring. A més, s'ha fet ús de les llibreries de Spring Cloud Netflix per tal de configurar l'entorn de microserveis. D'altra banda, s'ha dissenyat un procés d'integració contínua utilitzant TravisCI que monitoritza el desplegament de les noves funcionalitats del sistema a un entorn de producció basat en serveis de Heroku i Github Pages. També, s'ha pogut investigar sobre característiques relacionades amb l'arquitectura de microserveis com el servei de descobriment de microserveis i l'*API Gateway*, i desenvolupar funcionalitats utilitzant llibreries reactives com Project Reactor i bases de dades NoSQL com MongoDB. Aquest projecte presenta una implementació funcional i satisfactòria d'una botiga online, que tradicionalment s'hauria creat seguint un disseny en monolit, utilitzant una arquitectura de microserveis.

Abstract

With the generalization of the mobile devices, personal computers and Internet, a wide variety of applications which ease some of our daily tasks have appeared. With this in mind, newly created companies have emerged focusing only on the e-retail market.

The main objective of this project consists on learning and getting familiar with the new technologies and techniques that are used for the implementation of cloud systems.

In order to achieve it, the architecture of an online shop has been designed following a microservice pattern keeping a separation of concerns and the SOLID principles.

The ReactJS and Bootstrap 4 libraries have been selected to develop the frontend. Also, the Redux library has been used to complement it and maintain a global state. On the other hand, Java 8 with the Spring framework have been chosen to develop the backend. Moreover, the Spring Cloud Netflix libraries have been used to configure the microservice environment. Finally, TravisCI has been used to design a continuous integration process which monitors the deployment of the new funcionalities included in the system on a production environment composed by Heroku and GitHub Pages services.

Also, features of a microservice architecture such as Service Discovery and API Gateway have been investigated, and reactive functionalities and NoSQL patterns have been used with the help of Project Reactor and MongoDB.

This project shows a functional and successful implementation of an online shop using a microservice architecture, which traditionally would have been created following a monolith design.

Agraïments

A la meva estimada Mònica,
per ser la llum que il·lumina el meu dia,
per fer-me costat en els moments més difícils,
per la seva estima i paciència infinita.

Als meus estimats pares,
per ser l'estrella que em guia,
pel seu suport i estima incondicional,
perquè tot el que sóc és gràcies a ells.

Índex

Índex d'Algoritmes

Índex de figures

1 Introducció

1.1 Context i justificació del treball

Actualment, amb la generalització dels altaveus intel·ligents, gadgets, dispositius mòbils, ordinadors i l'Internet, ha aparegut un ventall d'aplicacions i serveis que ens faciliten i simplifiquen algunes tasques de la nostra vida diària.

En aquest sentit, s'han creat moltes empreses que es dediquen exclusivament a la venta online de productes. D'altres, que es basaven en un model de negoci tradicional amb botigues físiques, han hagut d'adaptar-se i posar a l'abast dels clients una plataforma de venta online.

D'aquesta manera, els usuaris tenen la possibilitat de cercar els productes pels quals estan interessats, comparar-los amb altres de similars o, inclús, amb el mateix producte en diferents portals per tal d'obtenir-lo a la millor relació de qualitat-preu.

Inicialment, la venta online estava principalment enfocada a la venta de dispositius electrònics (mòbils, tablets, televisions...), però, poc a poc, s'està expandint a tots els sectors de venta al detall. De fet, la compra setmanal que una família pugui realitzar en un supermercat ja es pot encarregar a través de la web i rebre-la a casa.

En els últims anys, la tecnologia web ha evolucionat molt, així com la metodologia i la definició, a nivell tècnic, de l'arquitectura d'aquestes aplicacions.

1.2 Objectius

L'objectiu principal d'aquest projecte és aprendre i familiaritzar-se de les noves tecnologies i tècniques que s'usen per a la implementació de sistemes al núvol i empapar-se de les noves i bones pràctiques de desenvolupament d'un sistema basat en microserveis. En aquest sentit, s'ha plantejat dissenyar l'arquitectura d'una botiga online seguint una estructura de microserveis, incloent serveis addicionals que sovint es troben ocults al desenvolupador com el *Service Discovery* i l'*API Gateway*.

Un altre objectiu consisteix en aprofundir en l'estudi del *framework* Spring i les seves llibreries més utilitzades per al *backend*, així com investigar sobre les llibreries ReactJS i Redux per al *frontend*. A més, també s'ha plantejat l'estudi de llibreries reactives, la tècnica *event sourcing* i bases de dades NoSQL.

També és molt important seguir els principis *SOLID* (*Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segration Principle, Dependency Inversion Principle*) i, així, crear un projecte amb una

qualitat de codi, mantenibilitat i legibilitat molt elevades.

El deute tècnic és una gran preocupació per a les empreses ja que ralentitza la velocitat de desenvolupament de funcionalitats quan les aplicacions arriben a un cert estat de maduresa. Si les aplicacions no s'han dissenyat amb cura, pensant en com poden evolucionar en un futur i amb un cert grau d'abstracció i generalització que permeti adaptar-se ràpidament a canvis i noves funcionalitats, poden comportar molts problemes per als desenvolupadors en un futur proper.

Per tant, tot i que s'ha implementat un *frontend* funcional i intuitiu amb les funcionalitats bàsiques, l'esforç no s'ha focalitzat exclusivament en els estils (CSS) i l'experiència d'usuari (UX).

Finalment, un altre objectiu que s'ha tingut en compte, és configurar un entorn de producció d'accés públic des d'internet i un procés d'integració contínua ben definit que permeti desplegar automàticament les noves versions que es desenvolupin.

1.3 Enfocament i mètode seguit

Donat que l'abast d'un projecte d'aquest estil pot ser molt ampli, se l'ha limitat tant a nivell tecnològic com a nivell de producte.

A nivell tecnològic, un sistema complet d'una botiga online, estaria format, simplificant-ho molt, per les següents aplicacions:

- Aplicació *Frontend* web. De tipus *responsive* per tal que es pugui visualitzar correctament als dispositius mòbils. Permet a l'usuari interactuar amb el sistema des d'un navegador.
- Aplicació Android. Permet a l'usuari interactuar amb el sistema des d'una aplicació nativa per a mòbils de tipus Android.
- Aplicació iOS. Permet a l'usuari interactuar amb el sistema des d'una aplicació nativa per a iPhones o iPads.
- Aplicació *Backend*. Part no visible del sistema que gestiona tota la lògica de negoci i persisteix les dades.

Donat que implementar totes aquestes aplicacions hauria complicat molt el projecte i no s'hagués pogut aprofundir tot lo desitjable en elles, s'ha decidit focalitzar-se en la implementació del *backend* i proporcionar una interfície bàsica de *frontend* web *no-responsive*. Per tant, l'aplicació resultant està únicament destinada a usuaris que utilitzin un ordinador.

A nivell de producte, s'ha escollit implementar les següents funcionalitats:

- Login.
- Llistat de productes.
- Perfil d'usuari.
- Carret de la compra.
- Històric de comandes per usuari.
- Resum de la comanda.
- Llistat d'usuaris.
- Històric de comandes del sistema.
- Crear un producte.

Cal notar que tots els usuaris no poden accedir a aquestes funcionalitats i casos d'ús, i depenendrà del rol que tinguin assignat. S'han definit tres tipus de rols:

- Usuari anònim. Només pot accedir a la visualització dels productes. La resta de funcionalitats estan desactivades o no se li mostren.
- Usuari bàsic registrat. Pot accedir a les funcionalitats bàsiques d'un usuari registrat: inici de sessió, visualització de perfil, afegir productes al carret, crear comandes i revisar el llistat de les seves comandes.
- Usuari administrador. A més, de poder accedir a totes les funcionalitats d'un usuari bàsic, també té accéss a poder crear productes, al llistat complet d'usuaris i comandes.

A part, cal comentar que, com no es tracta d'un projecte real, no s'ha implementat la integració amb cap passarel·la de pagament, simplement s'ha emulat la seva funcionalitat.

1.4 Planificació

Task Name	D...	Start	Finish	Pred
TFG	81d	20/02/19	12/06/19	
PAC 1	9d	20/02/19	04/03/19	
Planificació i descripció	9d	20/02/19	04/03/19	
PAC 2	25d	05/03/19	08/04/19	
Redacció de la memòria (seccions: Requeriments, anàlisi i disseny)	21d	05/03/19	02/04/19	3
Casos d'ús	5d	05/03/19	11/03/19	
Disseny relacional	3d	12/03/19	14/03/19	7
Diagrama de classes principals	5d	15/03/19	21/03/19	8
Diagrama d'arquitectura	3d	22/03/19	26/03/19	9
UX	5d	27/03/19	02/04/19	10
Configuració de l'entorn	7d	31/03/19	08/04/19	
PAC 3	33d	09/04/19	23/05/19	
Desenvolupament del sistema	33d	09/04/19	23/05/19	12
PAC 4	14d	24/05/19	12/06/19	
Redacció (secció: Implementació) i cloenda de la memòria	9d	24/05/19	05/06/19	13
Últimes modificacions a la memòria	5d	06/06/19	12/06/19	16
Preparació de la presentació	5d	06/06/19	12/06/19	16

Figura 1: Planificació del PFG.

1.5 Productes obtinguts

Els productes obtinguts durant el desenvolupament d'aquest projecte són:

- Compte a Github amb 7 repositoris corresponents als diferents microserveis necessaris per a fer funcionar la botiga online:
  https://github.com/jaarques-uoc
- Sistema desplegar a Heroku i Github Pages que es pot accedir des de qualsevol ordinador a través de la *URL*:
  https://jaarques-uoc.github.io
- Registre dels processos d'integració contínua a TravisCI:
  https://travis-ci.com/jaarques-uoc
- Registre de les imatges Docker que s'han creat:
  https://cloud.docker.com/u/jaarquesuoc
- Documentació a cadascun dels repositoris per tal de saber com executar el sistema.
- Memòria del projecte.

2 Disseny i anàlisi previ

Fins ara, s'ha introduït breument el producte a desenvolupar. En aquesta secció es detalla amb profunditat les diferents funcionalitats amb els seus casos d'ús, el diagrama de classes i el disseny de la interfície gràfica a la qual té accés l'usuari.

2.1 Model de casos d'ús i actors

Les figures 2 i 3 mostren els casos d'ús que s'han tingut en compte durant la implementació del projecte de la botiga virtual. Tal i com s'ha comentat anteriorment, existeixen tres tipus d'usuaris (usuari anònim o no registrat, usuari registrat bàsic i usuari registrat administrador) que poden accedir a unes funcionalitats determinades segons el rol que tinguin assignat.

S'ha volgut separar el cas d'ús especial d'iniciar sessió/registrar-se del diagrama complet de casos d'ús ja que només és visible per als usuaris no registrats. A més, afegir-lo amb el diagrama general de casos d'ús complicaria la seva visibilitat i entenibilitat ja que molts dels casos d'ús inclouen el de login per defecte. El diagrama general de casos d'ús mostra tota la resta de funcionalitats per a usuaris registrats (bàsic i administrador) i no registrats.

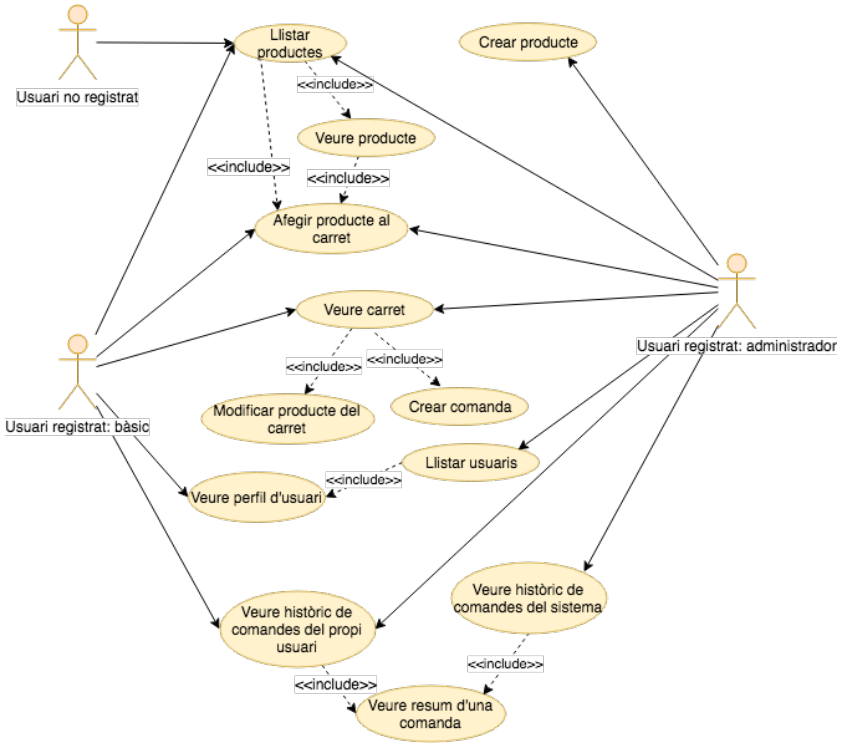


Figura 2: Diagrama complet de casos d'ús.

Figura 3: Diagrama del casos d'ús de registre i inici de sessió.

A continuació, es detalla l'especificació dels diferents casos d'ús:

Cas d'ús: Registre/Login	
Actors	Usuari no registrat
Precondició	L'usuari no ha iniciat sessió.
Descripció	Un usuari accedeix a la web per tal d'iniciar una sessió.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que no ha iniciat sessió i li mostra el botó d'iniciar sessió a la barra superior de navegació. 3. L'usuari clica el botó. 4. El sistema redirigeix a l'usuari a la pàgina d'inici de sessió. En aquesta pàgina també es mostra un botó per a registrar-s'hi. 5. L'usuari, que no s'ha registrat mai al sistema, clica al botó de registrar-se. 6. El sistema redirigeix a l'usuari a la pàgina de registre. 7. L'usuari introdueix l'email, la contrassenya, el nom complet, l'adreça i el país de residència. 6. El sistema detecta que tota la informació està afegida i habilita el botó de registrar usuari. 7. L'usuari clica el botó de registrar. 8. El sistema mostra un missatge de confirmació de registre i d'inici de sessió.
Postcondició	L'usuari ha iniciat sessió al sistema.
Excepcions a)	5. L'usuari, que ja s'ha registrat prèviament al sistema, introdueix l'email i la contrassenya. 6. El sistema mostra un missatge de confirmació d'inici de sessió.
Excepcions b)	8. El sistema mostra un error degut a que aquest email ja ha estat registrat.

Cas d'ús: Veure productes	
Actors	Usuari no registrat, registrat bàsic i administrador
Precondició	-
Descripció	Un usuari accedeix a la web per tal de veure els productes.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema mostra el llistat de productes. 3. L'usuari clica en un dels productes. 4. El sistema mostra la fitxa descriptiva del producte.
Postcondició	-
Excepcions	-

Cas d'ús: Crear un producte	
Actors	Usuari registrat administrador
Precondició	L'usuari ha iniciat sessió com a administrador.
Descripció	Un administrador accedeix a la web per tal de crear un producte.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que es tracta d'un usuari de tipus administrador i mostra a la barra de navegació un botó de crear producte. 3. L'usuari clica el botó de crear producte. 4. El sistema redirigeix a l'usuari a la pàgina de crear producte. 5. L'usuari afegeix la informació del producte: nom, descripció, preu i enllaç *URL* a la seva imatge. 6. El sistema detecta que tota la informació està afegida i habilita el botó de crear el producte. 7. L'usuari clica el botó de crear el producte. 8. El sistema mostra un missatge de confirmació de que el producte s'ha creat correctament.
Postcondició	L'administrador ha creat un producte.
Excepcions	8. El sistema mostra un missatge d'error avisant de que el producte no s'ha pogut crear.

Cas d'ús: Crear comanda	
Actors	Usuari registrat bàsic i administrador
Precondició	L'usuari ha iniciat sessió.
Descripció	Un usuari accedeix a la web per tal de crear una comanda.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema mostra el llistat de productes. 3. El sistema detecta que l'usuari ja ha iniciat sessió i mostra el botó d'afegir el producte al carret. 4. L'usuari clica el botó d'afegir el producte al carret. 5. L'usuari clica el botó de veure el carret. 6. El sistema redirigeix a l'usuari a la pàgina del carret i li mostra el resum de productes i la quantitat seleccionada de cadascun d'ells. 7. El sistema detecta que hi ha, com a mínim, un element al carret i habilita el botó de realitzar la comanda. 8. L'usuari clica el botó de realitzar comanda. 9. El sistema li informa que s'ha realitzat la comanda correctament i se li mostra el número de comanda corresponent.
Postcondició	L'usuari ha creat una comanda.
Excepcions a)	2.1. L'usuari clica en un dels productes. 2.2. El sistema mostra la fitxa descriptiva del producte. 3. Continua el procés normal des del punt 3.
Excepcions b)	4.1. L'usuari repeteix el procés des del punt 2 tantes vegades com vulgui. 5. Continua el procés normal des del punt 5.
Excepcions c)	4.1. El sistema detecta que aquest producte ja ha estat afegit i augmenta el comptador del nombre d'elements d'aquest tipus de producte. 5. Continua el procés normal des del punt 5.
Excepcions d)	6.1. L'usuari clica al botó de + o - per tal d'afegir o reduir el nombre d'elements d'un producte. 6.2. El sistema actualitza correctament el comptador d'elements del producte seleccionat. 7. Continua el procés normal des del punt 7.
Excepcions e)	6.1. L'usuari clica al botó de - per tal reduir el nombre d'elements d'un producte que només té 1 element. 6.2. El sistema detecta que la quantitat d'elements demanat per aquest producte és 0 i l'elimina del carret. 7. Continua el procés normal des del punt 7.
Excepcions f)	7. El sistema detecta que no hi ha cap element al carret i no habilita el botó de realitzar la comanda.
Excepcions g)	9. El sistema mostra un missatge d'error avisant de que la comanda no s'ha pogut crear.

Cas d'ús: Llistar usuaris	
Actors	Usuari registrat administrador
Precondició	L'usuari ha iniciat sessió com a administrador.
Descripció	Un administrador accedeix a la web per tal de veure el llistat d'usuaris del sistema.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que es tracta d'un usuari de tipus administrador i mostra a la barra de navegació un botó de llistar usuaris. 3. L'usuari clica el botó de llistar usuaris. 4. El sistema redirigeix a l'usuari a la pàgina del llistat d'usuaris. 5. L'usuari clica a sobre del nom d'un dels usuaris. 6. El sistema redirigeix a l'usuari a la pàgina del perfil de l'usuari seleccionat.
Postcondició	-
Excepcions	-

Cas d'ús: Veure el perfil de l'usuari	
Actors	Usuari registrat bàsic i administrador
Precondició	L'usuari ha iniciat sessió.
Descripció	Un usuari accedeix a la web per tal de veure el seu perfil.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que es tracta d'un usuari registrat i mostra a la barra de navegació un botó per a accedir al seu perfil. 3. L'usuari clica el botó de perfil. 4. El sistema redirigeix a l'usuari a la pàgina del seu perfil.
Postcondició	-
Excepcions a)	-

Cas d'ús: Llistar comandes d'un usuari	
Actors	Usuari registrat bàsic i administrador
Precondició	L'usuari ha iniciat sessió.
Descripció	Un usuari accedeix a la web per tal de veure les seves pròpies comandes.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que es tracta d'un usuari registrat i mostra a la barra de navegació un botó per a accedir a les seves comandes. 3. L'usuari clica el botó de llistar les seves comandes. 4. El sistema redirigeix a l'usuari a la pàgina del llistat de totes les seves comandes realitzades. 5. L'usuari clica a sobre de l'identificador d'una de les comandes. 6. El sistema redirigeix a l'usuari a la pàgina del resum de la comanda.
Postcondició	-
Excepcions a)	5. L'usuari encara no ha realitzat cap comanda i el llistat es troba buit.

Cas d'ús: Llistar totes les comandes dels usuaris del sistema	
Actors	Usuari registrat administrador
Precondició	L'usuari ha iniciat sessió com a administrador.
Descripció	Un administradir accedeix a la web per tal de veure el llistat de comandes del sistema.
Procés	1. L'usuari accedeix a la pàgina principal. 2. El sistema detecta que es tracta d'un usuari registrat de tipus administrador i mostra a la barra de navegació un botó de llistar totes les seves comandes del sistema. 3. L'usuari clica el botó de llistar totes les comandes. 4. El sistema redirigeix a l'usuari a la pàgina del llistat de totes les comandes realitzades per tots els usuaris. 5. L'usuari clica a sobre de l'identificador d'una de les comandes. 6. El sistema redirigeix a l'usuari a la pàgina del resum de la comanda.
Postcondició	-
Excepcions a)	5. Cap usuari encara no ha realitzat cap comanda i el llistat es troba buit.

2.2 Diagrama de classes

El diagrama de classes del sistema és el mostrat a la figura 4. És un diagrama bastant senzill degut a que la complexitat del sistema recau en la seva arquitectura i disseny de microserveis, més que en el model de classes utilitzat.

Cal remarcar que un `OrderItem` manté una relació de composició tant amb `Order` com amb `Cart`, per tant, `OrderItem` no existirà per si sola, sempre anirà associada amb una de les dues classes anteriors. A més, un `OrderItem` sempre anirà associat a un producte.

També cal comentar que una comanda `Order` tindrà, com a mínim, un carret associat. Això és així perquè es manté l'històric de carrets i no se n'eliminen els antics. Simplement, el carret més actual serà a partir del qual es realitzarà la comanda. Per altra banda, un carret pot tenir o no, una comanda associada, ja que pot ser que mai es faci efectiva.

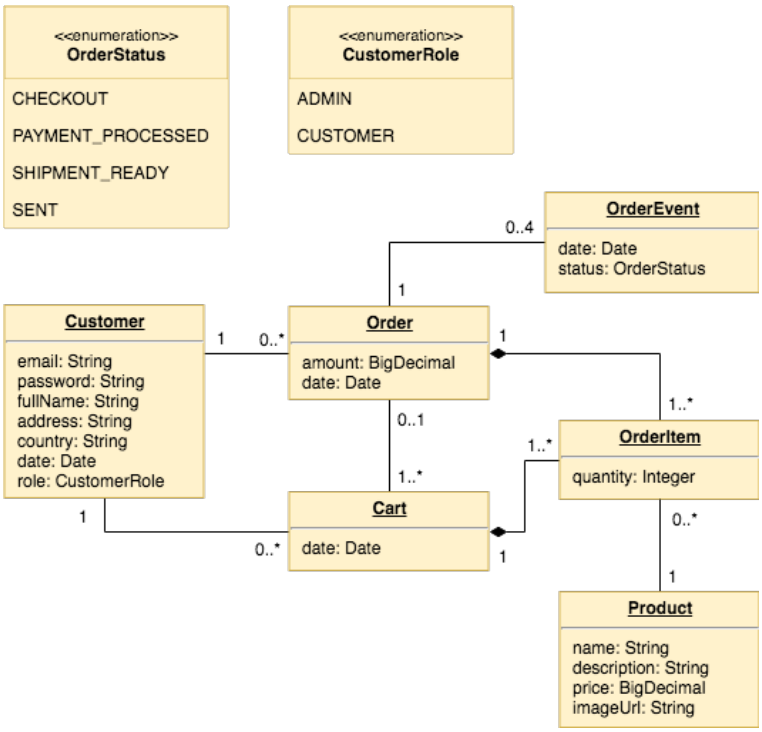


Figura 4: Diagrama de classes del sistema.

Per la mateixa raó, un usuari `Customer`, pot tenir un nombre indeterminat de carrets i comandes associades a ell ja que es manté tot l'històric.

D'altra banda, s'ha pres la decisió de no crear una herència amb dos classes addicionals a `Customer` (per exemple, podrien ser `BasicCustomer` i `Admin`) perquè no aporta cap valor extra en la implementació d'aquest sistema i, donat que es tracta d'un sistema basat en microserveis, s'hauria de retornar igualment una variable indicant el rol de l'usuari per tal que els altres sistemes i el *frontend* puguin determinar si es tracta d'un usuari bàsic o un administrador sense haver de duplicar crides *REST*. També, la implementació d'una herència de classes a una base de dades NoSQL és més complicada de gestionar que en una de relacional.

2.3 Model de pantalles (*UX*)

A continuació, es mostren els dissenys preliminars de les pantalles que s'han dissenyat per a l'aplicació juntament amb les captures de pantalla del disseny i implementació final.

Cal mencionar que els botons de navegació de la part superior, varien en funció del rol de l'usuari:

- Si no està registrat només pot accedir a les pantalles bàsiques de llistat i detall de productes, i es mostra el botó d'identificar-se o registrar-se al sistema.
- Si està registrat i és un usuari bàsic pot accedir a la resta de pàgines (excepte al llistat d'usuaris, al llistat de totes les comandes del sistema i a la pàgina de creació de producte) i es mostren tres botons a la barra de navegació: les seves comandes, el carret i el seu perfil.
- Si està registrat i és un usuari administrador, pot accedir a totes les pàgines i, a més, es mostren tres botons addicionals que li donaran accés a la pantalla de llistat d'usuaris, llistar totes les comandes del sistema i crear un nou producte.

Es pot observar com els resultats finals han seguit els dissenys inicials, tot i que, en algun cas, s'ha afegit algun aspecte que no estava contemplat prèviament. Un exemple d'això és la taula d'events que es mostra a la pantalla de resum de la comanda (figura 22), on es pot observar els diferents esdeveniments i passos que s'haurien de realitzar després de la creació d'una comanda (aquesta part no està implementada al projecte, simplement està emulada).

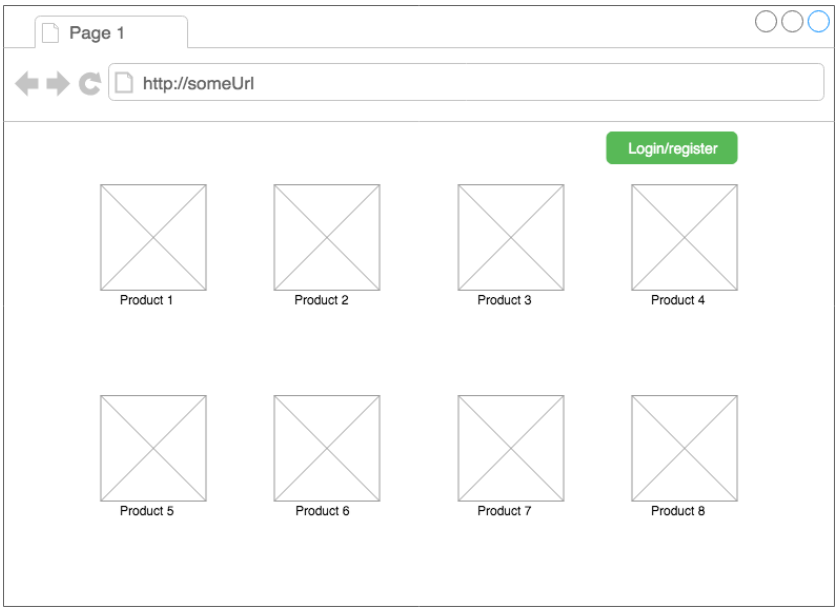


Figura 5: *Mock up* de la pantalla del llistat de productes a comprar

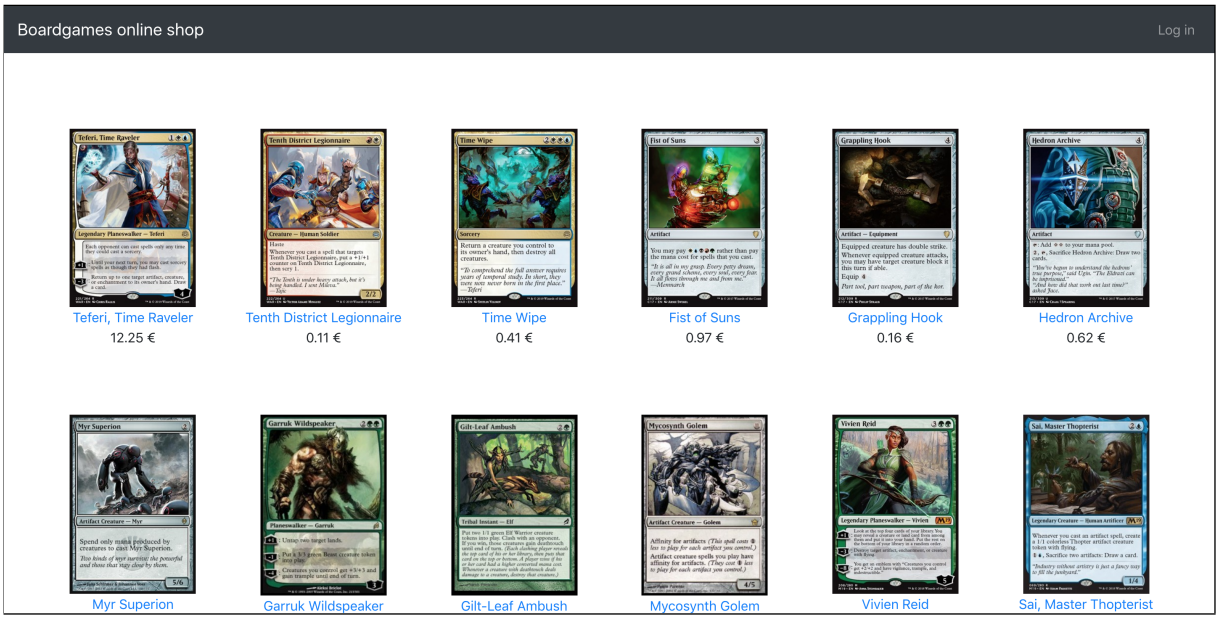


Figura 6: Captura de la pantalla del llistat de productes a comprar

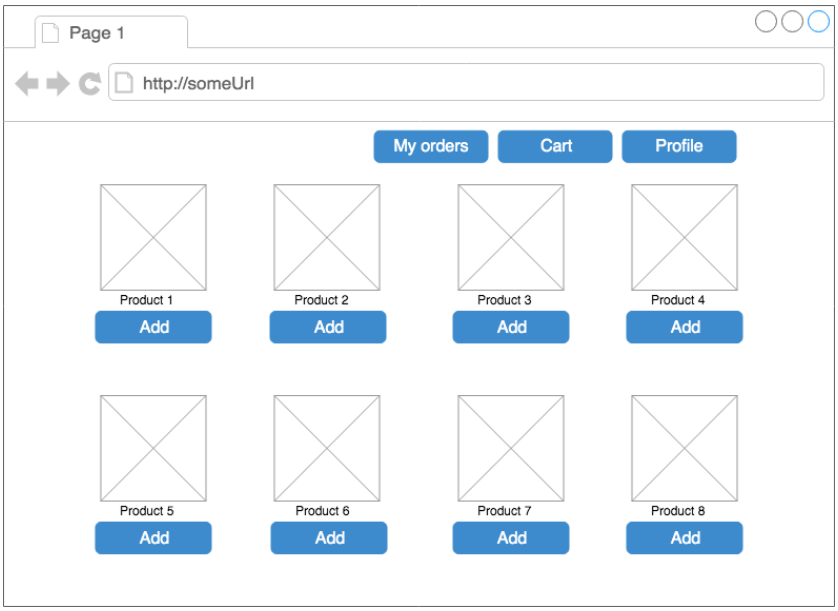


Figura 7: *Mock up* de la pantalla del llistat de productes per un usuari amb sessió iniciada

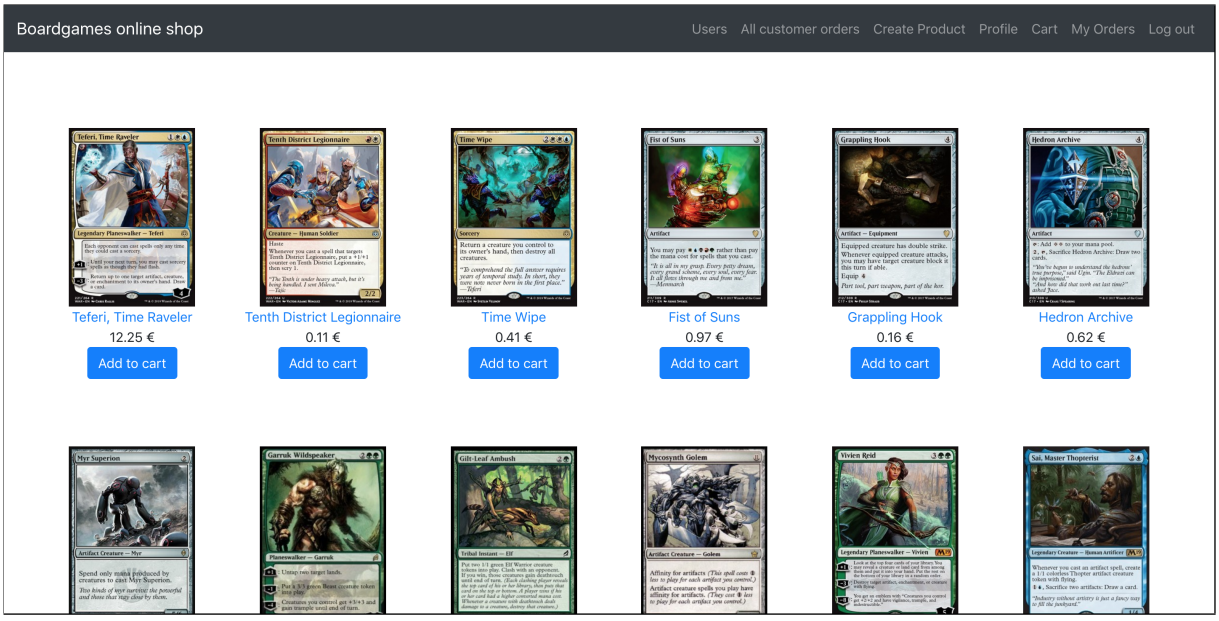


Figura 8: Captura de la pantalla del llistat de productes per un usuari amb sessió iniciada

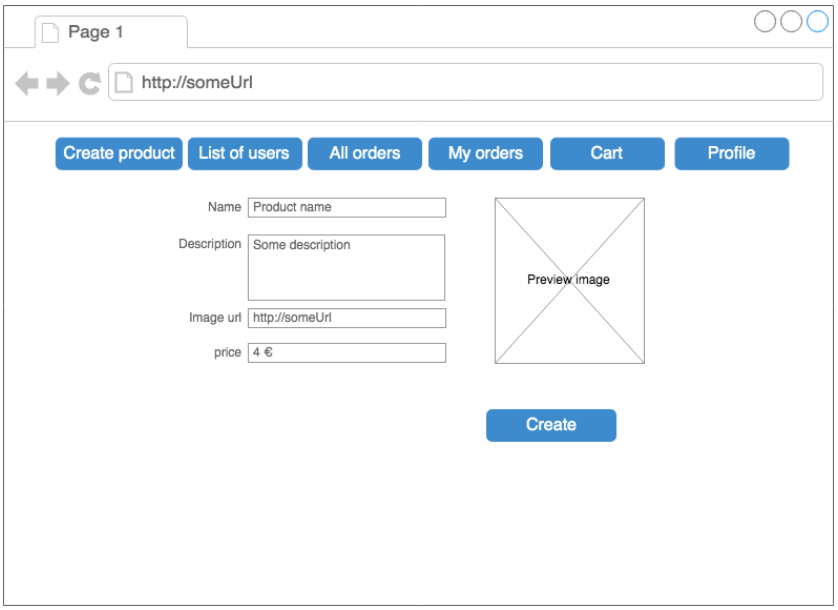


Figura 9: *Mock up* de la pantalla de creació del producte

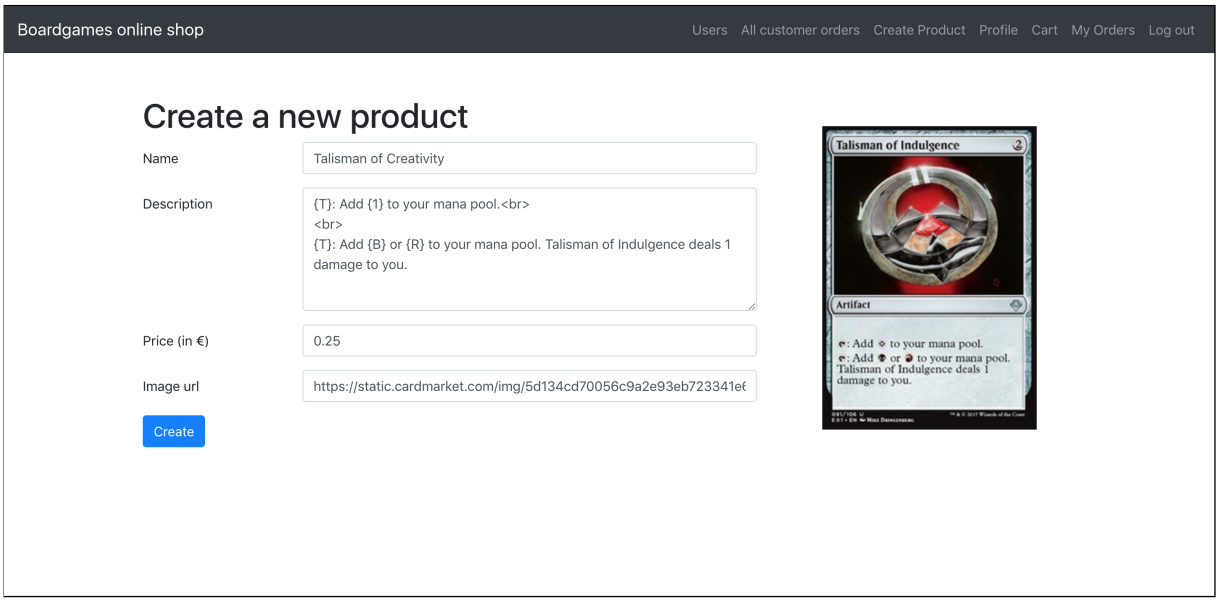


Figura 10: Captura de la pantalla de creació del producte

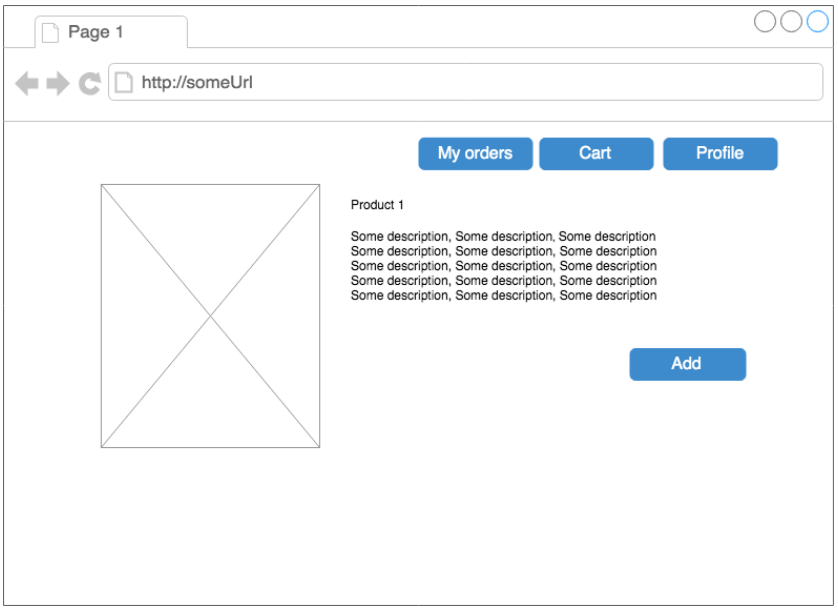


Figura 11: *Mock up* de la pantalla de descripció del producte

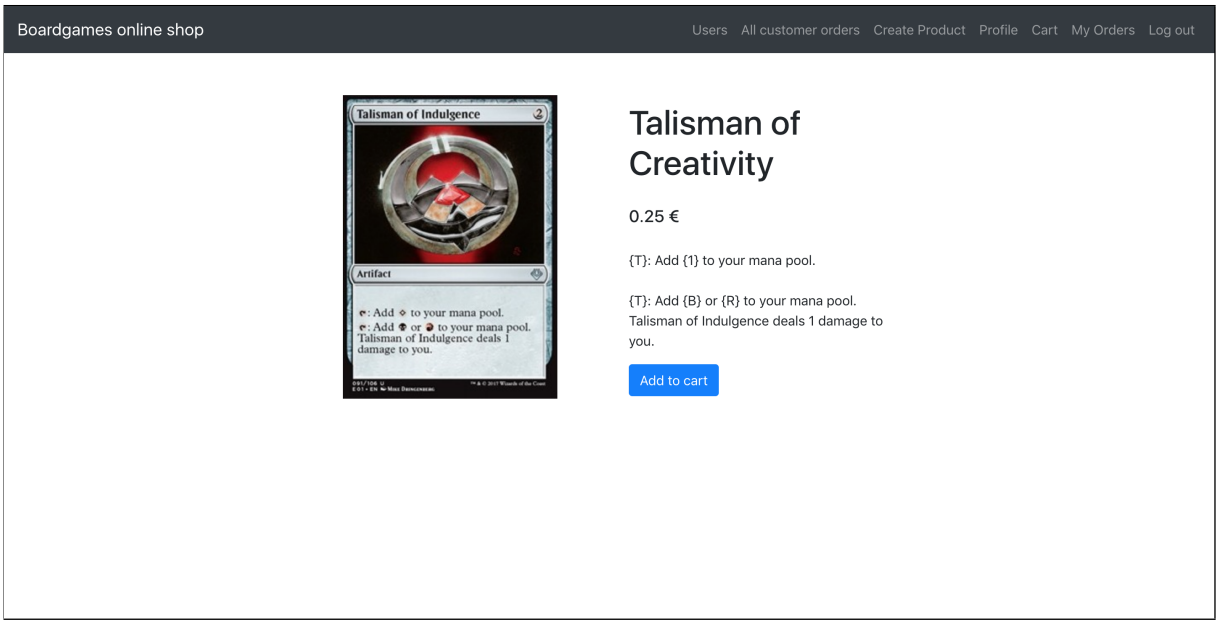


Figura 12: Captura de la pantalla de descripció del producte

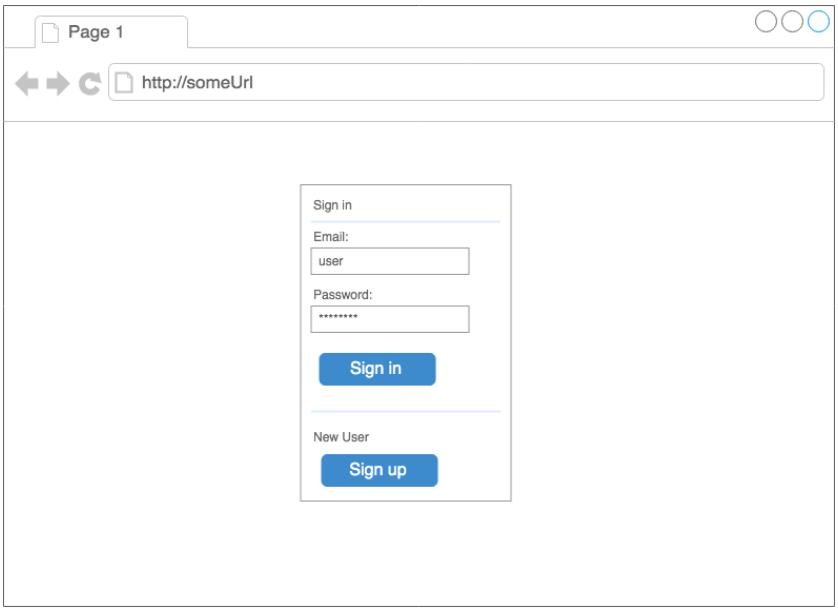


Figura 13: *Mock up* de la pantalla de login

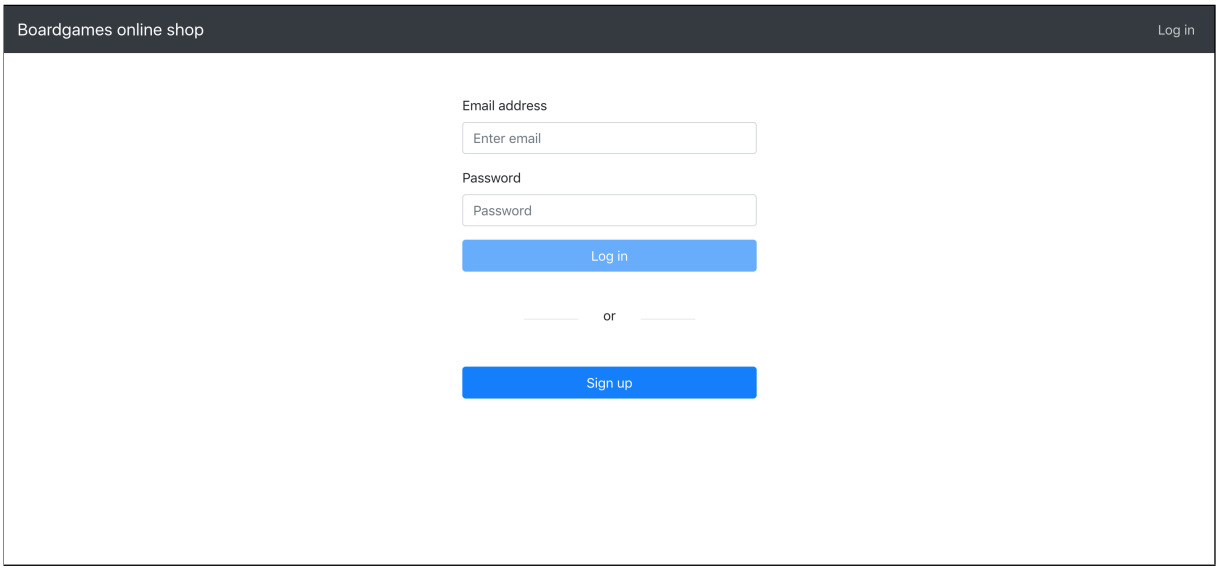


Figura 14: Captura de la pantalla de login

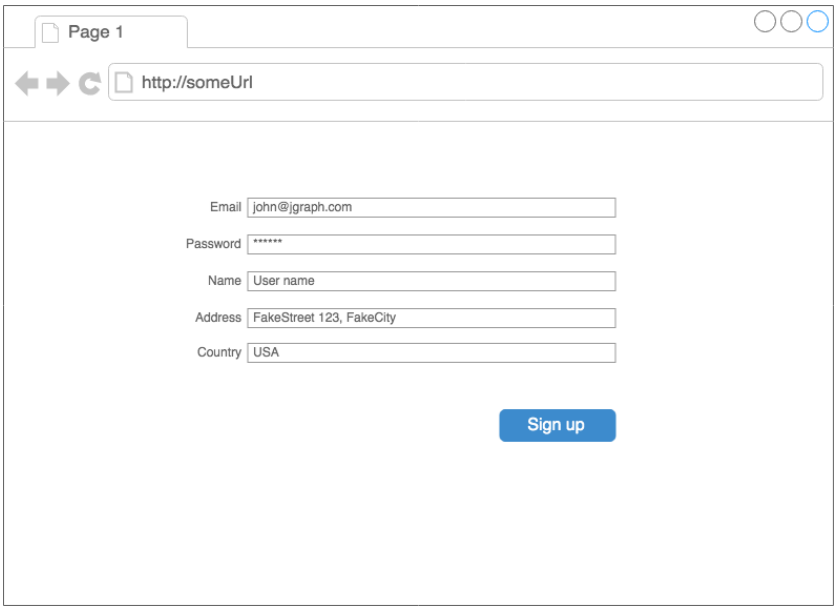


Figura 15: *Mock up* de la pantalla del registre

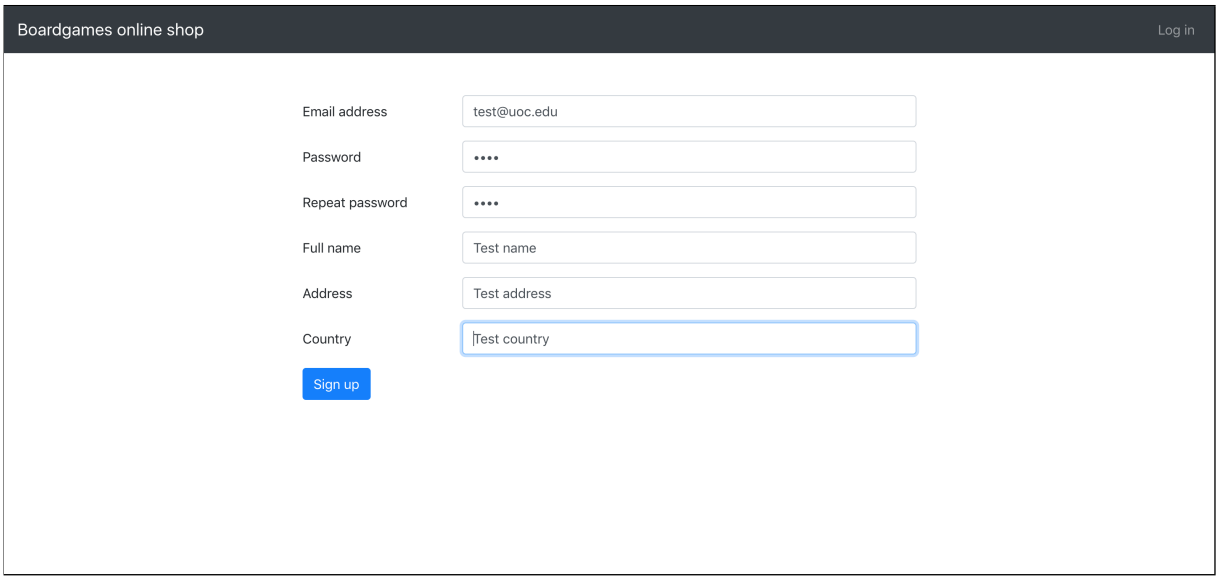


Figura 16: Captura de la pantalla del registre

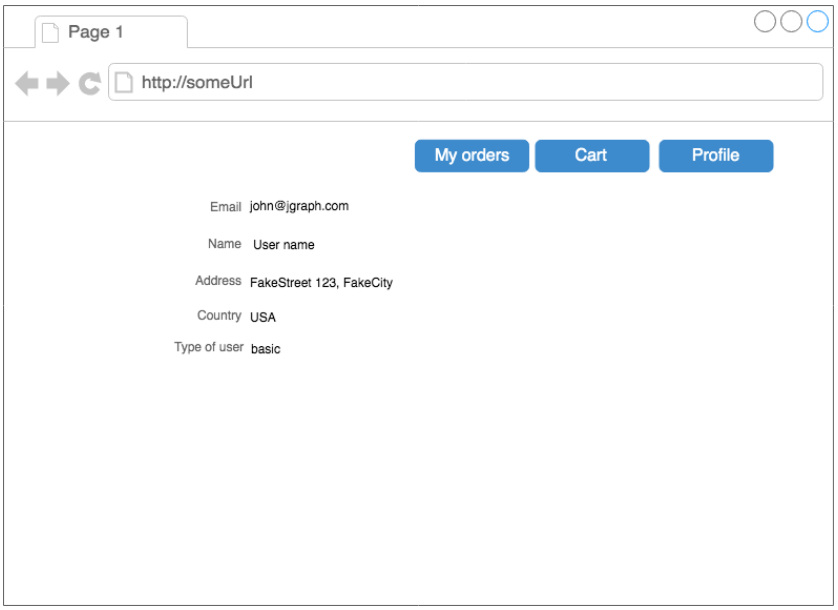


Figura 17: *Mock up* de la pantalla del perfil de l'usuari

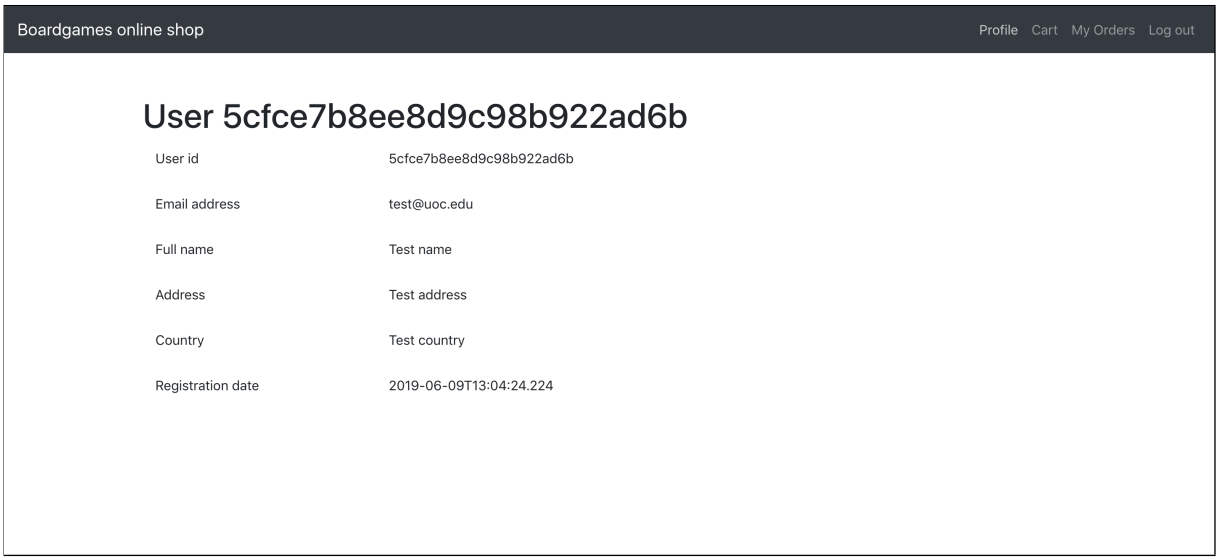


Figura 18: Captura de la pantalla del perfil de l'usuari

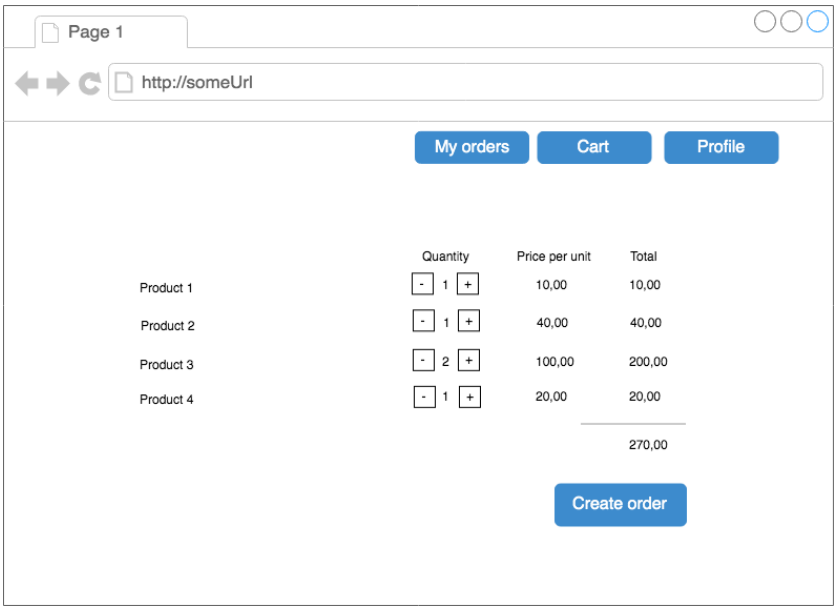


Figura 19: *Mock up* de la pantalla del carret

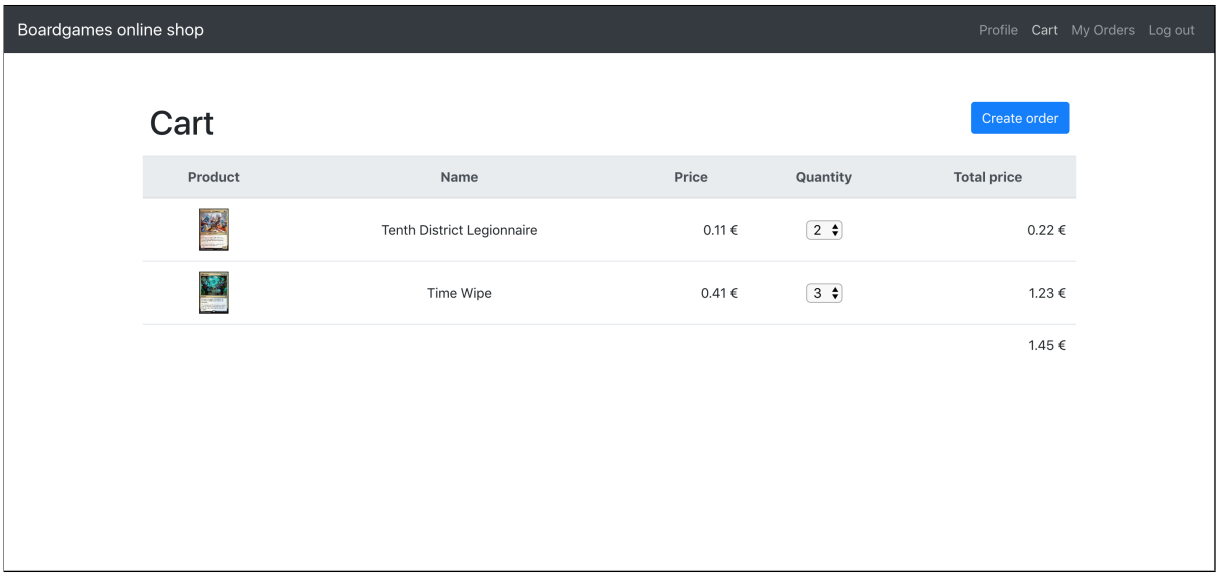


Figura 20: Captura de la pantalla del carret

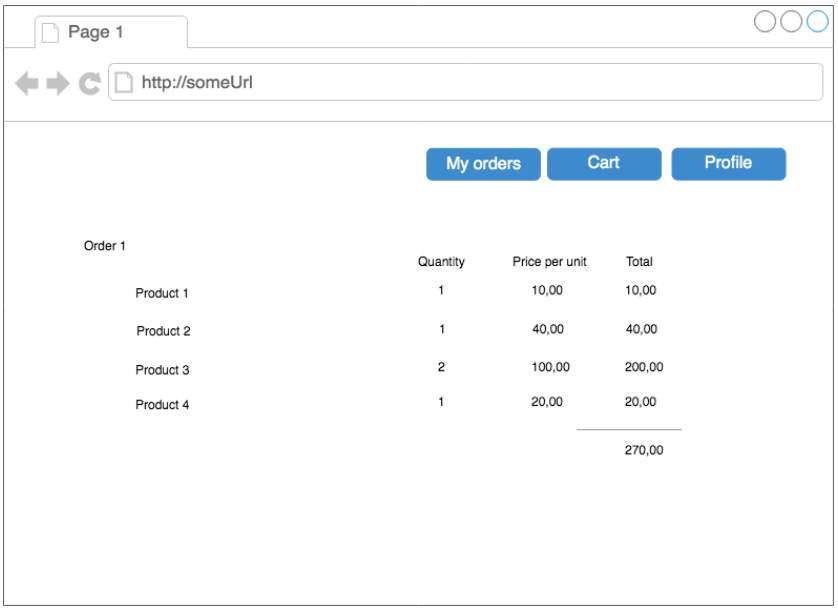


Figura 21: *Mock up* de la pantalla del resum d'una comanda

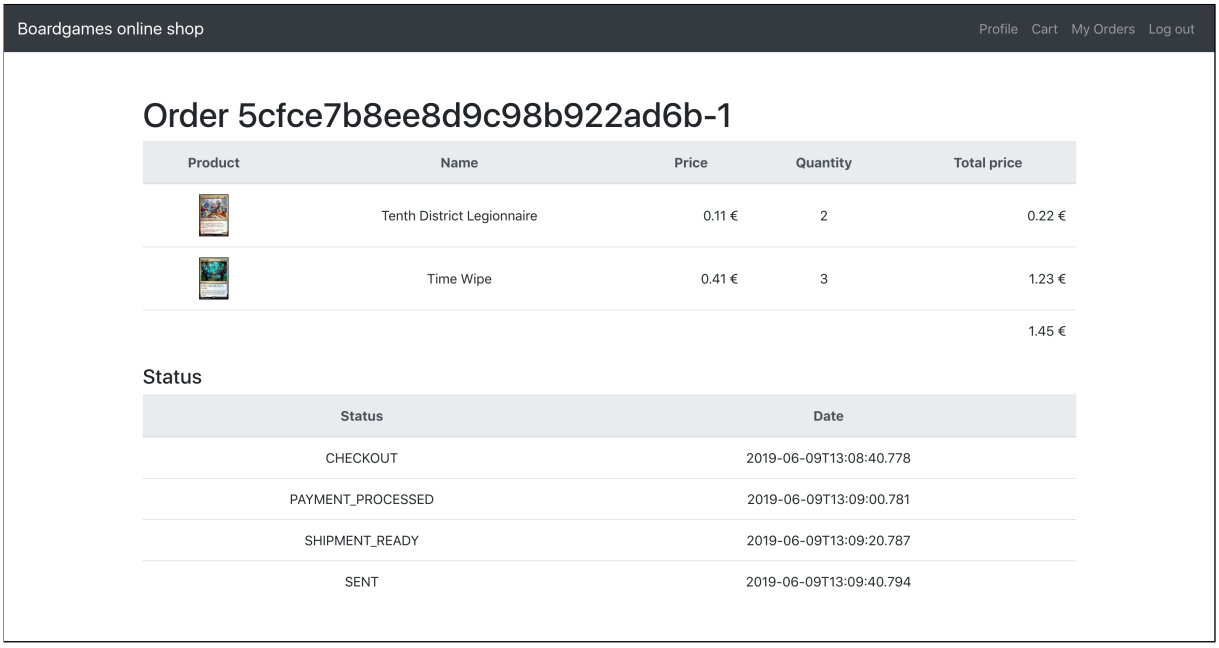


Figura 22: Captura de la pantalla del resum d'una comanda

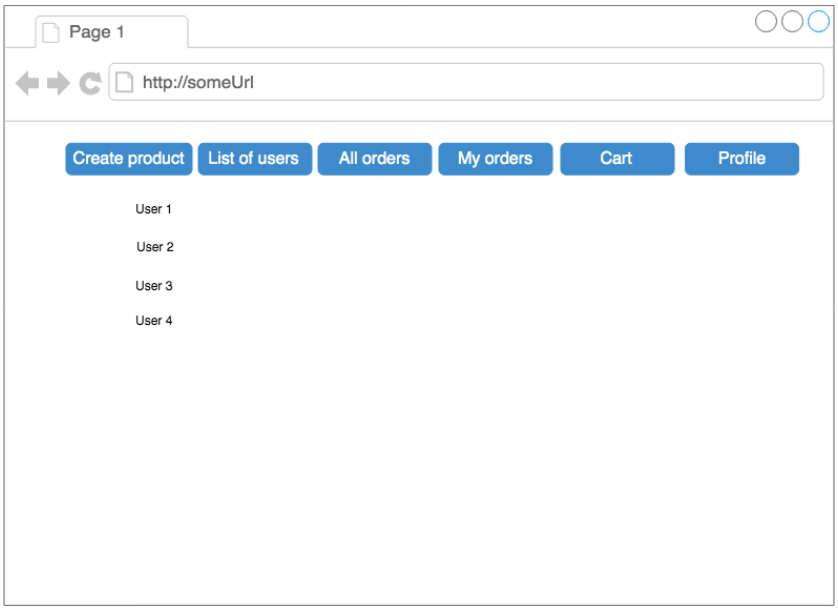


Figura 23: *Mock up* de la pantalla del llistat d'usuaris registrats al sistema

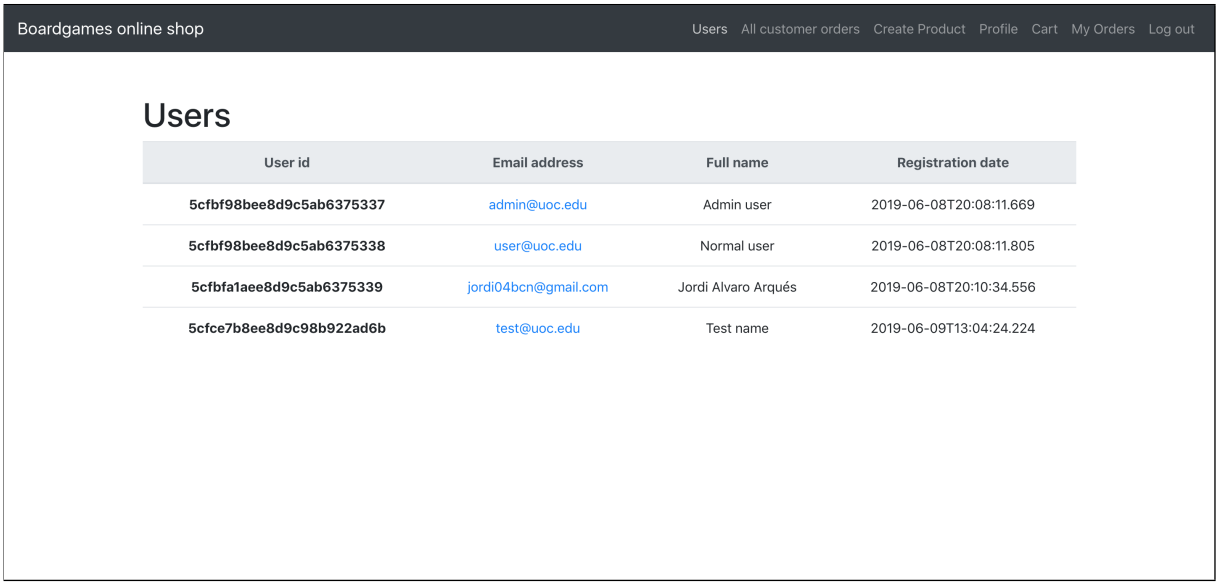


Figura 24: Captura de la pantalla del llistat d'usuaris registrats al sistema

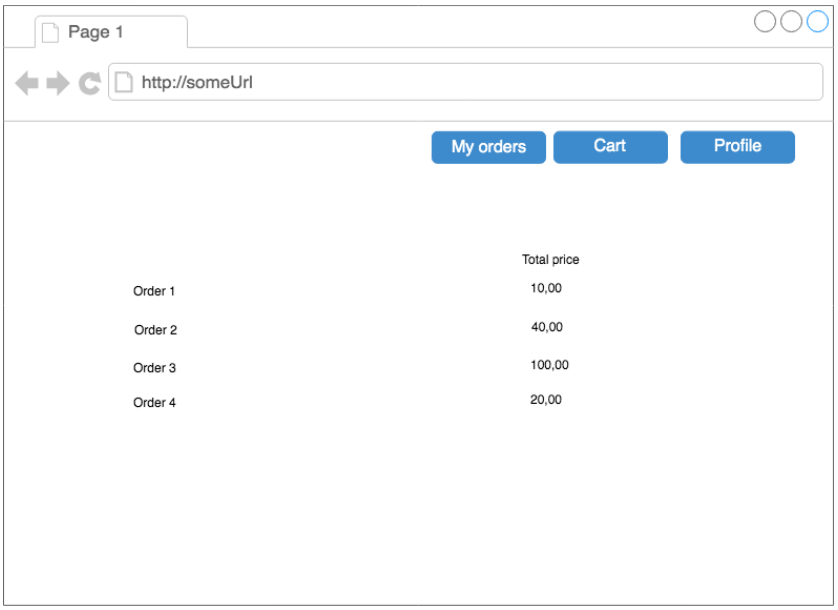


Figura 25: *Mock up* de la pantalla del llistat de comandes realitzades per un usuari

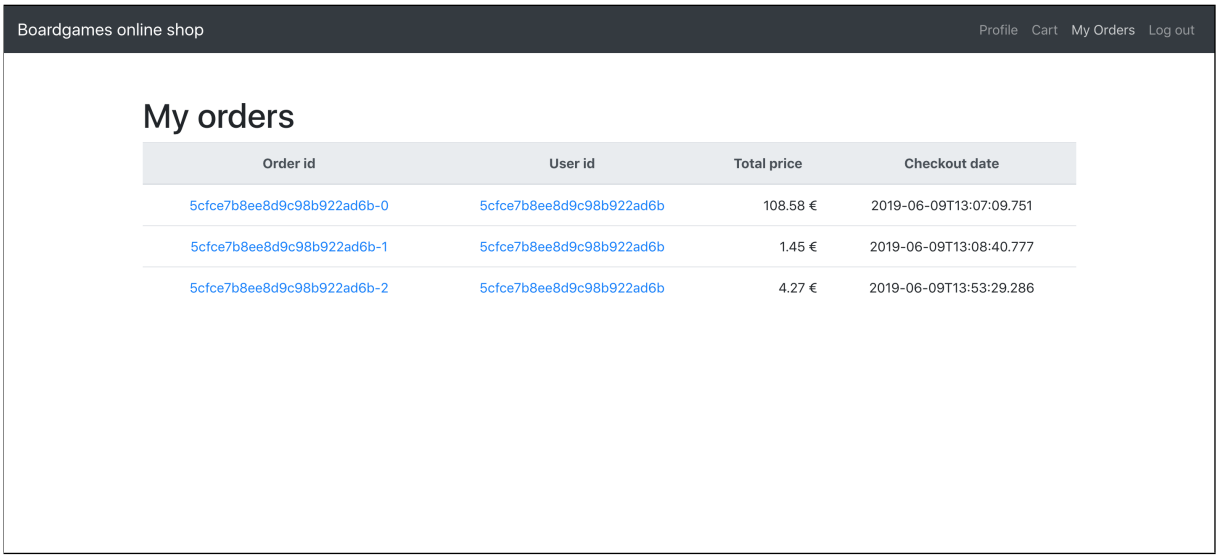


Figura 26: Captura de la pantalla del llistat de comandes realitzades per un usuari

3 Tecnologies utilitzades

En aquesta secció es descriuen les tecnologies, llibreries i *frameworks* utilitzats més rellevants. També es detalla com es configuren i les seves principals característiques i funcionalitats.

3.1 Frontend

Les llibreries ReactJS i Bootstrap 4 s'han escollit per a desenvolupar el microservei que exerceix de frontend del sistema. També s'ha utilitzat la llibreria Redux com a complement de ReactJS.

3.1.1 Bootstrap 4

És una llibreria de components molt popular que simplifica el desenvolupament de les pàgines web. Es basa en CSS 3 i el seu principal objectiu és poder mantenir una consistència i uniformitat en els estils utilitzats a tota l'aplicació web.

Bootstrap defineix un estil bàsic per defecte, però també disposa d'altres modalitats i temes, com per exemple, els temes *dark* i *light*. Configurar Bootstrap amb un altre tema és molt senzill i intuïtiu. No obstant, en aquest projecte se n'ha escollit el que apareix per defecte.

Bootstrap disposa d'una gran varietat de components HTML i classes CSS prèviament definides, com alertes, botons, taules, formularis... Un cop la llibreria és inclosa al projecte només cal utilitzar els estils i els components seguint la documentació i els exemples que mostren a la seva web (https://getbootstrap.com/). A l'algoritme 1 es mostra un exemple de com es definiria una alerta utilitzant Bootstrap.

```
1 <div class="alert alert-success alert-dismissible" role="alert">
2     <div>This is an alert</div>
3     <button type="button" class="close" data-dismiss="alert" aria-label
      ="Close">
4         <span aria-hidden="true">&times;</span>
5     </button>
6 </div>;
```

Algoritme 1: Codi d'un missatge d'alerta de tipus *success*

Figura 27: Representació gràpica del codi de l'algoritme 1

3.1.2 ReactJS

ReactJS és una llibreria JavaScript dissenyada per a desenvolupar interfícies d'usuari i facilitar la creació de components interactius i reutilitzables. Pot ser utilitzada com a base per al desenvolupament d'aplicacions de tipus *single-page* o per a dispositius mòbils. Està optimitzada, a nivell de rendiment i modularitat, per visualitzar ràpidament els canvis que s'hi produeixen.

El desenvolupament d'una aplicació ReactJS es basa principalment en la creació de funcions i components. ReactJS utilitza una extensió de JavaScript anomenada JSX que permet definir funcions Javascript amb una notació molt similar a HTML. ReactJS acopla la lògica de visualització amb la de la interfície d'usuari (*UI*). D'aquesta manera els components o funcions creats, contenen tant la lògica de la seva representació com el tractament de les variables i casuístiques possibles.

Continuant amb l'exemple de l'alerta de Bootstrap, s'ha implementat de la manera següent al codi del microservei del frontend:

```
1  const Alert = ({text, hideAlert, type}) =>
2      <div className={`alert alert-${type} alert-dismissible`} role="
       alert">
3          <div>{text}</div>
4          <button type="button" className="close" data-dismiss="alert"
       aria-label="Close" onClick={hideAlert}>
5              <span aria-hidden="true">&times;</span>
6          </button>
7      </div>;
8
9  const SuccessAlert = ({text, hideAlert}) => <Alert text={text}
       hideAlert={hideAlert} type='success'/>;
10
11 const ErrorAlert = ({text, hideAlert}) => <Alert text={text} hideAlert
       ={hideAlert} type='danger'/>;
```

Algoritme 2: Codi de les alertes a l'aplicació del frontend

Es pot veure com JSX és molt similar a HTML, però permet definir components en funcions JavaScript. D'aquesta manera, permet crear components genèrics que es poden especialitzar en altres que els cridin. Aquests components poden ser de dos tipus:

- Components de funcions (*Function Components*). Consisteix en definir una funció que retorna un contingut de tipus JSX. Es poden definir variables i funcions locals. No pot guardar l'estat. L'algoritme 2 en mostra un exemple.
- Components de classe (*Class Components*). Consisteix en definir una classe que extengui de `React.Component`. Pot guardar l'estat del component i ReactJS proveeix d'un seguit de mètodes per tal de gestionar els canvis i actualitzacions de l'estat. L'algoritme 3 en mostra un exemple.

A l'algoritme 2 s'ha creat un component base i genèric anomenat `Alert`. A aquest component se li passen tres paràmetres:

- `text`. Missatge que apareixerà a l'alerta.
- `hideAlert`. Funció que fa desaparèixer l'alerta de la *UI*.
- `type`. Tipus de l'alerta. Pot ser: `success`, `danger`, `warning`, `info`...

També es pot observar que s'han definit dos funcions més (`SuccessAlert` i `ErrorAlert`) que simplement referencien al component genèric `Alert` fixant el tipus d'alerta i passant els paràmetres `text` i `hideAlert`.

```
1  class OrderEvents extends React.Component {
2      constructor(props) {
3          super(props);
4
5          this.loadOrderEvents();
6      }
7
8      state = {
9          ...
10     };
11
12     loadOrderEvents = () => {...};
13
14     render() {
15         return (
16             <div>
17                 ...
18             </div>
19         );
20     }
21 }
```

Algoritme 3: Codi simplificat a l'aplicació del frontend

A l'algoritme 3 s'observa com l'estructura i la definició del component ha canviat. En aquest cas es tracta d'una classe i es pot definir una variable anomenada `state` que és l'encarregada de gestionar i mantenir l'estat del component. Normalment, la variable `state` es defineix al constructor, però gràcies la utilització de *polyfill* a través d'una dependència de ReactJS es pot definir dirèctament a nivell de classe.

També cal mencionar que l'estat no es pot modificar dirèctament, sinó que s'han d'utilitzar un mètode asíncron (`this.setState()`) per tal d'actualitzar-lo. A més, existeixen altres mètodes per tal de gestionar el cicle de vida d'un component de classe com `componentDidUpdate()` i `componentDidMount()`.

L'aplicació inicial de ReactJS s'ha configurat utilitzant l'eina *Create React App* (https://github.com/facebook/create-react-app) que crea una aplicació ReactJS

funcional de forma molt senzilla i ràpida. Per aconseguir-ho, només cal executar la comanda `npx create-react-app my-app`. A partir d'aquí, s'ha anat modificant i afegint els components, classes i funcions necessàries.

3.1.3 Redux

És una llibreria JavaScript que permet gestionar l'estat a nivell d'aplicació. És molt utilitzada com a complement de ReactJS. ReactJS permet mantenir estat als components i es podria aconseguir gestionar un estat global passant-lo als components niats, però, en funció de l'abast de l'aplicació, en molts casos, es pot convertir en una estructura complexa i difícil de mantenir.

Per tal de mantenir un estat global, Redux es basa en l'ús de tres elements:

- *Store*. És l'element encarregat de mantenir l'estat global. Només se'n defineix un en tota l'aplicació.
- *Actions*. Defineixen les accions que es poden realitzar. L'única manera de modificar l'estat del *Store* és executar una acció en concret. Aquestes accions no modifiquen dirèctament l'estat global, sinó que s'envia un event indicant l'acció a realitzar que es pot acompanyar de la informació que s'ha de mantenir a l'estat global.
- *Reducers*. Són els encarregats de gestionar les accions i actuar en conseqüència. Poden modificar i actualitzar l'estat global.

Per facilitar la lectura i la mantenibilitat de l'aplicació, normalment, les accions i els *reducers* es defineixen en diferents arxius depenent separant-los seguint el Principi de la Responsabilitat Única.

A continuació es pot veure un exemple simplificat del codi utilitzat al frontend del sistema que s'ha desenvolupat.

```
1 const saveSession = sessionCustomer => ({
2     type: 'SAVE_SESSION_CUSTOMER',
3     sessionCustomer
4 });
5
6 const removeSession = () => ({
7     type: 'REMOVE_SESSION_CUSTOMER'
8 });
```

Algoritme 4: Codi de les accions de sessió d'usuari

```
const sessionReducer = (prevState, action) => {
    switch (action.type) {

        case SAVE_SESSION_CUSTOMER: {
            //update state
            ...
        }

        case REMOVE_SESSION_CUSTOMER: {
            //update state
            ...
        }
        default: {
            return prevState;
        }
    }
};
```

Algoritme 5: Codi dels *reducers* de sessió d'usuari

```
const configureStore = () => {
    return createStore(
        appReducer,
        applyMiddleware(thunk)
    );
};
```

Algoritme 6: Codi del *Store*

A part d'aquesta configuració, per tal que els components de ReactJS tinguin accés a aquestes accions i les puguin executar, se'ls ha de connectar amb Redux. Això s'aconsegueix de la següent manera:

```
class LoginComponent extends React.Component {
    ...
}

const mapStateToProps = state => ({
    ...state.sessionReducer
});

const mapDispatchToProps = {saveSession};

const Login = connect(mapStateToProps, mapDispatchToProps)(
    LoginComponent);
```

Algoritme 7: Codi dels *reducers* de sessió d'usuari

Per tant, la classe `LoginComponent` com està connectada amb Redux i se li ha injectat la funció *reducer* `saveSession()` i l'estat `sessionReducer`, ara pot accedir a aquestes variables excutant `this.props.saveSession()` i `this.state.sessionReducer` respectivament.

3.2 Backend

El llenguatge de programació escollit per a desenvolupar els microserveis del Backend ha estat Java 8. Per tal de complementar i facilitar el desenvolupament de les aplicacions, s'ha fet ús del *framework* Spring (versió 5).

3.2.1 Spring

Spring és un *framework* de codi lliure que proporciona un model complet de programació i configuració per a aplicacions empresarials modernes basades en Java. Algunes de les tecnologies en que es basa aquest *framework* són: injecció de dependències, AOP (*Aspect-Oriented Programming*), events, i18n, validació, conversió de tipus...

En concret, en aquest projecte s'ha utilitzat Spring Boot, que és una extensió del *framework* de Spring que elimina gran part de la configuració necessària per a posar en marxa una aplicació Spring. De fet, aquesta configuració la defineix per defecte internament, i pot ser sobreescrita pel desenvolupador en cas de que necessiti uns altres paràmetres.

Algunes de les característiques que incorpora són:

- Dependències `starter` per simplificar el *build* i la configuració de l'aplicació.
- Servidor encastat per tal d'evitar la complexitat del desplegament de l'aplicació. Per defecte, es deplega en un Tomcat.
- Mètriques, *Health check* i configuració externalisada.
- Configuració automàtica per algunes funcionalitats de Spring.

Spring es basa principalment en l'ús de *beans*, que són instàncies úniques d'objectes gestionats per Spring IoC. Habilita la seva creació i configuració a partir d'unes anotacions especials que permeten definir els diferents tipus de components especialitzats (controlador, servei, configuració...). A continuació, es fa un resum de les anotacions més importants i utilitzades en el desenvolupament d'aquest projecte. A part de les que s'anomenen, també existeixen `@Component` i `@Repository`, i són molt utilitzades, però no se n'ha fet ús en aquest projecte.

- `@RestController`. És una anotació que simplifica la creació de serveis *REST*.

  Per tal de determinar els mètodes de tipus GET, POST, etc, existeixen unes altres anotacions que serveixen per aquest propòsit: `@GetMapping`, `@PostMapping`... També existeixen altres anotacions que ens permeten extreure els valors dels paràmetres de la *query* (`@PathVariable`) i del cos de la crida *REST* (`@RequestBody`).

Se'n pot veure un exemple d'utilització a l'Algoritme 8.

```
@RestController
public class CustomersController {
    ...

    @GetMapping("/customers/{customerId}/orders")
    public List<OrderDto> getCustomerOrders(@PathVariable("
    customerId") final String customerId) {
        ...
    }

    @PostMapping("/customers/{customerId}/orders")
    public OrderDto createCustomerOrder(@RequestBody final
    OrderDto orderDto, @PathVariable("customerId") final String
    customerId) {
        ...
    }

    ...
}
```

Algoritme 8: Exemple d'utilització de l'anotació `@RestController` a l'aplicació Orders (arxiu: `CustomersController.java`)

- `@Service`. Serveix per indicar que es tracta d'un *bean* que conté lògica de negoci.

```
@Service
public class OrdersService {
    ...
}
```

Algoritme 9: Exemple d'utilització de l'anotació `@service`

- `@Configuration`. Principalment utilitzada per afegir configuració addicional a l'aplicació. Les classes que contenen aquesta anotació poden contenir mètodes que creein *beans* anotant-los amb `@Bean`.

  També es pot afegir l'anotació `@ConfigurationProperties` a aquestes classes, amb la qual cosa permet inicialitzar-la amb les propietats definides a algun arxiu de configuració.

  Als algoritmes 11 i 10 es pot observar aquest cas.

```
...

initServers:
  - https://api-gateway-ws.herokuapp.com/customers-ws
  - https://api-gateway-ws.herokuapp.com/products-ws
  - https://api-gateway-ws.herokuapp.com/orders-ws
  - https://api-gateway-ws.herokuapp.com/carts-ws


edge-hostname: https://jaarques-uoc.github.io
```

Algoritme 10: Arxiu `application.yml` de l'aplicació Api Gateway

```
@Configuration
@ConfigurationProperties
public class ServersProperties {
    private List<String> initServers;

    private String edgeHostname;
}
```

Algoritme 11: Exemple d'aplicació de `@ConfigurationProperties` a l'arxiu `ServersProperties.java` de l'aplicació Api Gateway

- `@SpringBootApplication`. Aquesta anotació agrupa l'efecte de les tres anotacions següents:
  - `@EnableAutoConfiguration`. Habilita l'auto-configuració del sistema.
  - `@ComponentScan`. Habilita l'escaneig dels components.
  - `@Configuration`. Permet afegir configuració extra a la classe principal de l'aplicació.

  El següent exemple mostra el seu ús:

```
@SpringBootApplication
public class ProductsApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProductsApplication.class, args);
    }
}
```

Algoritme 12: Exemple d'utilització de l'anotació `@SpringBootApplication`

Per crear la base del projecte en Spring s'ha utilitzat la web de *Spring initializer* (https://start.spring.io/) que permet seleccionar el tipus d'eina d'automatització de *builds*, el llenguatge de programació i la seva versió (en aquest cas Java 8), la versió de Spring Boot, metadata del projecte i algunes de les dependències disponibles.

3.2.2 Llibreries incloses a Spring

El *framework* Spring proveeix d'una gran quantitat de llibreries per a facilitar el desenvolupament d'aplicacions. A continuació, es fa un resum de les utilitzades en aquest projecte:

- Spring Boot Actuator. Exposa informació operacional de l'aplicació que s'està executant. Així, permet monitoritzar-la i obtenir-ne mètriques. Per activar-la, només cal afegir la dependència a `build.gradle`. Per defecte, només estan habilitats els *endpoints* `/health` i `/info`.
- Spring Data MongoDB. Permet la integració amb MongoDB i facilita la implementació de les crides que s'han de fer a la base de dades.

  Per utilitzar les funcionalitats d'aquesta llibreria s'han de realitzar quatre passos:

  – Importar la llibreria. Se'n pot veure un exemple a l'algoritme 19.

  – Crear un model que s'utilitzarà per guardar a la BD. Es pot fer ús de l'anotació `@Id` per tal que es generi automàticament un identificador per a la instància del model a emmagatzemar si no en té un d'assignat. Veure algoritme 13.

```
1  public class OrderEvent {
2
3      @Id
4      private String id;
5
6      private String orderId;
7
8      ...
9
10     private OrderStatus status;
11 }
```

Algoritme 13: `OrderEvent.java` de l'aplicació Orders

  – Crear un repository que extengui de `MongoRepository` i definir, en cas que sigui necessari, crides específiques a BD. Per defecte, la interfície `MongoRepository` ja genera bastantes operacions d'accés a BD, d'entre elles les operacions estàndar CRUD (*Create*, *Read*, *Update*, *Delete*). Veure algoritme 14.

```
1  public interface OrderEventsRepository extends MongoRepository
       <OrderEvent, String> {
2      List<OrderEvent> findAllByOrderId(final String orderId);
3  }
```

Algoritme 14: `OrderEventsRepository.java` de l'aplicació Orders

  - Afegir l'anotació `@EnableMongoRepositories` que habilita l'escanneig i reconeixement de les classes que extenen `MongoRepository`. L'anotació es pot afegir a la classe principal de l'aplicació.

- Spring Cloud Netflix: Proveeix les integrations de Netflix OSS per a aplicacions Spring. Inclou el servei de descobriment Eureka, enrutament intel·ligent (Zuul), curt circuit (Hystrix) i balanceig de càrrega a nivell de client (Ribbon). La seva integració amb el projecte s'explicarà amb detall en els apartats d'Api Gateway i Service Discovery.

- Feign Client. Inicialment també formava part de les llibreries de Netflix de Spring, però en canviar a la versió de Spring Boot 2, es va moure a un altre paquet. Permet generar clients de serveis *REST* de forma declarativa.

  - Importar la llibreria.
  - Crear una interfície i afegir-li l'anotació `@FeignClient("feign_name")` a nivell de classe. Per identificar la tipologia dels mètodes a cridar es fan servir les mateixes anotacions que amb els Controladors de Spring: `@RestMapping`, `@GetMapping`, `@PostMapping`...
  - Afegir l'anotació `@EnableFeignClients`. L'anotació es pot afegir a la classe principal de l'aplicació.

```
1 @FeignClient("products-ws")
2 public interface ProductsClient {
3
4     @GetMapping("/products/{productId}")
5     ProductDto getProduct(@PathVariable("productId") final String
      productId);
6
7     @GetMapping("/products")
8     List<ProductDto> getProducts(@RequestParam("ids") final List<
      String> productIds);
9 }
```

Algoritme 15: `ProductsClient.java` extret de l'aplicació Orders

A l'algoritme 19 es pot veure com s'importen la majoria d'aquestes llibreries.

3.2.3 Altres llibreries

En aquest projecte també s'han utilitzat altres llibreries externes al *framework* Spring. Són les següents:

- Project Reactor. És una llibreria que permet crear aplicacions reactives. Ofereix 2 tipus de classes per a aconseguir-ho: Flux (útil quan s'ha de gestionar N events) i Mono (útil quan s'ha de gestionar 0 o 1 event). S'ha utilitzat puntualment al projecte, en particular, quan s'ha fet un *mock* dels següents passos que pateix una comanda un cop és creada. Se'n parlarà en detall quan s'expliqui l'aplicació d'Orders.

- Project Lombok. És una llibreria que permet minimitzar i reduir la gran quantitat de codi repetit que apareix en una aplicació Java. Està principalment enfocat a reduir el codi relatiu a *getters*, *setters* i constructors, tot i que, també, abarca altres àrees.

  Per a la seva utilització, simplement cal importar la llibreria a gradle i fer ús de la gran quantitat d'anotacions de que disposa. Uns exemples són:

  - `@Setter` i `@Getter`. Genera *setters* o *getters* respectivament a les variables objectiu. Pot afegir-se a nivell de classe o a nivell de variable.
  - `@AllArgsConstructor`. Genera un constructor amb un paràmetre per a cada variable que estigui definida a la classe.
  - `@NoArgsConstructor`. Genera un constructor sense paràmetres.
  - `@RequiredArgsConstructor`. Genera un constructor amb un paràmetre per a cada variable que requereixi una gestió especial. Per exemple, per a les variables de classe definides com a `final` s'afegirà un paràmetre al constructor autogenerat. Un cas particular d'aquesta anotació és:
    `@RequiredArgsConstructor(onConstructor = @__(@Autowired))`
    que permet inicialitzar les variables de tipus `final` injectant les instàncies dels *beans* requerits que Spring gestiona de l'aplicació.
  - `@Builder`. Crea les classes necessàries per a poder aplicar el patró de disseny *Builder* i així poder crear instàncies d'una classe d'una manera més elegant i explicativa. Per tal de poder definir valors per defecte, també existeix l'anotació `@Builder.Default`.
  - `@Data`. És una anotació que engloba l'efecte de les següents anotacions pertanyents a Lombok: `@ToString`, `@EqualsAndHashCode`, `@Getter`, `@Setter` i `@RequiredArgsConstructor`.

  Als següents algoritmes es pot observar l'utilització de les anotacions mencionades:

```
1  @Data
2  @Builder
3  @AllArgsConstructor
4  @NoArgsConstructor
5  public class Order {
6
7      private String id;
8
9      private BigDecimal amount;
10
11     @Builder.Default()
12     private LocalDateTime date = LocalDateTime.now();
13
```

```
14     private String customerId;
15
16     private List<OrderItem> orderItems;
17 }
```

Algoritme 16: `Order.java` de l'aplicació Orders

```
1 OrderEvent orderEvent = OrderEvent.builder()
2     .orderId(order.getId())
3     .status(orderStatus)
4     .build();
```

Algoritme 17: Exemple d'utilització del patró *Builder* amb Lombok extret de l'aplicació Orders

- MapStruct. És una llibreria que simplifica la implementació dels *mappings* entre *POJOs* (*Plain Old Java Object*). Al projecte s'usa per transformar els *DTOs* (*Data Trransfer Object*), que s'utilitzen per comunicar-se entre microserveis, a models, que s'utilitzen per emmagatzemar informació a la base de dades. Per utilitzar-la, cal:
  - Importar la llibreria a gradle.
  - Crear una interfície i afegir-li l'anotació `@Mapper`.
  - Crear una instància de mapper a la pròpia interfície.
  - Crear els mètodes que realitzaran les transformacions. Per als tractaments especials de variables, s'ha de definir amb l'anotació `@Mapping` aplicada al mètode en qüestió.

```
1 @Mapper
2 public interface OrderMapper {
3
4     OrderMapper INSTANCE = Mappers.getMapper(OrderMapper.class);
5
6     @Mapping(source = "orderItems", target = "orderItemDtos")
7     OrderDto toOrderDto(Order order);
8
9     @Mapping(target = "date", ignore = true)
10    @Mapping(source = "orderItemDtos", target = "orderItems")
11    Order toOrder(OrderDto orderDto);
12 }
```

Algoritme 18: `OrderMapper.java` de l'aplicació Orders

A l'algoritme 19 també es pot veure com s'importen aquestes 3 llibreries externes.

```
plugins {
    id 'org.springframework.boot' version '2.1.4.RELEASE'
    id 'java'
}

apply plugin: 'io.spring.dependency-management'

group = 'com.jaarquesuoc.shop'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    maven { url 'https://repo.spring.io/milestone' }
    mavenCentral()
}

ext {
    set('springCloudVersion', 'Greenwich.SR1')
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-
    actuator'
    implementation 'org.springframework.boot:spring-boot-starter-data-
    mongodb'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-starter-
    netflix-eureka-client'
    implementation 'org.springframework.cloud:spring-cloud-starter-
    openfeign'
    compile "io.projectreactor:reactor-core:3.3.0.M1"
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-
    test'
    compile 'org.mapstruct:mapstruct:1.3.0.Final'
    annotationProcessor 'org.mapstruct:mapstruct-processor:1.3.0.Final'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-dependencies:$
    {springCloudVersion}"
    }
}
```

Algoritme 19: Exemple de `build.gradle` de l'aplicació Orders

3.3 Base de dades

La base de dades escollida per al desenvolupament d'aquest projecte ha estat de tipus NoSQL i s'ha seguit un patró de disseny de tipus "Event sourcing" simplificat.

Els beneficis d'una base de dades de tipus NoSQL comparada amb una relacional, són els següents:

- Permet l'escalabilitat vertical i horizontal. És a dir, es pot incrementar la RAM, la CPU i altres paràmetres d'una BD (escalat vertical), però també es pot replicar la BD i afegir més instàncies depenent la càrrega i utilització del sistema (escalat horizontal).
- No requereix un esquema, per tant, el manteniment de la BD és molt més simple i aporta molta flexibilitat a l'hora d'afegir noves funcionalitats al sistema.

Tot i que també té alguns desavantatges:

- Les BD NoSQL no tenen suport per a transaccions. La transaccionalitat és una de les característiques principals de les bases de dades relacionals, però les BD NoSQL normalment no ho integren.
- *Eventual consistency*. Aquest model s'utilitza en els sistemes distribuits per tal d'aconseguir una alta disponibilitat del sistema. Assegura que, si no arriben més actualitzacions de les dades, amb el temps, el sistema aconseguirà entrar en un estat de consistència i estabilitat.

Idealment, cada microservei ha de tenir la seva pròpia base de dades, de forma que es pugui escalar i replicar independentment de la càrrega i rendiment dels altres microserveis.

La base de dades de tipus NoSQL escollida, és MongoDB. Cal comentar que, tot i els desavantatges descrits anteriorment, la versió 4.0 de MongoDB dóna suport a transaccions de tipus ACID. De totes formes, al projecte desenvolupat no se n'ha fet ús.

Haver escollit un patró de disseny basat en event sourcing implica que no s'actualitzen ni eliminen registres a la base de dades, només s'afegeixen registres que representen els nous events. El cas més interessant es troba a la BD del carret. Així doncs, cada cop que s'actualitza un producte del carret, es crea i emmagatzema un nou carret amb la nova configuració sense eliminar ni actualitzar l'anterior.

Un altre cas interessant, és el relatiu al processat de les comandes, ja que passen per diversos estats fins que són finalment marcades com a enviades a l'usuari.

La configuració de MongoDB ha estat molt senzilla. En local només cal descarregar l'aplicatiu i executar-lo. A Heroku existeix un *plugin* de connexió amb MongoDB que genera un compte gratuït a https://www.mlab.com que permet que el microservei del backend corresponent pugui connectar-s'hi. Aquest *plugin* defineix una variable d'entorn amb la URL i el port a on s'ha de connectar el microservei.

3.4 Integració Contínua

Quan parlem d'integració contínua, normalment es generalitza el seu significat amb l'intenció de definir aquelles pràctiques modernes de desenvolupament que permeten, d'alguna manera, facilitar ràpidament i de manera automàtica l'actualització de l'aplicació amb els canvis introduïts pel desenvolupador.

Sent estrictes, cal diferenciar entre:

- Integració contínua (*Continuous integration*). Pràctica per la qual els desenvolupadors, quan acaben una funcionalitat nova, afegeixen els seus canvis a un repositori central que executa automàticament uns tests i genera una nova versió de l'aplicació.
- Entrega contínua (*Continuous delivery*). Pràctica que amplia la integració contínua desplegant automàticament els canvis de la nova versió de l'aplicació en un entorn de proves.
- Desplegament continu (*Continuous deployment*). Pràctica que amplia l'entrega contínua desplegant automàticament els canvis de la nova versió de l'aplicació a l'entorn de producció.

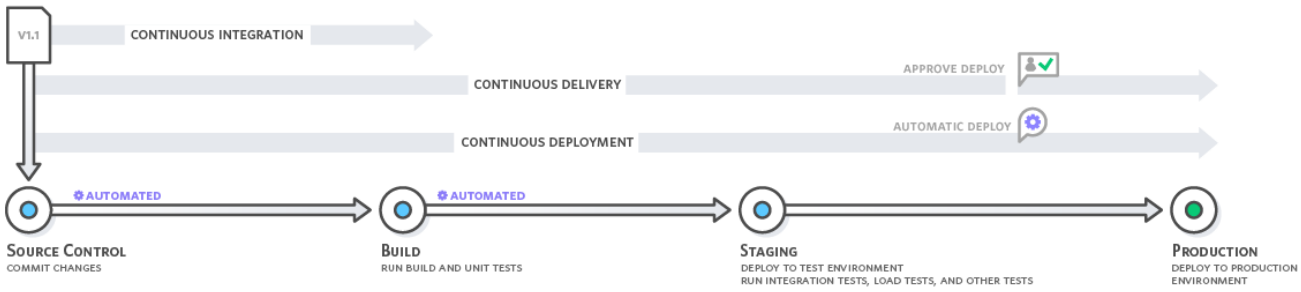


Figura 28: Comparació entre les tres pràctiques.

Per al desenvolupament d'aquest projecte s'ha seguit la tècnica del desplegament continu. Per tal de posar aquesta pràctica en funcionament, s'han utilitzat les següents eines:

- Git. És un sistema de control de versions distribuït de codi. S'ha utilitzat Github com a proveïdor i s'ha creat un repositori per a cada aplicació.
- Docker. Automatitza el desplegament d'aplicacions dins de contenidors de software, proporcionant una capa adicional d'abstracció en relació amb el sistema operatiu on s'executin. S'ha utilitzat DockerHub com a *image registry* (https://hub.docker.com/).
- TravisCI. És un sistema distribuït d'integració contínua que facilita la interconnexió amb GitHub.
- Heroku. És un *platform as a service (PaaS)* que suporta múltiples llenguatges de programació que permet desplegar aplicacions al núvol. També té suport a contenidors Docker.
- GitHub Pages. És un servei d'allotjament de llocs web estàtics que permet desplegar un projecte dirèctament des d'un repositori de GitHub.

A continuació s'explicarà la configuració d'aquestes eines.

3.4.1 Git

S'ha registrat un compte a GitHub (https://github.com/jaarques-uoc) i s'ha donat accés a TravisCI per a que pugui inspeccionar i monitoritzar els diferents repositoris que s'han creat.

Donat que aquest projecte ha estat desenvolupat únicament per una persona, s'ha decidit no seguir una estratègia de branques i s'han realitzat tots els *commits* dirèctament a `master`. No obstant, hi ha hagut un parell d'excepcions on s'han creat branques per tal intentar desenvolupar una funcionalitat específica i que conflictia amb la resta del projecte i microserveis.

De totes formes, en entorns de treball en equip, és molt recomanable el desenvolupament de noves funcionalitats en branques separades i la creació de *Pull Requests* o *Merge Requests* per tal que l'equip pugui revisar la feina feta i donar la seva opinió i detectar possibles errors o punts conflictius.

3.4.2 Docker

S'ha registrat un compte (https://hub.docker.com/u/jaarquesuoc) i s'ha donat accés a TravisCI per a que, cada cop que hagi generat una imatge de docker, l'actualitzi al registre d'imatges del núvol.

No s'ha utilitzat cap tipus de versionat especial i només es manté la última imatge generada amb el tag `latest`.

A continuació, es pot veure la configuració genèrica utilitzada per crear els contenidors Docker de les aplicacions als diferents repositoris de Git. A cada

repositori de les aplicacions Java de Github, s'ha afegit un arxiu `Dockerfile` amb el següent contingut:

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 COPY build/libs/*.jar fat.jar
4 CMD ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/fat.jar"
      ]
```

Algoritme 20: Arxiu Dockerfile

Amb aquest script, bàsicament, es crea una nova imatge amb l'executable de l'aplicació que s'ha desenvolupat. Primerament, s'obté la imatge base de docker amb la distribució Alpine Linux i OpenJDK per a Java 8, s'hi copia l'arxiu amb extensió `.jar` que s'hagi generat i s'especifica la comanda que l'iniciarà quan s'executi aquesta imatge.

3.4.3 TravisCI

S'ha registrat un compte (https://travis-ci.com/jaarques-uoc). A continuació, es pot veure la configuració genèrica utilitzada per a executar els processos d'integració contínua als diferents repositoris de Git. A cada repositori de les aplicacions Java de GitHub, s'ha afegit un arxiu `.travis.yml`.

Cal mencionar que, per tal de poder crear l'arxiu `.travis.yml` de forma genèrica s'han definit unes variables d'entorn per a cada aplicació. A continuació, es pot veure un exemple d'aquestes variables extret de TravisCI:

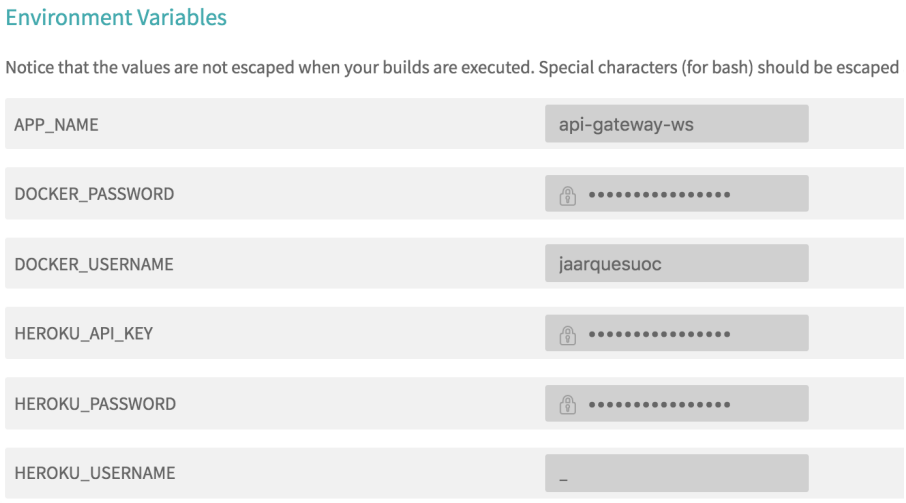


Figura 29: Variables d'entorn configurades a travis per a l'aplicació API Gateway.

```
1  language: java
2  jdk: oraclejdk8
3
4  sudo: required
5  services:
6    - docker
7
8  branches:
9    only:
10     - master
11
12 before_install:
13   # install heroku CLI
14   - wget -qO- https://toolbelt.heroku.com/install.sh | sh
15   # login to docker registries (dockerhub + heroku)
16   - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --
       password-stdin
17   - echo "$HEROKU_PASSWORD" | docker login -u "$HEROKU_USERNAME" --
       password-stdin registry.heroku.com
18
19 script:
20   - docker build --tag $DOCKER_USERNAME/$APP_NAME .
21   - docker tag $DOCKER_USERNAME/$APP_NAME registry.heroku.com/$APP_NAME
       /web
22
23 deploy:
24   provider: script
25   script:
26     # push to dockerhub & heroku
27     docker push $DOCKER_USERNAME/$APP_NAME;
28     docker push registry.heroku.com/$APP_NAME/web;
29     heroku container:release web --app $APP_NAME;
```

Algoritme 21: Arxiu `.travis.yml` per als backends

Entre les línies 1 i 6 es defineix les característiques bàsiques que necessita travis per a executar el procés (Java 8, `sudo` i `docker`). A les línies 8-10, s'especifica que el pipeline només s'executa quan es realitza un *commit* a la branca de `master` del repositori de Git.

De la línea 12 a la 17, es defineix els passos que s'executen abans de la instal·lació: configurar el client de Heroku i iniciar sessió als registres d'imatge de docker (DockerHub i Heroku). Cal comentar que la imatge es carrega als dos repositoris, tot i que la que s'utilitza per al desplegament a producció és la que s'emmagatzema al registre de Heroku. La imatge que es guarda al registre de DockerHub s'utilitza únicament com a còpia de seguretat.

A continuació, entre les línies 19 i 21 es detalla com es crea la imatge de Docker. Finalment, de la línea 23 a la 29, es descriu com es fa el desplegament: es carreguen les imatges als registres de Heroku i DockerHub, i s'executa la comanda de realitzar una *release* utilitzant el client de Heroku prèviament instal·lat.

A partir d'aquest moment, Heroku pren la responsabilitat de desplegar automàticament la nova versió de l'aplicació.

D'altra banda, per a l'aplicació de frontend s'ha utilitzat un altre script per a TravisCI. Aquest és més simple:

```
1  language: node_js
2  node_js:
3    - "stable"
4
5  branches:
6    only:
7      - gh-pages
8
9  script:
10   - git config --global user.name "Jordi"
11   - git config --global user.email "jaarques-uoc@github.com"
12   - git remote rm origin
13   - git remote add origin https://jaarques-uoc:${GITHUB_TOKEN}@github.
       com/jaarques-uoc/jaarques-uoc.github.io.git
14   - CI=false npm run deploy
```

Algoritme 22: Arxiu `.travis.yml` per al frontend

En aquest cas, es pot observar com s'utilitza Nodejs com a llenguatge de programació, s'inicia la sessió amb un usuari conegut a GitHub i, potseriorment, s'executa `npm run deploy`. Aquesta comanda, genera un *build* i si hi ha hagut diferències amb l'anterior, fa `push` a la branca de `master` que és la que GitHub Pages monitoritza i desplega a producció.

3.4.4 Heroku

S'hi ha registrat un parell de comptes, ja que Heroku només facilita registrar fins a un màxim de 5 aplicacions de forma gratuïta a cada compte. En aquest cas se'n necessitaven 6 (`api-gateway-ws`, `eureka-ws`, `customers-ws`, `products-ws`, `carts-ws`, `orders-ws`) per tal de desplegar el sistema complet a producció. Les instàncies d'aquest tipus de compte tenen un aprovisionament màxim de 512Mb de RAM, que tot i ser just per a una aplicació Spring, és suficient per a poder executar el projecte.

Crear una aplicació a Heroku és molt simple, només cal donar-li un nom, que serà l'utilitzat per a exposar-la a Internet, i configurar les variables d'entorn que aquesta aplicació pugui necessitar. Com a exemple, per a l'aplicació de Customers, s'ha escollit el nom `customers-ws` i, per tant, la seva URL d'accés és https://customers-ws.herokuapp.com/. A continuació, es mostra les variables d'entorn per a aquesta aplicació.

Figura 30: Variables d'entorn configurades a Heroku per a l'aplicació Customers.

Tal i com es veurà posteriorment amb més detall, algunes aplicacions (`customers-ws`, `products-ws`, `carts-ws`, `orders-ws`) necessiten accés a la base de dades de MongoDB. Això, a Heroku, s'aconsegueix afegint-hi un *plugin* de connexió i amb la seva variable d'entorn corresponent.

3.4.5 GitHub Pages

S'ha creat un repositori que ha servit per contenir tot el codi de l'aplicació web (frontend). El nom d'aquest repositori ha seguit la convenció necessària per GitHub per tal de reconèixer el repositori que ha de servir com a aplicació web: *username*.github.io.

A continuació es resumeixen els passos realitzats per tal de configurar l'aplicació ReactJS per a que es desplegui a GitHub Pages:

- Crear un repositori amb el nom `jaarques-uoc.github.io`.
- Crear una branca anomenada `gh-pages`.
- Canviar a GitHub la branca per defecte a `gh-pages` en comptes de `master`.
- Fer *push* de l'aplicació ReactJS creada amb *React Create App* a la branca `gh-pages`.
- Afegir la propietat `homepage` a l'arxiu `package.json`:

```
{
  "name": "boardgames-online-shop",
  "version": "0.1.0",
+ "homepage": "https://jaarques-uoc.github.io",
  "private": true,
  "dependencies": {
    "gh-pages": "^2.0.1",
```

Figura 31: Propietat a afegir a `package.json`.

- Afegir a l'arxiu `package.json` una nova comanda de `deploy`. Aquesta comanda permet generar el *build* i fer-ne *push* automàticament a la branca de `master`. La comanda és la següent:

```
  "scripts": {
    "predeploy": "npm run build",
+   "deploy": "gh-pages -b master -d build",
    "start": "react-scripts start",
    "build": "react-scripts build",
```

Figura 32: Comanda a afegir a `package.json`.

- Executar `npm run deploy` per tal d'afegir la primera versió compilada de l'aplicació a la branca de `master`.

Un cop realitzats els passos anteriors, l'aplicació ja està preparada i exposada públicament a Internet a l'enllaç: https://jaarques-uoc.github.io.

4 Arquitectura del sistema

El projecte segueix una arquitectura basada en microserveis. Aquesta estructura ofereix una sèrie d'avantatges respecte altres dissenys tradicionals, com el basat en un monolit. L'arquitectura en microserveis organitza un sistema com un conjunt d'aplicacions o serveis que són altament mantenibles i *testables*, desacoplats, desplegables i escalables de manera independent i organitzats seguint les necessitats de negoci. A més, aporta més agilitat i flexibilitat alhora de dissenyar cada aplicació (es podria escollir el llenguatge i base de dades que millor s'adaptés per a cadascuna), escalabilitat per aplicacions i compleix amb el Principi de Responsabilitat Única.

També, s'ha adaptat i simplificat el disseny basat en *event sourcing*. Teòricament, un disseny d'aquest tipus implica que s'han de persistir tots els events que succeeixin en un sistema de forma que es pugui reproduir l'estat actual a partir de l'històric de tots els events. El patró de disseny event sourcing permet solucionar problemes de transaccionalitat i actualitzacions atòmiques d'entitats o valors. Permet arribar a un estat consistent i estable si no hi ha més actualitzacions de dades al sistema (*eventual consistency*).

La simplificació que s'ha dut a terme ha consistit en no emmagatzemar tots els events, i simplement emmagatzemar el nou estat produït per aquests events, sense modificar l'estat anterior. O sigui, quan el sistema rep un nou event, no actualitza ni elimina registres a la base de dades, però tampoc hi guarda aquests events dirèctament, sinó que els processa, obté el nou estat resultant i els afegeix a la base de dades.

A més, s'ha decidit utilitzar una solució reactiva en el cas particular del processament de la comanda, ja que així s'alliberen recursos del sistema (*threads*) i s'aconsegueix una disponibilitat i capacitat més alta.

Un altre aspecte on s'ha fet molt d'èmfasi ha estat en la qualitat, legibilitat i mantenibilitat del codi, ja que s'ha aplicat els principis *SOLID* durant la implementació de tot el projecte.

Per altra banda, s'ha decidit afegir un parell d'elements extres al sistema en sí. Aquests elements són l'*API Gateway* i el *Service Discovery*. Inicialment no estaven planificats per al desenvolupament d'aquest projecte, però finalment s'han decidit afegir ja que són una part fonamental de qualsevol sistema dissenyat seguint una arquitectura en microserveis.

L'estructura en microserveis del sistema és la següent:

- *Frontend Service*. Conté el codi de l'aplicació web estàtica.
- *API Gateway*. És l'accés des de l'exterior als serveis interns del sistema. Enruta les crides del frontend cap als serveis corresponents.

- *Service Discovery*. S'encarrega de mantenir un registre dels serveis i instàncies que es troben funcionant del sistema. Informa de la *URL* i port on es troben els diferents serveis.
- *Products Service*. Conté la lògica referent a la gestió del catàleg de productes i l'inventari.
- *Customers Service*. Conté la lògica de sessió i la gestió del perfil d'un usuari registrat.
- *Carts Service*. Conté la lògica referent a la gestió del carret de la compra d'un usuari registrat.
- *Orders Service*. Conté la lògica referent a la gestió de les comandes.

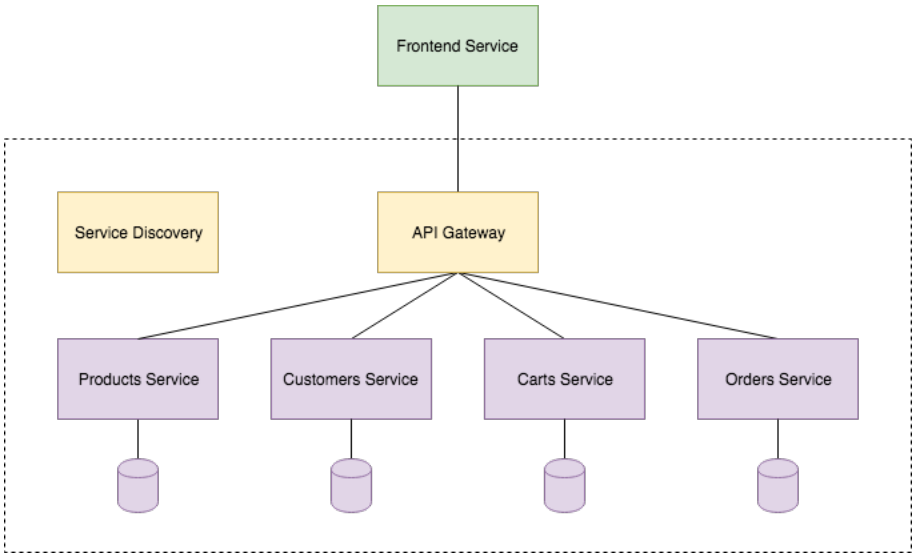


Figura 33: Disseny del sistema de la botiga online seguint una arquitectura de microserveis.

A les següents seccions es descriuran en detall els aspectes de la implementació dels diferents serveis i les decisions de disseny preses.

4.1 *Frontend Service*

Es tracta d'una aplicació *frontend* estàtica desenvolupada en ReactJS i utilitzant la llibreria Redux per a mantenir un estat global. Està desplegada a GitHub Pages. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/jaarques-uoc.github.io.

El codi s'organitza de la següent manera:

- El directori arrel de l'aplicació conté els arxius relacionats amb la creació del *build* i el procés d'integració contínua. Els arxius més importants són: `.travis.yml` i `package.json`. També conté l'arxiu `README.md` on es documenta la informació més destacada de l'aplicació.
- `/src`. Conté el codi JS de l'aplicació. S'estructura en directoris en funció de les funcionalitats del sistema:
  - El directori base conté els arxius Javascript relacionats amb l'aplicació a nivell general.
  - A `commons` s'hi ha afegit els components comuns i reutilitzables
  - A `routes` s'hi ha inclós les funcionalitats del sistema (carret de la compra, comandes, gestió de la sessió dels usuaris...).

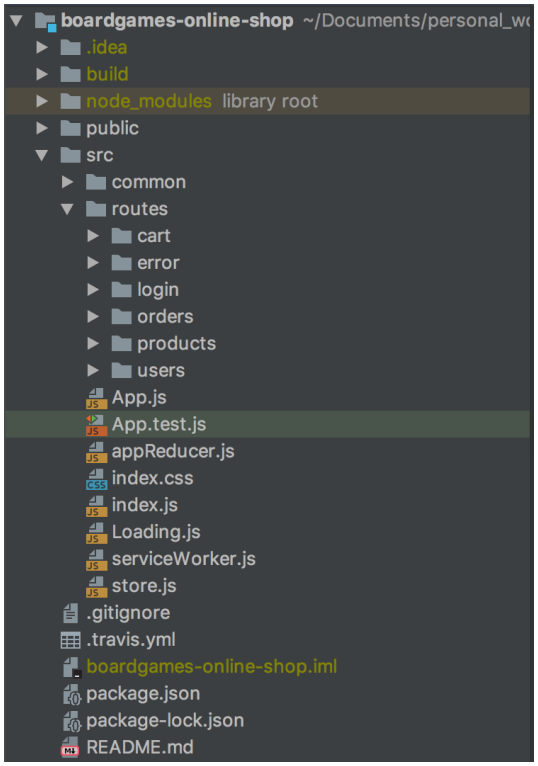


Figura 34: Estructura del codi de l'aplicació reactJS

La nomenclatura que s'ha seguit per a anomenar els arxius JS és la següent:

- *Pascal case* (exemple: `AllOrders.js`). S'ha utilitzat per anomenar arxius que contenen i exposen components (de classe o de funció) de ReactJS.

- *Camel case* (exemple: `emailValidation.js`). S'ha utilitzat per anomenar arxius que contenen i exposen únicament funcions JavaScript (no són estrictament components de ReactJS). Hi ha dos casos particulars que segueixen una nomenclatura especial:
  - Arxiu *DAO* (*Data Access Object*) (exemple: `ordersDAO.js`). Són arxius de connexió amb el *backend*. Gestionen les crides *REST* que es realitzen als *endpoints* per obtenir informació.
  - *Action* o *Reducer* de Redux (exemple: `sessionReducer.js`). Són arxius que contenen les accions o la gestió d'aquests events per part de Redux.

Un punt important del frontend és la gestió de la sessió. Quan l'usuari inicia la sessió i s'autentica correctament, s'executa una acció de Redux (`"SAVE_SESSION_CUSTOMER"`) i el *reducer* corresponent guarda la sessió a l'estat global de Redux i al *Session Storage* del navegador per a que, en cas que tanqui el navegador, li mantingui la sessió en futures visites a la pàgina web.

Aquesta sessió conté informació del rol de l'usuari (`CUSTOMER` o `ADMIN`) i, segons quins privilegis tingui, se li mostraran uns enllaços o uns altres a la barra de navegació. Les imatges 36 i 37 mostra aquesta barra per als casos comentats anteriorment i els algoritmes 23 i 24 mostren les parts més representatives del component `Navbar` amb la comprovació dels rols.

Figura 35: Barra de navegació per a un usuari anònim

Figura 36: Barra de navegació per a un usuari amb sessió i rol `CUSTOMER`

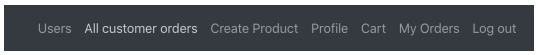


Figura 37: Barra de navegació per a un usuari amb sessió i rol `ADMIN`

```
const NavbarComponent = ({sessionCustomer}) =>
    <nav className="navbar navbar-expand-md navbar-dark bg-dark">
        ...
    </nav>;


const mapStateToProps = state => ({
    ...state.sessionReducer
});

const Navbar = connect(mapStateToProps)(NavbarComponent);

export {Navbar};
```

Algoritme 23: Implementació de la barra de navegació (arxiu: `Navbar.js`)

```
{sessionCustomer.type === 'ADMIN' &&
<React.Fragment>
    <li className="nav-item">
        <Link className="nav-link" to="/users">Users</Link>
    </li>
    ...
</React.Fragment>
}
{isLoggedIn(sessionCustomer) &&
<React.Fragment>
    <li className="nav-item">
        <Link className="nav-link" to={`/users/${sessionCustomer.id}`}>
    Profile</Link>
    </li>
    ...
</React.Fragment>
}
{!isLoggedIn(sessionCustomer) &&
<li className="nav-item">
    <Link className="nav-link" to="/login">Log in</Link>
</li>
}
```

Algoritme 24: Comprovació dels rols de la barra de navegació (arxiu: `Navbar.js`)

4.2 Parts comunes de les aplicacions de *backend*

Donat que les aplicacions de *backend* estan desenvolupades amb Java 8 i Spring, totes elles seguiran una mateixa arquitectura interna i estructura de codi.

4.2.1 Estructura del codi del projecte

El codi s'organitza seguint el patró de disseny d'aplicacions Java:

- El directori arrel de l'aplicació conté els arxius relacionats amb la creació del *build* i el procés d'integració contínua. Els arxius més importants són:

`.travis.yml`, `build.gradle` i `Dockerfile`. També conté l'arxiu `README.md` on es documenta la informació més destacada de l'aplicació.

- `/src/main/java`. Conté el codi Java de l'aplicació. S'estructura en els següents possibles directoris (no totes les aplicacions tenen les mateixes necessitats): `configurations`, `controllers`, `dtos`, `exceptions`, `mappers`, `models`, `repositories`, `services`.

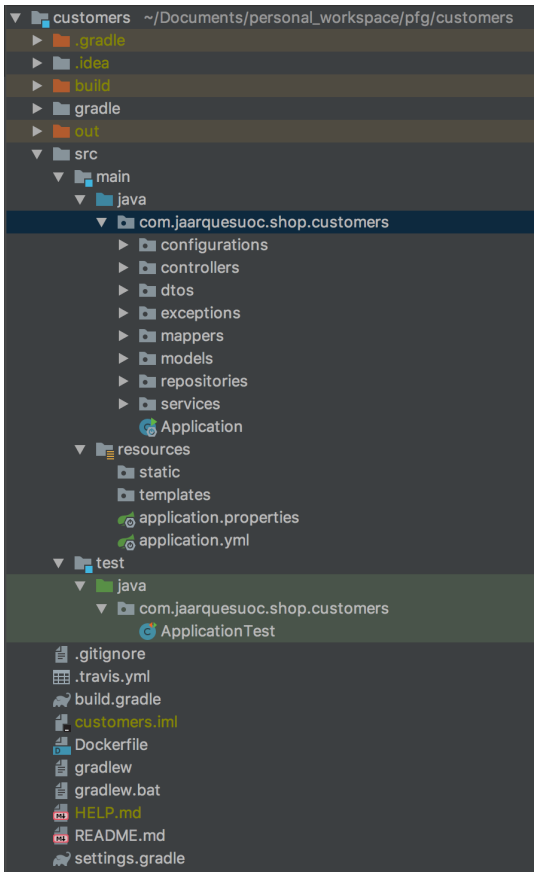


Figura 38: Estructura del codi del servei d'ususaris.

- `/src/main/resources`. Conté els arxius de propietats de l'aplicació: `application.yml` amb les propietats de producció, i en algun cas s'ha fet ús de l'arxiu `application-local.yml` per tal de sobreescriure les propietats per a desplegaments en local.
- `/src/main/test`. Conté les classes de test Java. Només s'ha creat una classe de test que comprova que l'aplicació s'inicia correctament. En un

desenvolupament amb més recursos i temps s'hauria de forçar als desenvolupadors a crear, com a mínim, una batèria de tests unitaris i d'integració per tal que la cobertura del codi fos elevada.

4.2.2 Arquitectura de l'aplicació

Tal i com s'ha vist en apartats anteriors, una aplicació Spring està formada per diversos components. Si se'n vol representar els més importants en un diagrama de capes, s'obté la figura 39.

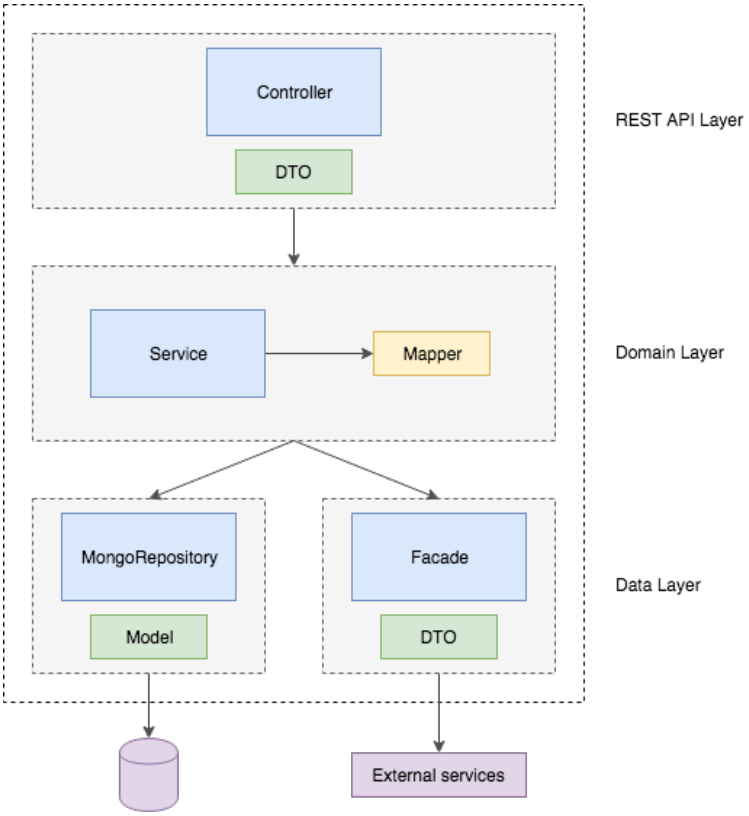


Figura 39: Disseny de les capes d'una aplicació de backend.

- A la capa de la *REST API* hi trobem els controladors, que són els responsables d'exposar els *endpoints* per a que altres aplicacions hi puguin accedir. Aquesta capa utilitza *DTOs* (*Data Transfer objects*) per a comunicar-se externament.
- A la capa de domini, s'hi troben els serveis que contenen la lògica de negoci. En aquesta capa es treballa tant amb els *DTOs* com amb els

models interns que s'utilitzaran per a emmagatzemar informació a la base de dades. Es defineixen uns *mappers* que s'encarreguen de transformar els *DTOs* a models i al revés.

- A la capa de dades, s'hi troben els repositoris encarregats de comunicar-se amb la base de dades Mongo, i els *facades* (juntament amb els seus clients) que s'utilitzen per a comunicar-se amb altres microserveis del sistema.

4.3 *API Gateway Service*

Un *API Gateway* (a vegades, també anomenat *Reverse Proxy*) és un servei que actua com a punt d'entrada a un sistema de microserveis des de la xarxa pública d'Internet. Juntament amb el frontend estàtic, és l'únic servei que ha d'estar exposat públicament a Internet. Tots els altres serveis del sistema, han d'estar protegits darrera d'aquest *API Gateway* i no s'hi ha de poder accedir directament. A més, aquesta aplicació s'encarrega d'enrutar les crides de tipus *REST* cap al microservei adient i pot realitzar tasques de composició de crides i traducció de protocols de comunicació. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/api-gateway-ws.

Cal mencionar que com en aquest projecte les aplicacions de backend s'han desplegat a Heroku, totes elles estan exposades directament a Internet ja que Heroku els proporciona una *URL* d'accés públic. No obstant, per a l'estudi i desenvolupament d'aquest projecte s'ha obviat aquest fet i s'ha actuat com si es trobessin desplegades en una xarxa privada sense accés des de l'exterior.

Per tal de configurar l'aplicació Spring com a *Reverse Proxy*, s'ha fet ús de la llibreria d'enrutament intel·ligent Zuul. Simplement s'ha utilitzat l'anotació `@EnableZuulProxy`, la qual configura l'aplicació Spring per redireccionar les crides que li arriben. Les crides que arriben a l'aplicació són del següent estil: `https://api-gateway-ws.herokuapp.com/{service}/{endpoint}`. Zuul reconeix el patró de la crida i la redirigeix a la *URL* del servei `{service}` mantenint l'*endpoint* `{endpoint}`.

A part, també ha calgut definir alguns *timeouts* (Hystrix, Ribbon i Feign *timeouts*) per tal que les crides no finalitzessin abans d'hora ja que, a vegades, les instàncies de Heroku triguen una mica més a respondre.

En aquest projecte, l'*API Gateway* realitza les següents tasques:

- Enrutament. L'aplicació ReactJS que composa el frontend es comunica única i exclusivament amb el servei *API Gateway*. Aquesta passarel·la és l'encarregada de redirigir les crides cap al microservei corresponent. Retorna l'agregació de les dades i l'estat d'inicialització de cada microservei.
- Composició. S'han preparat dos *endpoints* que actuen sobre la resta de microserveis:

- `/init`. Inicialitza tot el sistema amb dades de prova i elimina les dades que s'hi hagin creat anteriorment. Tots els serveis de backend disposen d'aquest *endpoint*
- `/system/health`. Comprova l'estat del sistema. Aquest *endpoint* és molt útil ja que Heroku apaga les instàncies de les aplicacions al cap d'una estona de no detectar-hi activitat. Aquest *endpoint* retorna l'agregació dels estats de totes les instàncies. Implicitament, aquesta crida també serveix per posar-les en marxa, ja que quan s'intenta accedir a una instància de Heroku que estigui apagada, la posa en funcionament.

Per tal que l'aplicació ReactJS es pugui comunicar sense problemes amb l'*API Gateway*, s'ha hagut de permetre *Cross-Origin Resource Sharing (CORS)* des de la *URL* del frontend (https://jaarques-uoc.github.io). Per raons de seguretat, les aplicacions Spring tenen l'opció de permetre *CORS* deshabilitada. A l'algoritme 25 es pot observar la classe Java de configuració utilitzada per habilitar-lo.

```
1  @Configuration
2  @RequiredArgsConstructor(onConstructor = @__(@Autowired))
3  public class CorsConfigurer implements WebMvcConfigurer {
4
5      private final ServersProperties serversProperties;
6
7      @Override
8      public void addCorsMappings(CorsRegistry registry) {
9          registry.addMapping("/**")
10             .allowedOrigins(serversProperties.getEdgeHostname());
11     }
12 }
```

Algoritme 25: Arxiu `CorsConfigurer.java` de l'*API Gateway*

Per tal de desenvolupar les funcionalitats anteriors (els dos *endpoints* i habilitar *CORS*) s'ha definit diverses classes Java de controladors, propietats i configuració. També s'ha definit unes propietats a l'arxiu `application.yml` per tal de fer-les fàcilment modificables.

Donat que totes les aplicacions de backend utilitzen els mateixos *endpoints* d'inicialització i de comprovació de l'estat de la instància i retornen el mateix tipus de *DTO (Data Transfer Object)*, s'ha creat una interfície genèrica amb l'anotació `FeignClient` però sense definir la *URL* del servei destinatari. Així, l'aplicació obté la instància del client genèric i per a cada servei de backend existent hi inserta la seva *URL*. A l'algoritme 27 es pot observar el client genèric i el 26 mostra el servei de Spring que crida recursivament a tots els servidors que estan definits a l'arxiu de propietats. Un altre punt important a veure és que s'ha fet ús de la llibreria Spring Reactor per tal de paral·lelitzar les crides als diferents backends i així reduir la latència de la resposta.

```
1 @Service
2 @RequiredArgsConstructor(onConstructor = @__(@Autowired))
3 public class InitialisationService {
4
5     private final InitialisationClient initialisationClient;
6
7     private final ServersProperties serversProperties;
8
9     private final Scheduler scheduler = Schedulers.newParallel("
      initScheduler", 10);
10
11     public List<InitialisationDto> initialiseSystem() {
12         return initialiseParallelServices(serversProperties.
      getInitServers())
13                 .collectList()
14                 .block();
15     }
16
17     private Flux<InitialisationDto> initialiseParallelServices(final
      List<String> serverUrls) {
18         return Flux.fromIterable(serverUrls)
19                 .flatMap(serverUrl -> Mono.defer(() -> Mono.just(
      initialiseService(serverUrl)))
20                         .subscribeOn(scheduler));
21     }
22
23     private InitialisationDto initialiseService(final String url) {
24         InitialisationDto initialisationDto;
25
26         try {
27             initialisationDto = initialisationClient.initialise(URI.
      create(url));
28             initialisationDto.setUrl(url);
29         } catch (Exception e) {
30             initialisationDto = InitialisationDto.builder()
31                 .url(url)
32                 .initialisationStatus(KO)
33                 .build();
34         }
35
36         return initialisationDto;
37     }
38 }
```

Algoritme 26: Arxiu `InitialisationService.java` de l'*API Gateway*

```
1 @FeignClient(name = "initialisation", url = "https://this-is-a-
      placeholder.com")
2 public interface InitialisationClient {
3
4     @GetMapping("/init")
5     InitialisationDto initialise(URI uri);
6 }
```

Algoritme 27: Arxiu `InitialisationClient.java` de l'*API Gateway*

4.4 *Service Discovery*

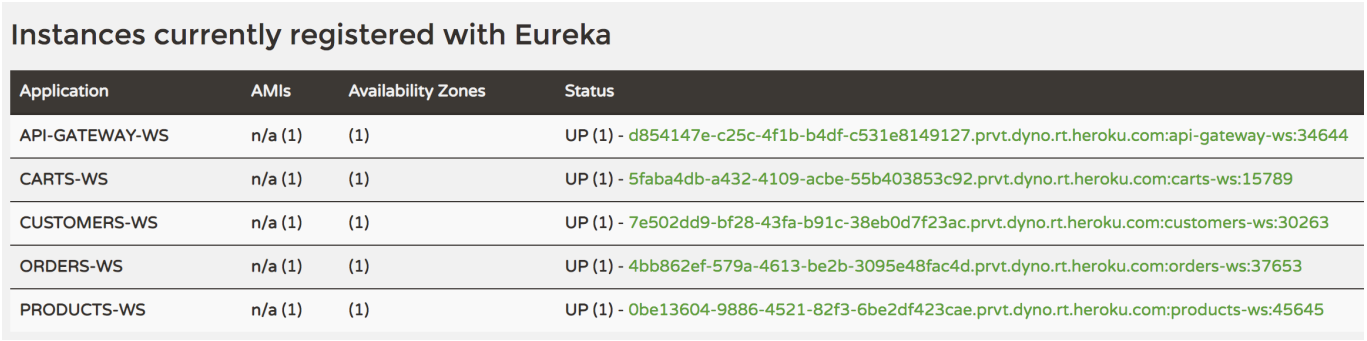
Aquesta aplicació proporciona un servei de descobriment i registre d'aplicacions al sistema de microserveis. Facilita la comunicació i configuració de les diferents aplicacions. En un entorn de microserveis amb múltiples instàncies per aplicació i/o desplegament *BLUE-GREEN* (tècnica de desplegament utilitzada en entorns de producció on es vol tenir una alta disponibilitat), no és pràctic configurar prèviament les *URLs* o *IPs* de les aplicacions. En aquest sentit, el *Service Discovery* simplifica aquesta tasca i permet conèixer en tot moment la *URL* correcta d'un servei. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/eureka-ws.

Bàsicament, el seu funcionament és el següent:

- L'aplicació que realitza el *Service Discovery* s'inicia.
- Els microserveis que es van iniciant i que sí que coneixen la *URL* de l'aplicació de *Service Discovery*, hi contacten i s'hi registren.
- En el moment en que una de les aplicacions del sistema (`A`) es vol comunicar amb una altra (`B`):
  - `A` contacta amb l'aplicació de *Service Discovery* i li demana la *URL* de `B`.
  - Aquesta aplicació li retorna la *URL* de `B`
  - `A` realitza la crida a `B`

Per configurar tot aquest sistema de descobriment de serveis, s'ha utilitzat la llibreria Eureka de Spring que en proporciona tota la implementació. Cal diferenciar dos tipus de configuracions:

- Servidor Eureka (aplicació que gestiona el *Service Discovery*). Per activar-lo s'ha de fer ús de l'anotació `@EnableEurekaServer` i afegir uns paràmetres de configuració a l'arxiu `application.yml`.
- Clients Eureka (microserveis del sistema: *Api Gateway, Customers, Products, Carts* i *Orders Service*). Per activar-lo s'ha de fer ús de l'anotació `@EnableDiscoveryClient` i afegir la la *URL* del servidor Eureka a la propietat `eureka.client.serviceUrl.defaultZone`.



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY-WS	n/a (1)	(1)	UP (1) - d854147e-c25c-4f1b-b4df-c531e8149127.prvt.dyno.rt.heroku.com:api-gateway-ws:34644
CARTS-WS	n/a (1)	(1)	UP (1) - 5faba4db-a432-4109-acbe-55b403853c92.prvt.dyno.rt.heroku.com:carts-ws:15789
CUSTOMERS-WS	n/a (1)	(1)	UP (1) - 7e502dd9-bf28-43fa-b91c-38eb0d7f23ac.prvt.dyno.rt.heroku.com:customers-ws:30263
ORDERS-WS	n/a (1)	(1)	UP (1) - 4bb862ef-579a-4613-be2b-3095e48fac4d.prvt.dyno.rt.heroku.com:orders-ws:37653
PRODUCTS-WS	n/a (1)	(1)	UP (1) - 0be13604-9886-4521-82f3-6be2df423cae.prvt.dyno.rt.heroku.com:products-ws:45645

Figura 40: Exemple de les aplicacions registrades a Eureka.

4.5 *Products Service*

Conté la lògica referent a la gestió del catàleg de productes. S'ha suposat que es disposa d'inventari il·limitat per a tots els productes. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/products-ws.

Els *endpoints* exposats per aquest servei són els següents:

- `GET /init`. Esborra tots els productes existents i inicialitza la base dades amb productes per defecte.
- `GET /products/{id}`. Retorna la informació del producte amb l'identificador `{id}`. Si no el troba retorna un error 404 (*NOT FOUND*).
- `GET /products`. Retorna la informació de tots els productes existents a la base de dades. També permet passar un *query parameter* anomenat `ids` amb una llista d'identificadors dels productes a consultar. En aquest cas només retorna la informació d'aquest subconjunt de productes.
- `POST /products`. Crea el producte a la base de dades que se li passa pel *body* de la crida.

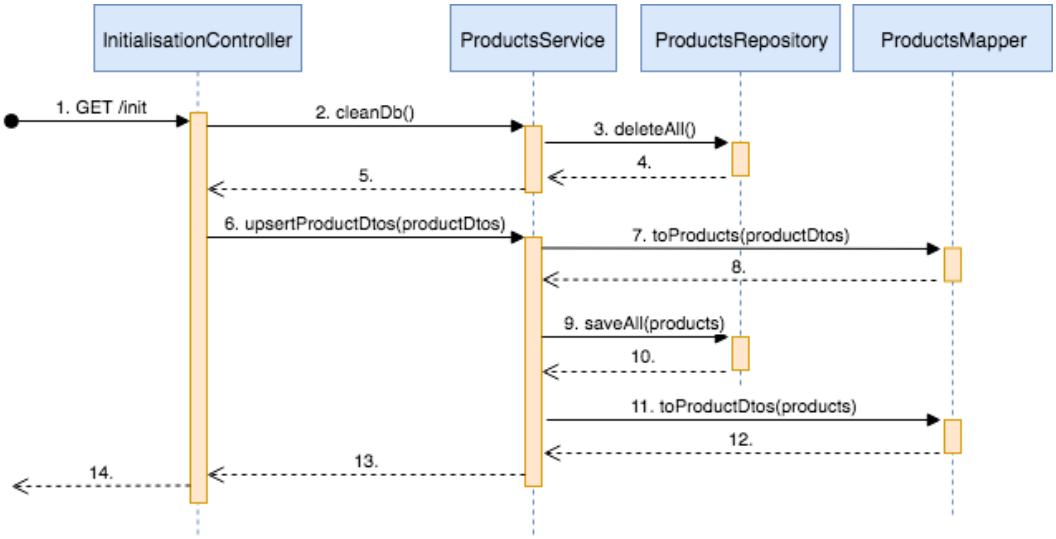


Figura 41: Diagrama de seqüència d'una crida `GET /init`

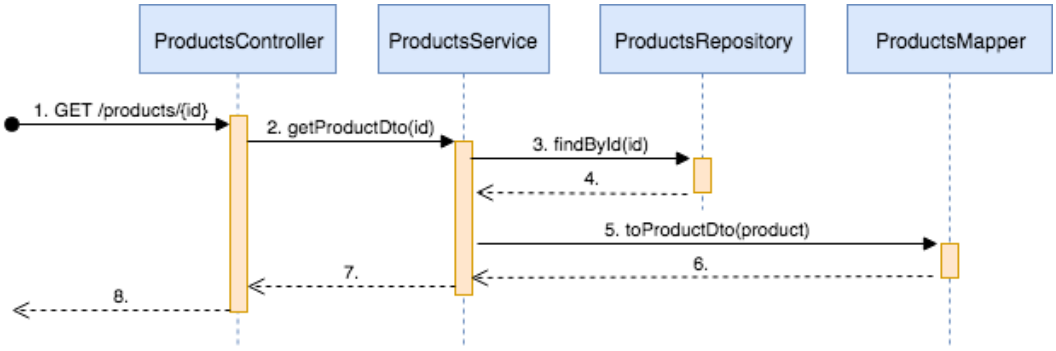


Figura 42: Diagrama de seqüència d'una crida `GET /products/{id}`

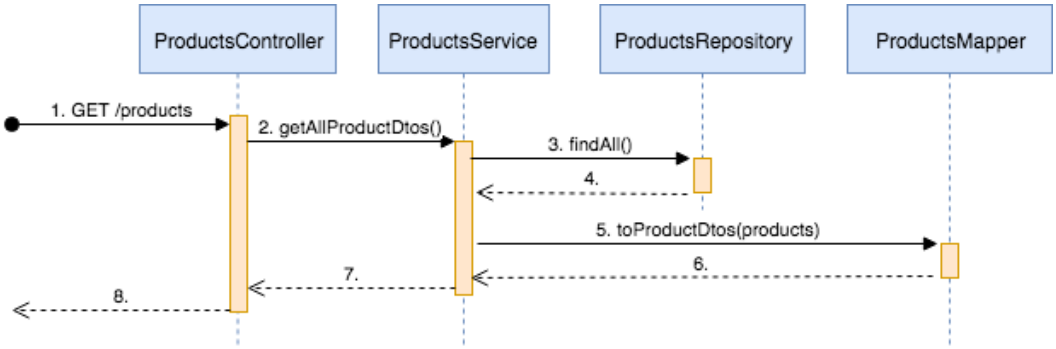


Figura 43: Diagrama de seqüència d'una crida `GET /products`

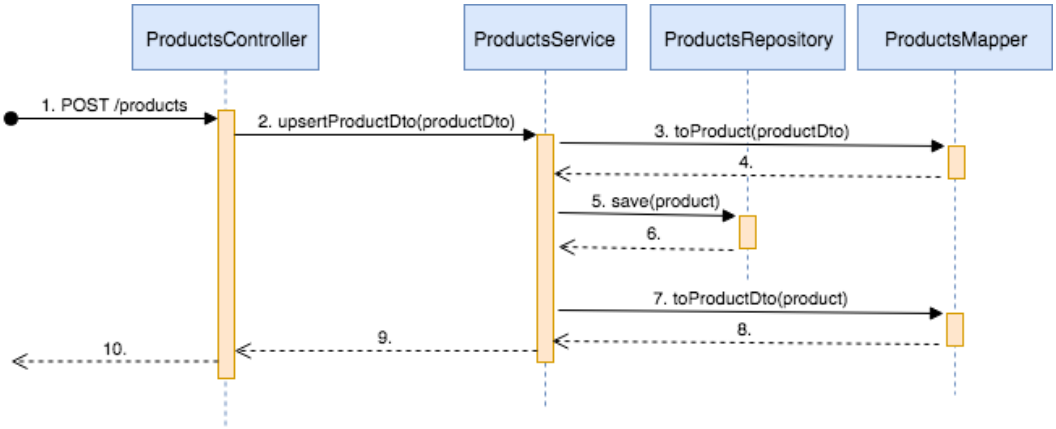


Figura 44: Diagrama de seqüència d'una crida `POST /products`

4.6 *Customers Service*

Conté la lògica de la gestió de la sessió i el perfil d'un usuari. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/customers-ws.

Els *endpoints* exposats per aquest servei són els següents:

- `GET /init`. Esborra tots els usuaris existents i inicialitza la base dades amb usuaris per defecte.
- `GET /customers/{id}`. Retorna la informació de l'usuari amb l'identificador `{id}`. Si no el troba retorna un error 404 (*NOT FOUND*).
- `GET /customers`. Retorna la informació de tots els usuaris existents a la base de dades.

- `POST /login`. Inicia una nova sessió per a l'usuari que se li passa pel *body* de la crida. Si l'email i la contrassenya no coincideixen amb els que hi ha a la base de dades retorna un 401 (*UNAUTHORISED*).
- `POST /signup`. Crea un nou usuari i inicia una nova sessió per a l'usuari que se li passa pel *body* de la crida. Si l'email ja està registrat a la base de dades retorna un error.

Cal comentar que les contrassenyes s'han encryptat a la base de dades utilitzant la classe `BCryptPasswordEncoder.java` de Spring. Aquest encriptador genera un salt per a augmentar la seguretat dels valors guardats a la base de dades i evitar atacs de diccionari. A més, sempre es comparen les contrassenyes encriptades i mai es retorna el seu valor a través de les crides *REST*.

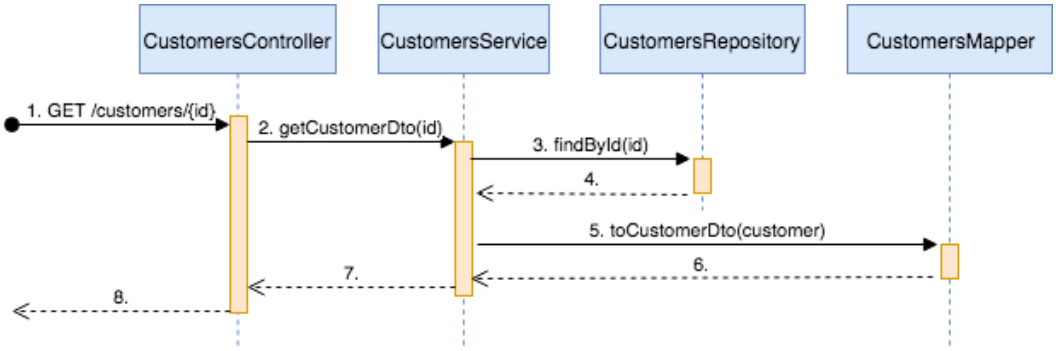


Figura 45: Diagrama de seqüència d'una crida `GET /customers/{id}`

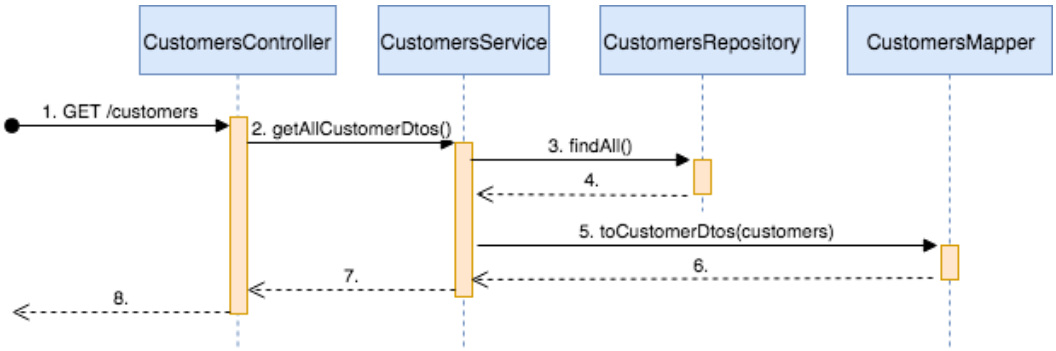


Figura 46: Diagrama de seqüència d'una crida `GET /customers`

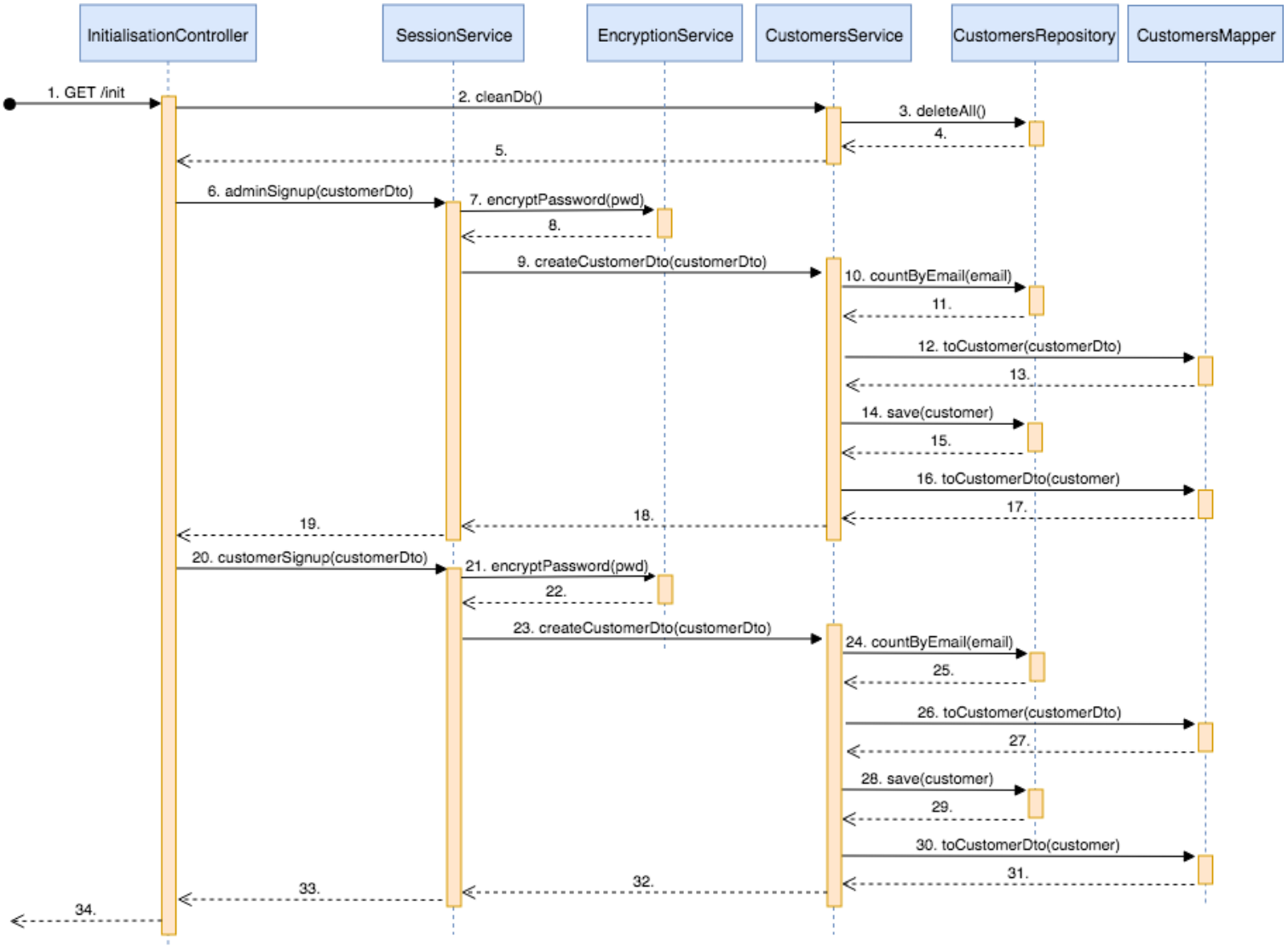


Figura 47: Diagrama de seqüència d'una crida `GET /init`

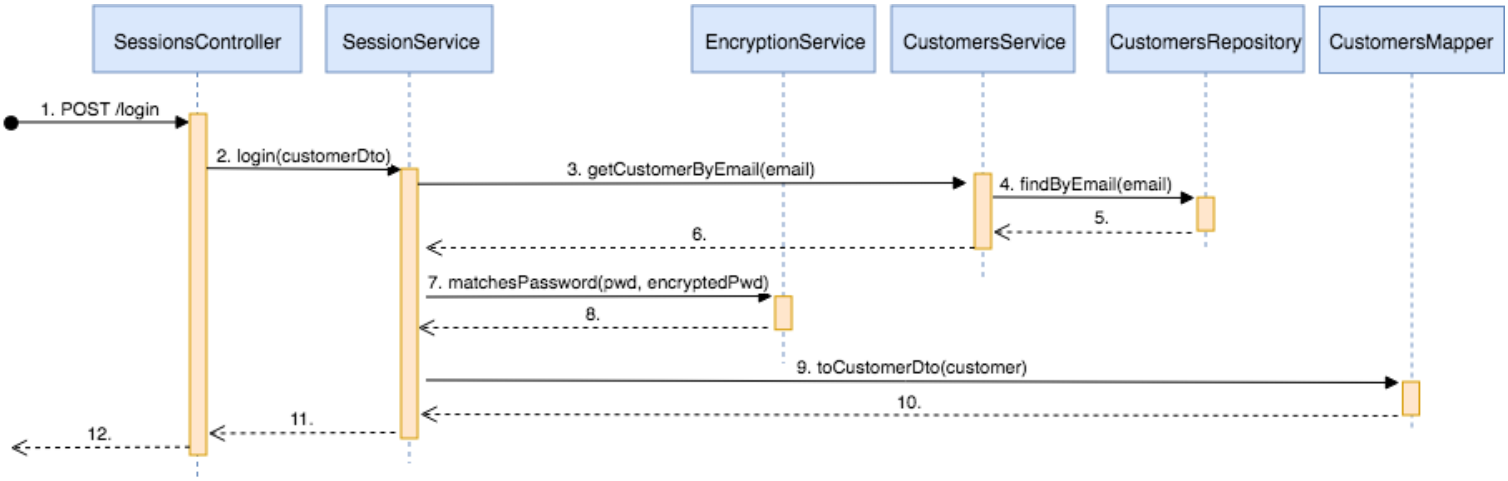


Figura 48: Diagrama de seqüència d'una crida `POST /login`

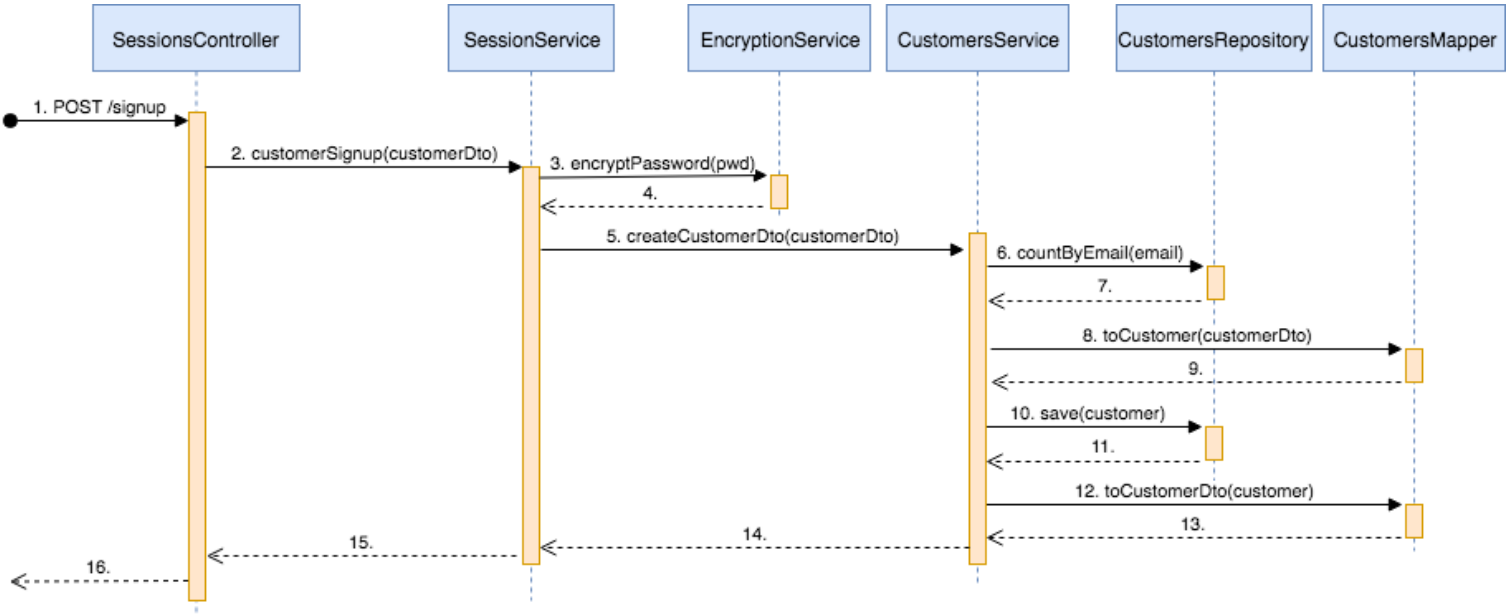


Figura 49: Diagrama de seqüència d'una crida `POST /signup`

4.7 *Carts Service*

Aquesta aplicació conté la lògica de la gestió del carret i procés de *checkout*. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/carts-ws.

Els *endpoints* exposats per aquest servei són els següents:

- `GET /init`. Esborra tots els carrets existents. No s'inicialitza la base dades amb cap carret per defecte.
- `GET /customers/{customerId}/carts/current`. Retorna la informació del carret actual de l'usuari amb l'identificador `{customerId}`.
- `POST /customers/{customerId}/carts/current/items`. Afegeix un element al carret que té un producte i una quantitat associats (aquesta informació s'envia al *body* de la crida *REST*).
- `POST /customers/{customerId}/carts/current/items/increment`. Incrementa en 1 la quantitat d'un producte al carret (aquesta informació s'envia al *body* de la crida *REST*).
- `POST /customers/{customerId}/carts/current/checkout`. Realitza el procés de compra del carret. No s'envia res al *body* ja que la informació del carret està tota emmagatzemada al microservei *Carts Service*.

A continuació, es comenta amb una mica més de detall les funcionalitats d'obtenir el carret i afegir o incrementar la quantitat dels productes. El cas del *checkout* no es detalla ja que utilitza com a base els mateixos conceptes i idees que les funcionalitats d'afegir o incrementar la quantitat de productes al carret.

- Obtenir la informació del carret de la compra. L'aplicació *frontend* realitza una crida de tipus `GET` al microservei del carret. Aquest microservei conté els identificadors de producte i el nombre d'elements que hi ha al carret, però no té la informació detallada del producte.

  D'aquesta manera, el microservei que gestiona el carret de la compra ha de fer una crida de tipus `GET` al microservei dels productes per tal que li retorni la informació detallada dels productes demanats.

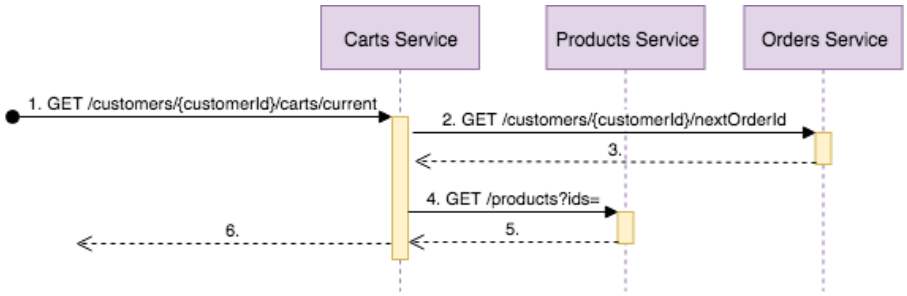


Figura 50: Diagrama de seqüència d'obtenir la informació del carret de la compra a nivell de microserveis.

- Afegir o incrementar la quantitat d'un producte al carret. L'aplicació *frontend* realitza una crida de tipus `POST` al microservei del carret. Aquest microservei ha d'afegir o incrementar la quantitat del producte que se li passa al carret actual de l'usuari.

  Com que es segueix un patró d'event sourcing simplificat, no s'eliminen ni es modifiquen els registres a la base de dades, només se n'afegeixen. Llavors, per tal de gestionar l'històric dels carrets, s'ha decidit utilitzar el sistema següent: se'ls assigna un identificador que correspon a l'identificador de la següent futura comanda per a aquest usuari. Aquest identificador serà una combinació de l'identificador d'usuari amb el nombre de comandes realitzades per aquest usuari (`{customerId}-{nOrders}`).

  Per exemple, un usuari amb identificador `10000` i que hagi realitzat `50` comandes, la següent comanda tindrà l'identificador: `10000-50`.

  D'aquesta manera, *Carts Service* ha de fer una crida `GET` a *Orders Service* per tal que li indiqui l'identificador de la futura comanda. Això és necessari ja que un cop un usuari decideix crear una comanda i aquesta es fa eficaç, el sistema ha de saber reconèixer que el carret que tenia fins a aquell moment és antic i no l'ha d'utilitzar. Per exemple, en el cas de l'usuari anterior, amb un carret amb identificator `10000-50`, quan creï la comanda i torni a intentar accedir al carret, el sistema cercarà a base de dades informació sobre el carret amb identificador `10000-51`, no en trobarà cap i retornarà correctament un carret buit.

  Finalment, en cas que hi hagi més d'un carret a base de dades amb el mateix identificador de carret, s'ordenen els carrets obtinguts per data i se'n retorna l'últim que s'ha creat.

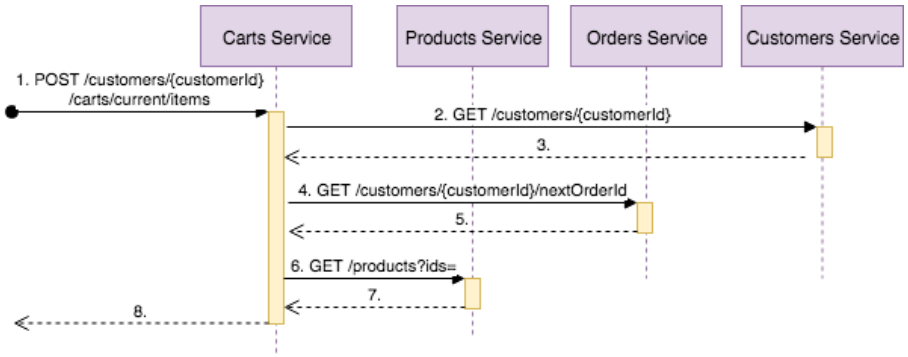


Figura 51: Diagrama de seqüència d'afegir un producte al carret de la compra a nivell de microserveis.

Cal esmentar que els diagrames de seqüència mostrats a les figures 50 i 51 són a nivell d'aplicació i comunicació entre microserveis. No entren en el detall intern de la implementació de *Carts Service*. En les figures següents, es detalla el flux intern d'aquesta aplicació per als diferents *endpoints* que té habilitats.

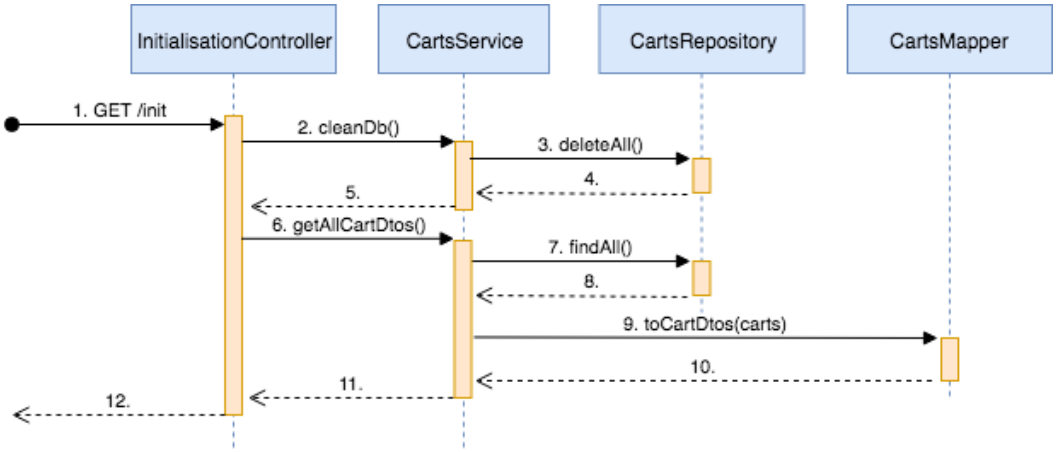


Figura 52: Diagrama de seqüència d'una crida `GET /init`

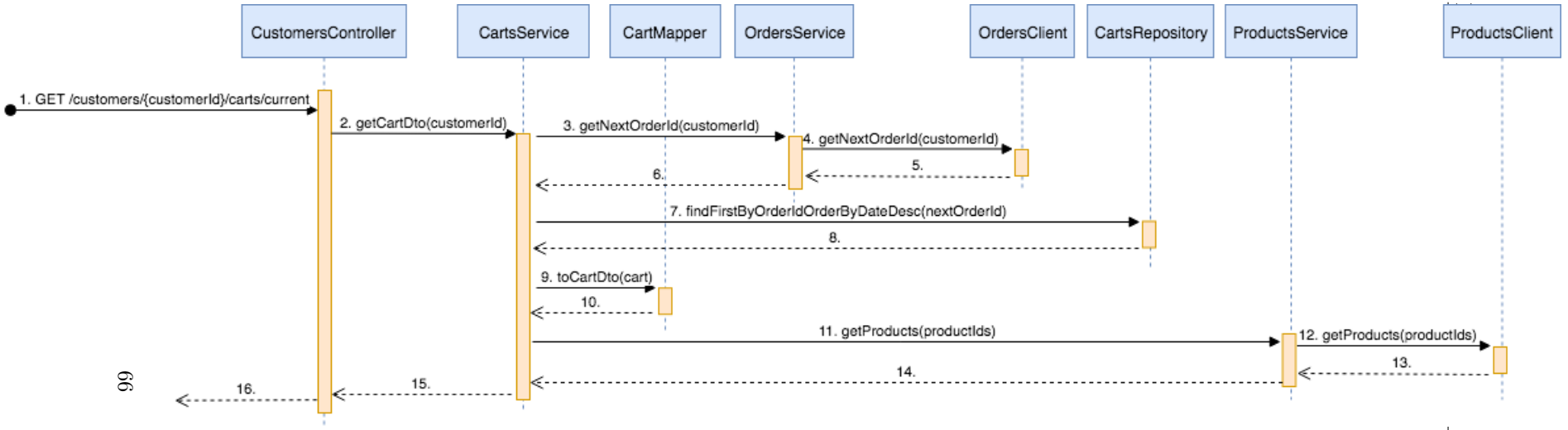


Figura 53: Diagrama de seqüència d'una crida `GET /customers/{customerId} /carts/current`

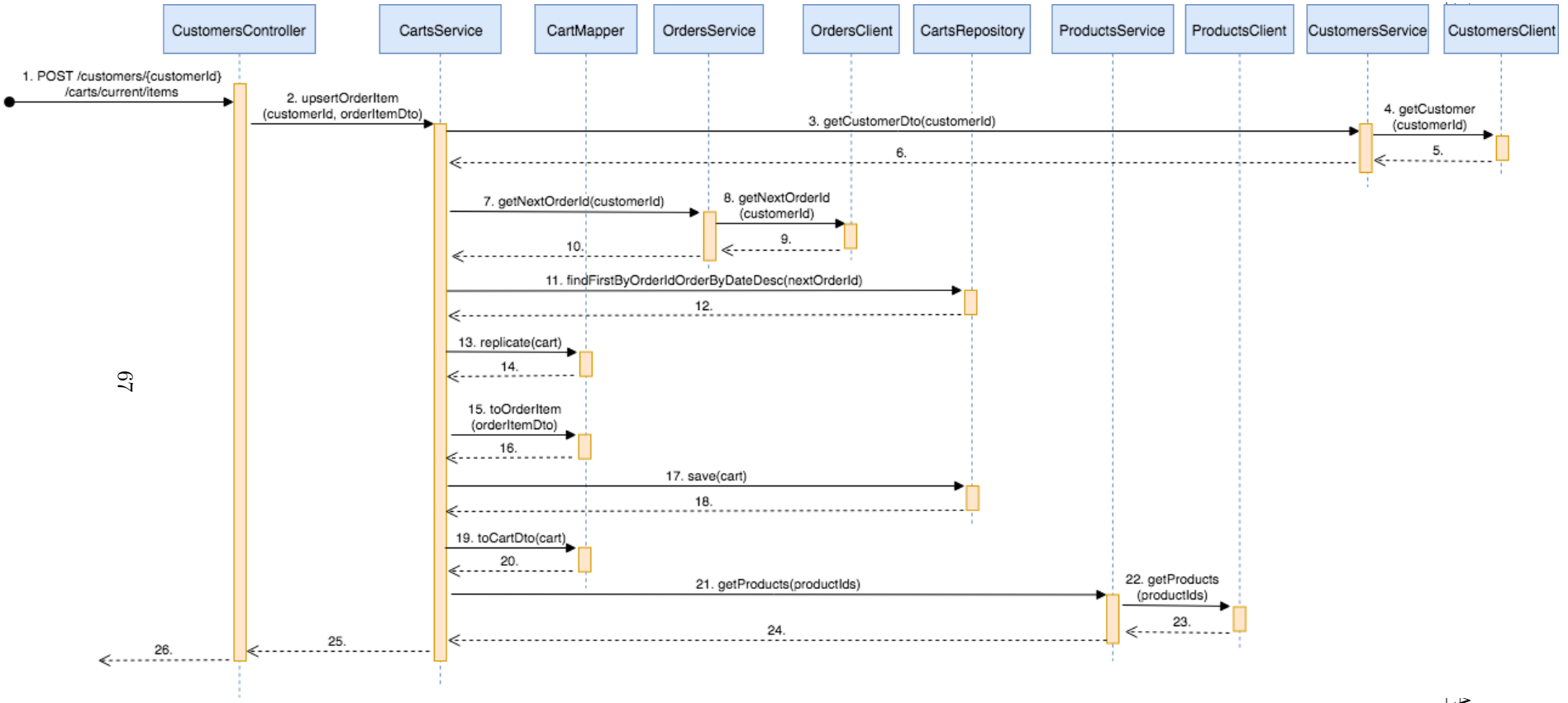


Figura 54: Diagrama de seqüència d'una crida `POST /customers/{customerId} /carts/current/items`

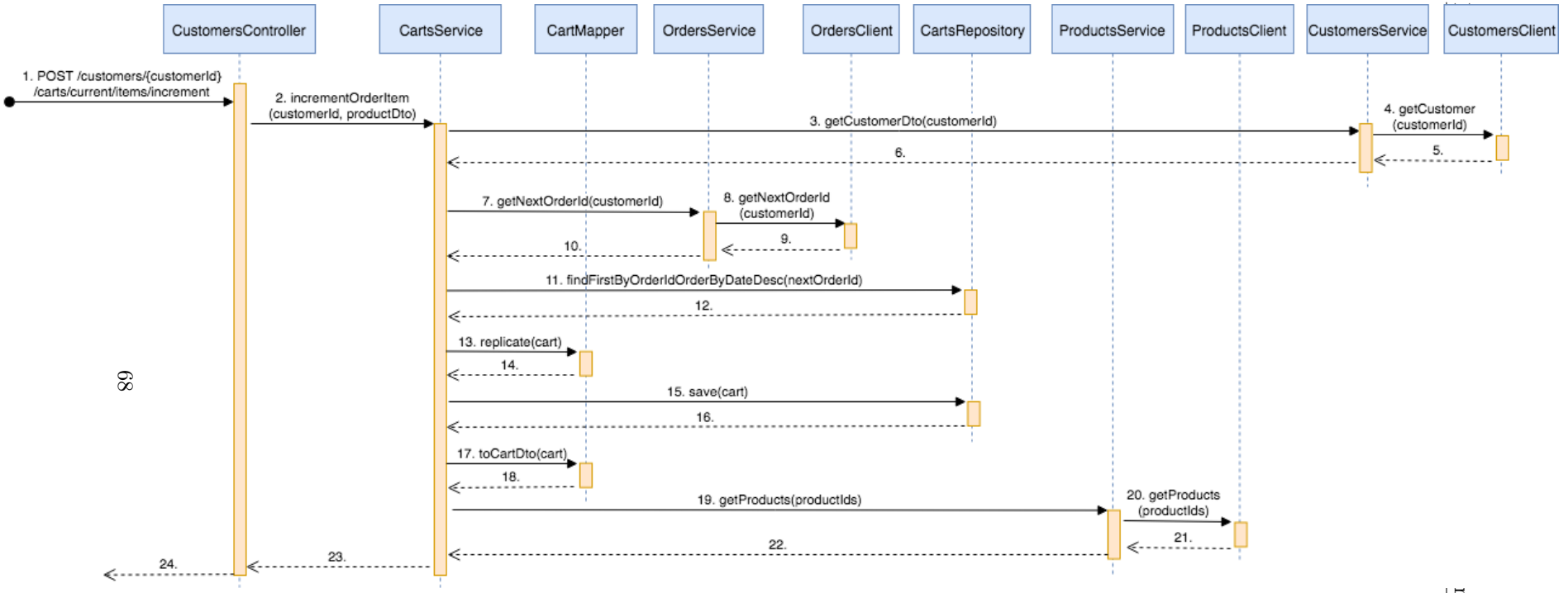


Figura 55: Diagrama de seqüència d'una crida `POST /customers/{customerId} /carts/current/items/increment`

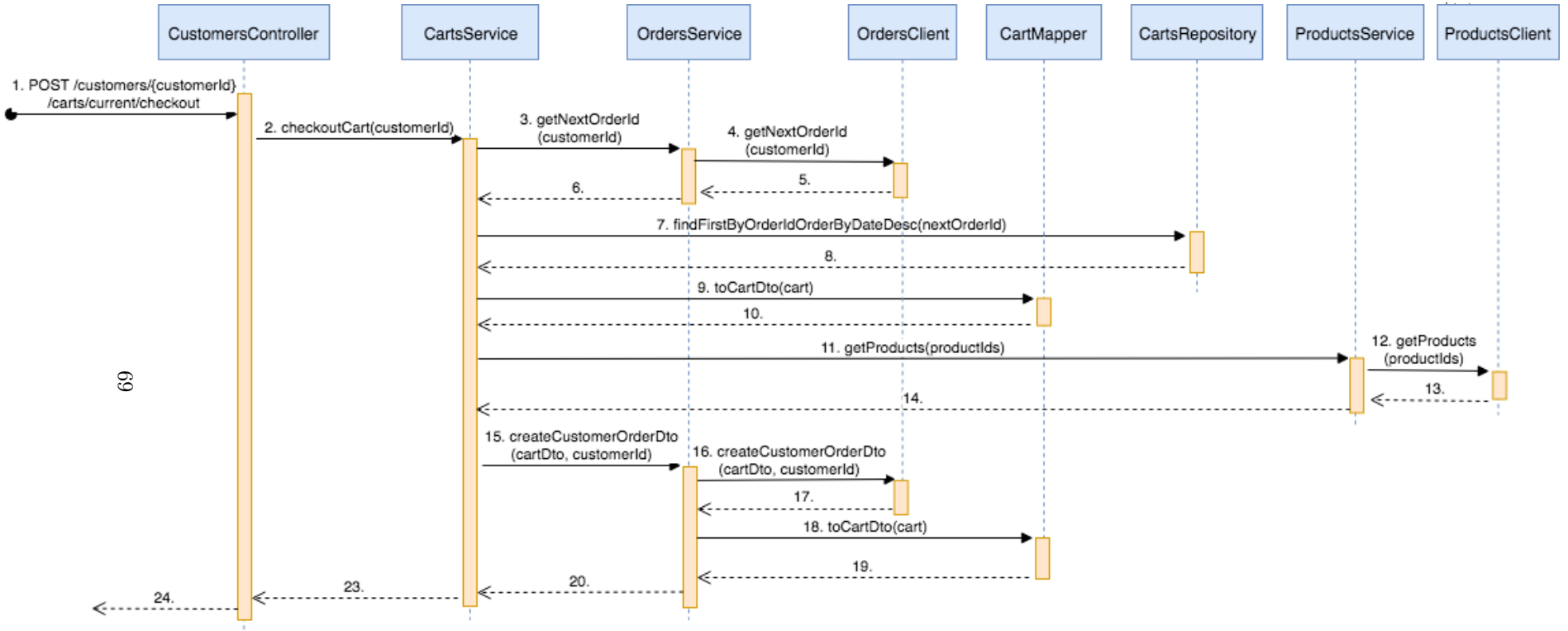


Figura 56: Diagrama de seqüència d'una crida `POST /customers/{customerId} /carts/current/checkout`

4.8 *Orders Service*

Aquesta aplicació conté la lògica de la gestió de comandes. El codi d'aquest servei es troba a https://github.com/jaarques-uoc/orders-ws.

Els *endpoints* exposats per aquest servei són els següents:

- `GET /init`. Esborra totes les comandes existents. No s'inicialitza la base dades amb cap comanda per defecte.
- `GET /customers/{customerId}/orders`. Retorna el llistat de comandes per a l'usuari amb l'identificador `{customerId}`.
- `POST /customers/{customerId}/orders`. Crea una nova comanda per a l'usuari amb l'identificador `{customerId}`. La informació dels productes de la comanda s'envia al *body* de la crida *REST*.
- `GET /customers/{customerId}/nextOrderId`. Obté l'identificador de la propera futura comanda a crear per a l'usuari amb l'identificador `{customerId}`.
- `GET /orders`. Retorna el llistat de totes les comandes que existeixen a la base de dades.
- `GET /orders/{id}`. Retorna la informació completa de la comanda amb identificador `{id}`.
- `GET /orders/{id}/events`. Retorna el llistat d'events de la comanda amb identificador `{id}`.

El cas de la creació d'una comanda és especialment interessant, ja que, de manera molt simplificada, s'hi ha tingut en compte els events que arriben a posteriori com el pagament, la preparació de la comanda o l'enviament. Aquestes funcionalitats s'han *mockejat* i simplement s'ha usat la comanda `Thread.sleep(20000)` per tal d'emular el retard amb el que apareixen, així cada 20s representa que s'ha realitzat una d'aquestes fases.

Per tal de no bloquejar a l'usuari mentre ess processa el pagament o qualsevol de les altres accions que es realitzen amb un retard, s'ha fet ús de la llibreria Reactor per tal d'executar aquestes crides en un *thread* paral·lel.

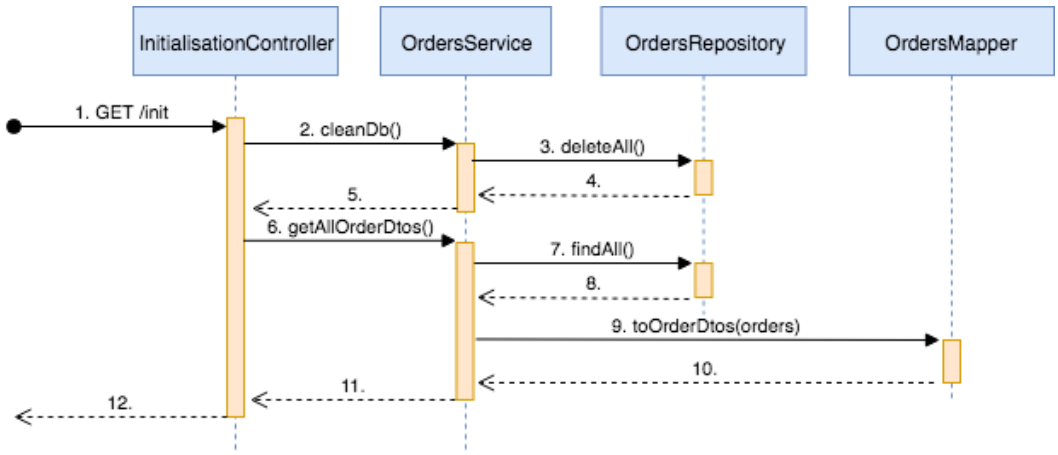


Figura 57: Diagrama de seqüència d'una crida `GET /init`

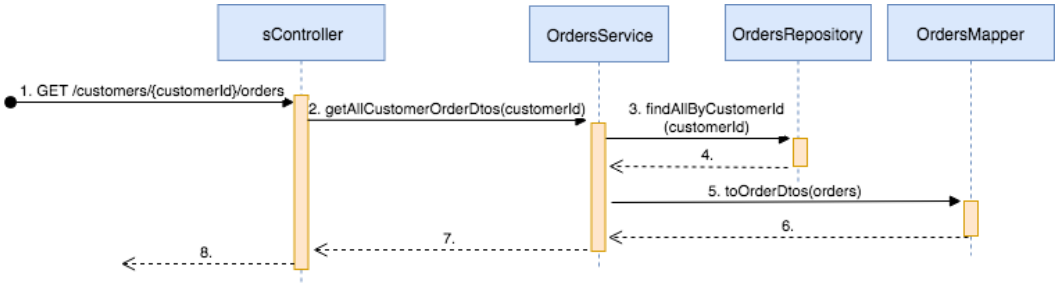


Figura 58: Diagrama de seqüència d'una crida `GET /customers/{customerId} /orders`

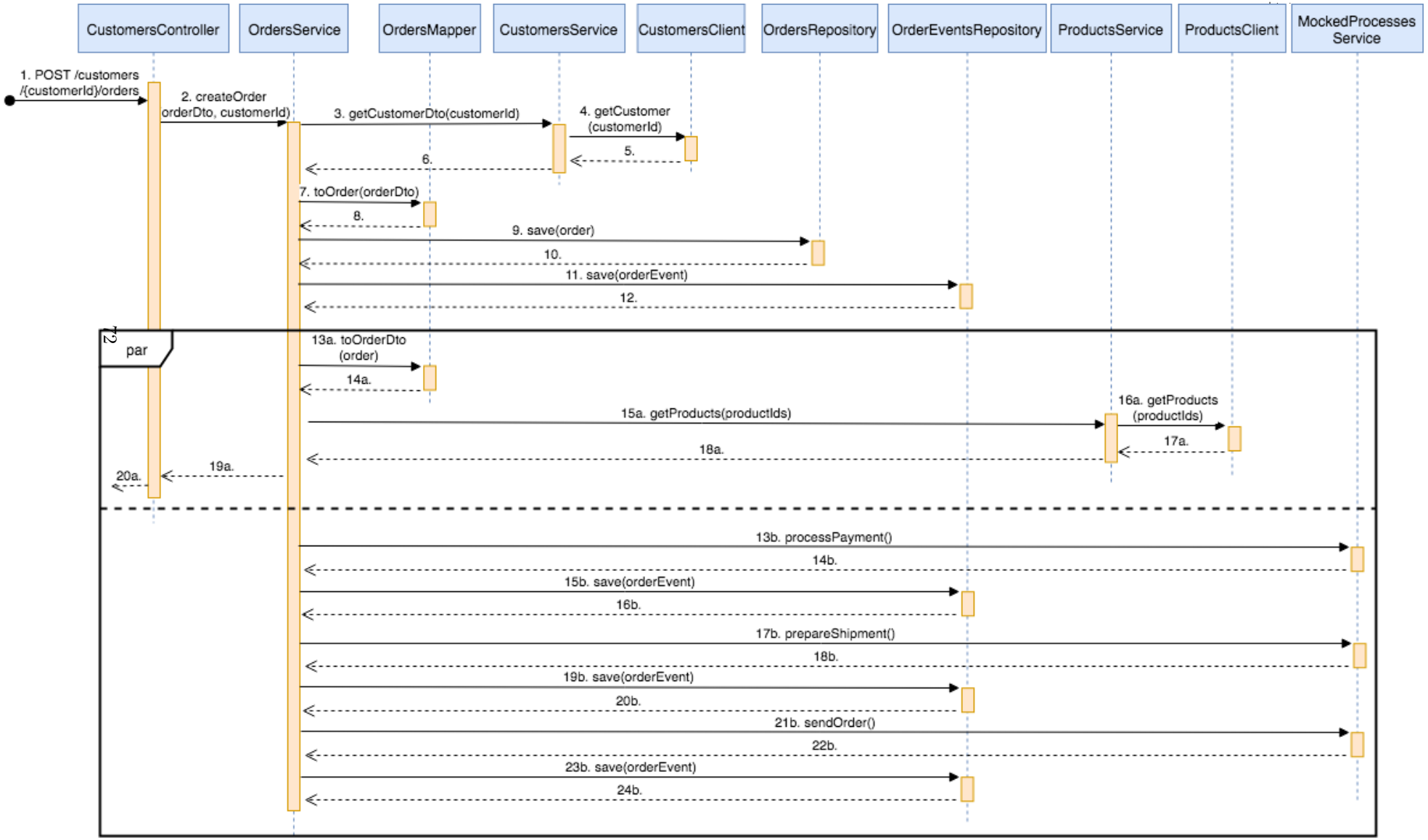


Figura 59: Diagrama de seqüència d'una crida `POST /customers/{customerId} /orders`

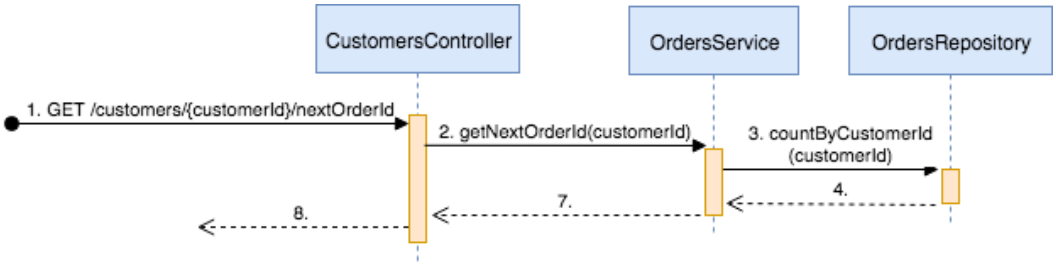


Figura 60: Diagrama de seqüència d'una crida `GET /customers/{customerId} /nextOrderId`

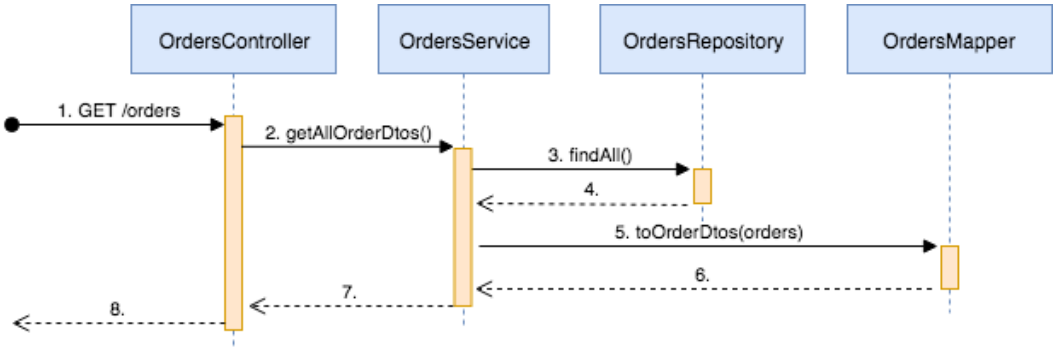


Figura 61: Diagrama de seqüència d'una crida `GET /orders`

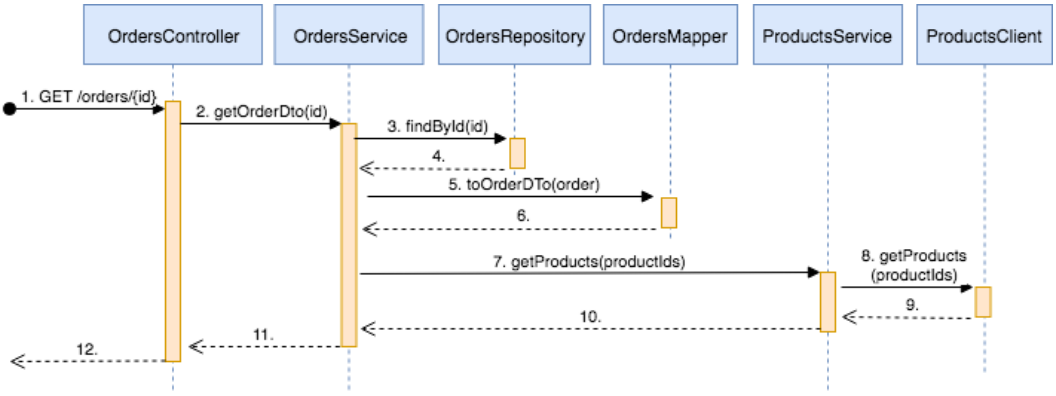


Figura 62: Diagrama de seqüència d'una crida `GET /orders/{id}`

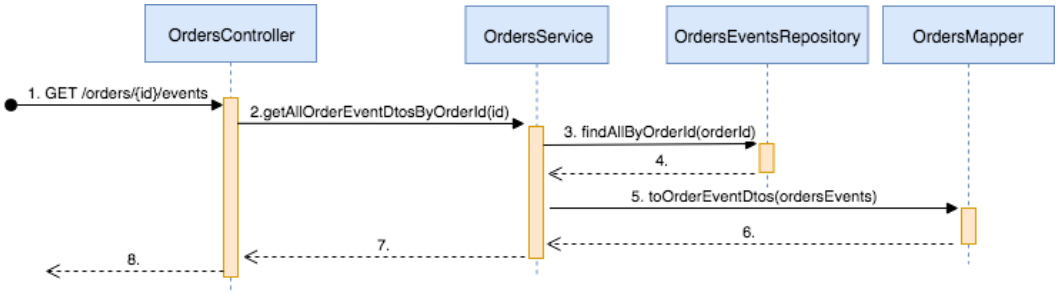


Figura 63: Diagrama de seqüència d'una crida `GET /orders/{id}/events`

5 Treball futur

Per tal de continuar amb el desenvolupament d'aquest projecte, els següents aspectes a treballar i a aprofundir són:

- Completar el sistema amb altres microserveis que cobreixin funcionalitats com: inventari, pagament, enviament i marketing. Durant el desenvolupament d'aquest projecte s'ha obviat aquests aspectes ja que haurien afegit una càrrega de treball molt elevada per a la seva realització i, probablement, no s'hagués pogut completar a temps.

  No obstant, tota botiga online ha de poder gestionar no solament els productes sinó també el seu inventari; ha de poder realitzar pagaments i, fins i tot, integrar-se amb altres passarel·les de pagament com PayPal; també, ha de poder preparar les comandes i realitzar els enviaments; i, finalment, és útil que tingui la seva pròpia plataforma de gestió d'ofertes i promocions.

- Millorar la capa de seguretat del sistema. La implementació actual és molt fàcilment *hackejable* i, de fet, no conté cap restricció ni control de seguretat a nivell dels *endpoints* per evitar que un usuari no identificat faci crides *REST* directament contra aquests endpoints.

  Per tant, una persona amb uns coneixements suficients de desenvolupament de *software* podria investigar la web de la botiga online i extreure les crides als diferents *endpoints*. A partir d'aquí, li seria molt fàcil fer-se passar per un altre usuari i realitzar comandes sense el seu permís.

  Per tal de resoldre aquest problema, s'hauria de crear un altre microservei que exercís de servidor d'autenticació i que limités l'accés als endpoints per a usuaris amb un rol determinat. Això es podria aconseguir mitjançant la llibreria Spring Security i amb la implementació de *Single Sign-On* (*SSO*).

  A part, també caldria afegir paginació per als *endpoints* que retornin llistes, sinó, es podria arribar a patir una fallida al sistema en intentar obtenir tots els registres de la base de dades.

- Heroku és un bon entorn de treball per a projectes petits i proves de concepte, però si es vol crèixer i tenir un entorn de desenvolupament complet i estable, la millor opció actualment és Amazon Web Services (AWS) que aporta moltes més funcionalitats i eines més professionals.

  Per exemple, una de les millores d'utilitzar AWS, d'entre moltes altres, consistiria en poder definir correctament els servidors que han der públics a Internet i els que han de residir a dintre d'una xarxa privada sense accés

directe des de l'exterior.

També, s'ha de deixar de fer servir les versions de prova de Heroku i TravisCI, ja que tenen unes limitacions importants. Heroku només habilita fins a 5 aplicacions gratuïtes per compte amb una limitació de 512Mb de RAM (que és sovint sobrepassada per les aplicacions Spring). TravisCI només permet monitoritzar 100 desplegaments gratuïts al mes.

- Moure tots els *Data Transfer Objects* (*DTOs*) a una llibreria externa i comuna que tots els microserveis afegeixin com a dependència. D'aquesta manera, s'evitaria la redundància i duplicitat de classes entre els diferents microserveis, tot i que afegiria un nivell d'acoplament que ara no existeix. Caldria pensar bé aquest disseny ja que un millor disseny podria consistir en repartir els *DTOs* en més d'una llibreria de manera que cadascuna contingués un subtipus d'aquests objectes i no tots els microserveis en depenguessin de la mateixa i incloguessin *DTOs* que no necessiten.

- Afegir tests i millorar la cobertura del codi. La millor forma d'assegurar-se que el codi desenvolupat funciona correctament i no espatlla cap funcionalitat existent consisteix en tenir una gran bateria de tests unitaris i d'integració que controlen que els microserveis responen correctament sota unes determinades situacions. També es podrien definir uns tests de sistema que comprovessin que la comunicació entre els diferents microserveis i el sistema en global funciona correctament.

- Habilitar la internacionalització de la botiga online. Al disseny actual només s'ha tingut en compte l'idioma anglés i l'Euro com a moneda. Si la intenció fos arribar a un públic global i no centrar-se en un país o regió en concret, caldria afegir diferents monedes i traduir la web a altres idiomes.

  Això implica que els textos de la web s'haurien de moure a arxius d'idioma i mantenir un mapatge de tipus clau-valor on la clau fos un *string* genèric i el valor fos la traducció a un idioma en concret. Cada arxiu d'idioma ha de mantenir el mateix mapatge per tal que les traduccions de la web siguin consistents.

- Documentació dels *endpoints* a Swagger. Swagger és una eina molt útil per a provar les crides *REST* d'un microservei. Ajuda molt a l'hora de desenvolupar funcionalitats que requereixin la integració entre dos microserveis diferents.

6 Conclusions

El desenvolupament presentat en aquesta memòria mostra un sistema de microserveis mantenint una separació de responsabilitats i seguint els principis *SOLID*.

S'ha mostrat com el desenvolupament d'una aplicació que, tradicionalment s'hauria creat seguint un disseny en monolit, és implementable seguint una arquitectura en microserveis. Tot i que, per a aplicacions petites i proves de concepte, la creació d'un sistema de microserveis pot semblar excessiva, aquesta divisió aconsegueix que el desenvolupament de futures funcionalitats sigui molt més àgil i fàcil d'implementar. També, s'ha vist que només cal desplegar i modificar els microserveis que es veuen afectats per aquests canvis, mantenint la resta del sistema estable i sense modificar.

Per a aconseguir aquest propòsit s'ha fet ús de moltes de les eines i llibreries que ens aporta el *framework* Spring per al desenvolupament del *backend*. Per a la implementació del *frontend* s'ha escollit la llibreria ReactJS que tot i acoplar les vistes HTML amb el codi JavaScript és un entorn de desenvolupament molt intuitiu i eficaç.

Així, s'ha pogut aprofundir i estudiar molts dels aspectes de Spring, que és, juntament amb J2EE, el *framework* més estable i utilitzat per al desenvolupament d'aplicacions a nivell professional en l'entorn de Java. D'altra banda, tot i la volatilitat de les llibreries JavaScript (apareixen noves llibreries amb molta més facilitat que per a aplicacions de *backend*), s'ha pogut observar com ReactJS és una llibreria molt completa, estable, amb una gran comunitat i que facilita la implementació del *frontend*.

També, s'ha pogut investigar i desenvolupar característiques com el servei de descobriment de microserveis o l'*API Gateway*, que sovint queden ocultes al desenvolupador ja que un cop configurades no és tan normal haver-les de modificar.

Cal mencionar que es va dedicar un temps i un esforç important a poder posar en marxa un sistema d'autenticació amb OAuth2 i SSO al backend, però que finalment es va haver de deixar a part degut a la manca de temps i a que no es va aconseguir fer-lo funcionar completament.

Per una altra banda, la incorporació de llibreries reactives com Reactor i la utilització d'una base de dades NoSQL ha permès aprofundir els coneixements i l'ha fet molt interessant des d'un punt de vista de formació i d'investigació.

Finalment, el procés d'integració contínua definit, ha permés entendre eines com TravisCI i Heroku amb més profunditat. També, ha permés veure les limitacions de les seves versions de prova i ser conscient de la necessitat d'utilitzar eines

més professionals i potents per als sistemes de producció.

7 Bibliografia

Richardson C, Smith F. Microservices: From Design to Deployment. NGINX Inc., 2006. Web

Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015. Print

Java™ Platform, Standard Edition 8 API Specification: https://docs.oracle.com/javase/8/docs/api/

Pattern: Backends For Frontends: https://samnewman.io/patterns/architectural/bff/

Martin Fowler. Microservices: https://martinfowler.com/articles/microservices.html
Spring Cloud Netflix: https://spring.io/projects/spring-cloud-netflix

Accessing Data with MongoDB: https://spring.io/guides/gs/accessing-data-mongodb/

Pattern: Event sourcing: https://microservices.io/patterns/data/event-sourcing.html

Project Reactor documentation: https://projectreactor.io/docs

Project Lombok documentation: https://projectlombok.org/

Baeldung courses: https://www.baeldung.com/

Create React App: https://github.com/facebook/create-react-app

Getting started with React Router: https://codeburst.io/getting-started-with-react-router-5c978f70df91

Deploying Spring Boot RestAPI using Docker, Maven, Heroku and accessing it using your custom domain name. Part 1 of 2: https://medium.com/@urbanswati/deploying-spring-boot-restapi-using-docker-maven-heroku-and-accessing-it-using-your-custom-aa04798c0112

CI/CD for SpringBoot applications using Travis-CI: https://sivalabs.in/2018/01/ci-cd-springboot-applications-using-travis-ci/

Push images to Docker Cloud: https://docs.docker.com/v17.12/docker-cloud/builds/push-images/

Travis CI User Documentation: https://docs.travis-ci.com/

Spring guides: https://spring.io/guides/gs/spring-boot-docker/

Boostrap 4.3 documentation: https://getbootstrap.com/docs/4.3/getting-started/introduction/

Setting Up a Redux Project With Create-React-App: https://medium.com/backticks-tildes/setting-up-a-redux-project-with-create-react-app-e363ab2329b8

Redux documentation: https://redux.js.org/introduction/getting-started

Single Page Apps for GitHub Pages: https://github.com/rafrex/spa-github-pages

OAuth2 Boot Documentation: https://docs.spring.io/spring-security-oauth2-boot/docs/current-SNAPSHOT/reference/htmlsingle/

Hystrix documentation: https://github.com/Netflix/Hystrix/wiki/Configuration