ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή Εργασία

Τίτλος Πτυχιακής Εργασίας	**Σχεδιασμός και Υλοποίηση ενός Saga Orchestrator με χρήση Μηχανής Καταστάσεων σε ένα Σύστημα Μικροϋπηρεσιών**
	Design and Implementation of a Saga Orchestrator using a State Machine in a Microservices System
Ονοματεπώνυμο Φοιτητή	Αντώνιος Ρούσσος
Πατρώνυμο	Δημήτριος
Αριθμός Μητρώου	Π20167
Επιβλέπων	Δημήτριος Βέργαδος, Καθηγητής

Ημερομηνία Παράδοσης: Φεβρουάριος 2026

Copyright ©

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες σε όλους όσοι συνέβαλαν, με άμεσο ή έμμεσο τρόπο, στην ολοκλήρωση της παρούσας πτυχιακής εργασίας.

Αρχικά, ευχαριστώ θερμά τον κ. Βέργαδο Δημήτριο, ο οποίος αποδέχθηκε την επίβλεψη της πτυχιακής μου εργασίας και ήταν πρόθυμος να συζητήσει και να προσφέρει πολύτιμη βοήθεια καθ' όλη τη διάρκεια της εκπόνησής της.

Επιπλέον, θα ήθελα να ευχαριστήσω όλους τους καθηγητές του Τμήματος Πληροφορικής του Πανεπιστημίου Πειραιά για τις γνώσεις και τα εφόδια που μου παρείχαν κατά τη διάρκεια των σπουδών μου.

Ιδιαίτερες ευχαριστίες οφείλω στους κ. Μιχάλα Άγγελο και κ. Χυτήρη Χρήστο, καθηγητές του Τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Δυτικής Μακεδονίας, για την ουσιαστική βοήθεια, τις παρατηρήσεις και την καθοδήγησή τους.

Τέλος, θα ήθελα να ευχαριστήσω τα κοντινά μου πρόσωπα για την συνεχή στήριξη και την ενθάρρυνση που μου προσέφεραν καθ' όλη τη διάρκεια της εκπόνησης της πτυχιακής μου εργασίας.

Περίληψη

Η παρούσα πτυχιακή εργασία επικεντρώνεται στο σχεδιασμό και την υλοποίηση ενός Saga Orchestrator για τη διαχείριση κατανεμημένων συναλλαγών σε μια αρχιτεκτονική μικροϋπηρεσιών. Στόχος της εργασίας είναι η μελέτη και η εφαρμογή του προτύπου Saga, προκειμένου να επιτευχθεί η τελική συνέπεια των δεδομένων χωρίς τη χρήση ACID συναλλαγών. Για τον συντονισμό της ροής χρησιμοποιείται μια μηχανή καταστάσεων με τη βιβλιοθήκη Spring Statemachine, η οποία επιτρέπει την διαχείριση των επιμέρους βημάτων μιας διαδικασίας, όπως η εξουσιοδότηση πληρωμής και η δέσμευση αποθέματος σε ένα απλοποιημένο σύστημα ηλεκτρονικού εμπορίου. Η επικοινωνία μεταξύ των μικροϋπηρεσιών πραγματοποιείται ασύγχρονα μέσω RabbitMQ, ενώ σε περιπτώσεις σφαλμάτων ενεργοποιούνται κατάλληλες αντισταθμιστικές ενέργειες. Η υλοποίηση αξιολογείται σε περιβάλλον Docker Compose, με σκοπό την παρακολούθηση της συμπεριφοράς του orchestrator τόσο σε κανονικά σενάρια εκτέλεσης όσο και σε βασικές περιπτώσεις αποτυχίας.

Λέξεις-Κλειδιά: Saga Pattern, Microservices Architecture, Distributed Transactions, State Machine Orchestration

Abstract

This thesis focuses on the design and implementation of a Saga Orchestrator for managing distributed transactions in a microservices architecture. The aim of the thesis is to study and apply the Saga pattern in practice, in order to achieve eventual data consistency without the use of ACID transactions. A state machine with the Spring Statemachine library is used to coordinate the flow, which allows for the organized management of the individual steps of a business process, such as payment authorization and inventory reservation in a simplified e-commerce system. Communication between microservices is performed asynchronously via RabbitMQ, while in case of errors, appropriate compensating actions are triggered. The implementation is evaluated in a Docker Compose environment, with the aim of monitoring the behavior of the orchestrator in both normal execution scenarios and basic failure cases.

Keywords: Saga Pattern, Microservices Architecture, Distributed Transactions, State Machine Orchestration

Περιεχόμενα

Κατάλογος σχημάτων

Κατάλογος πινάκων

Κατάλογος Αποσπασμάτων

1. Εισαγωγή

Στη σύγχρονη ανάπτυξη λογισμικού, η ανάγκη για συστήματα με υψηλή διαθεσιμότητα και δυνατότητα κλιμάκωσης ολοένα και αυξάνεται. Η μετάβαση από τη μονολιθική αρχιτεκτονική στην αρχιτεκτονική μικροϋπηρεσιών προσφέρει αρκετά οφέλη όσον αφορά την ευελιξία και τις δυνατότητες κλιμάκωσης. Ταυτόχρονα, εισάγει σημαντικές προκλήσεις στη διαχείριση της συνέπειας των δεδομένων. Για παράδειγμα, επειδή κάθε υπηρεσία διαθέτει τη δική της βάση δεδομένων, κάποιες ενέργειες που σε ένα μονολιθικό σύστημα θα υλοποιούνταν ως μια ACID συναλλαγή, σε ένα σύστημα μικροϋπηρεσιών απαιτούν συντονισμό πολλαπλών τοπικών συναλλαγών σε διαφορετικές βάσεις δεδομένων, όπου η άμεση ισχυρή συνέπεια δεν είναι πάντα εφικτή. Στο πλαίσιο αυτό, η παρούσα πτυχιακή εργασία εξετάζει το πρόβλημα της συνέπειας των δεδομένων εστιάζοντας στη μελέτη του προτύπου Saga, μιας στρατηγικής που εν μέρει λύνει αυτό το πρόβλημα και διασφαλίζει την τελική συνέπεια των δεδομένων.

Αντικείμενο της παρούσας πτυχιακής εργασίας είναι η κατανόηση, ο σχεδιασμός και η υλοποίηση των κύριων εσωτερικών μηχανισμών που χρησιμοποιεί ένας Saga Orchestrator με τη χρήση μιας μηχανής καταστάσεων. Σκοπός είναι να παρουσιαστεί πώς μπορεί να επιτευχθεί ο συντονισμός των μικροϋπηρεσιών σε μια κατανεμημένη συναλλαγή, όπως η δημιουργία παραγγελίας σε ένα σύστημα ηλεκτρονικού εμπορίου, διατηρώντας την αυτονομία των εμπλεκόμενων υπηρεσιών. Η υλοποίηση επικεντρώνεται στη χρήση της βιβλιοθήκης Spring Statemachine εντός του Spring οικοσυστήματος, προσφέροντας μια εναλλακτική λύση στα υπάρχοντα frameworks, με στόχο την καλύτερη κατανόηση των εσωτερικών μηχανισμών διαχείρισης καταστάσεων και μεταβάσεων.

Η δομή της εργασίας ξεκινά με τη βιβλιογραφική ανασκόπηση για την εξέλιξη των αρχιτεκτονικών και τους μηχανισμούς συνέπειας στα κατανεμημένα συστήματα και συνεχίζει με τον σχεδιασμό της αρχιτεκτονικής και της ροής δημιουργίας παραγγελίας. Έπειτα παρουσιάζεται η υλοποίηση του orchestrator, της μηχανής καταστάσεων και των συμμετεχουσών υπηρεσιών, μαζί με τον τρόπο ανταλλαγής μηνυμάτων μέσω του message broker. Τέλος, ακολουθεί μια πρακτική αξιολόγηση σε τοπικό περιβάλλον με Docker Compose, με σενάριο επιτυχούς εκτέλεσης και σενάριο αποτυχίας, ώστε να επιβεβαιωθεί ότι ενεργοποιούνται οι σωστές αντισταθμιστικές ενέργειες και το σύστημα καταλήγει σε συνεπή τελική κατάσταση.

2. Βιβλιογραφική Ανασκόπηση

2.1 Η Πορεία των Αρχιτεκτονικών Λογισμικού

Η εξέλιξη των αρχιτεκτονικών λογισμικού αποτελεί μια συνεχή προσπάθεια εξισορρόπησης μεταξύ της απλότητας, της απόδοσης και της αυξανόμενης ανάγκης για ευελιξία (flexibility) και κλιμάκωση (scalability). Ιστορικά, αυτή η πορεία μετακίνησε τα συστήματα από στενά συζευγμένα (tightly coupled) προς συστήματα που είναι πιο αυτόνομα και ευκολότερα επεκτάσιμα.

2.1.1 Από τη Μονολιθική Αρχιτεκτονική στην Αρχιτεκτονική Μικροϋπηρεσιών

Για δεκαετίες, η αρχιτεκτονική λογισμικού που κυριαρχούσε ήταν η Μονολιθική Αρχιτεκτονική (Monolithic Architecture), στην οποία όλες οι λειτουργίες της εφαρμογής είναι ενσωματωμένες σε μια βάση κώδικα. Αυτή η αρχιτεκτονική καθιστά ένα σαφές σημείο εκκίνησης για μικρά αλλά και μεγάλα έργα καθώς είναι πολύ απλή για την αρχική ανάπτυξη ενός συστήματος. Επιπλέον, η κοινή βάση κώδικα καθιστά πιο εύκολη την διαδικασία της αποσφαλμάτωσης (debugging) του κώδικα. Σημαντικό πλεονέκτημα των μονολιθικών συστημάτων είναι η υψηλή απόδοση και η χαμηλή καθυστέρηση (latency) σε συνθήκες χαμηλού φόρτου, καθώς η επικοινωνία μεταξύ των στοιχείων (components) γίνεται εντός της ίδιας διεργασίας, αποφεύγοντας την επιβάρυνση του δικτύου [4, 14]. Παρ'όλ'αυτά, το σύστημα πρέπει να εγκατασταθεί ολόκληρο σε περιβάλλον παραγωγής (deployment) μετά από οποιαδήποτε αλλαγή. Τα components δεν μπορούν να λειτουργήσουν αυτόνομα με αποτέλεσμα αυτή η στενή σύζευξη (tight coupling) να καθιστά τη μακροπρόθεσμη συντήρηση δύσκολη, ιδιαίτερα όταν η βάση κώδικα αυξάνεται [7, 45].

Καθώς τα έργα μεγάλωναν σε πολυπλοκότητα, τα μειονεκτήματα των μονολιθικών συστημάτων άρχιζαν να γίνονται όλο και πιο εμφανή. Η κύρια πρόκληση που αναφέρεται στη βιβλιογραφία αφορά το scalability, το οποίο είναι συχνά δαπανηρό και αναποτελεσματικό. Στα μονολιθικά συστήματα, η κλιμάκωση επιτυγχάνεται κυρίως κάθετα (vertical scaling) και έχει υψηλό κόστος. Ακόμη και στην περίπτωση της οριζόντιας κλιμάκωσης (horizontal scaling), απαιτείται το deployment ολόκληρης της εφαρμογής, γεγονός που οδηγεί στην σπατάλη πόρων [15, 7, 14]. Επιπλέον, όσο η βάση του κώδικα γίνεται μεγαλύτερη και πολυπλοκότερη, αυτό καθιστά τις διορθώσεις σφαλμάτων, τις βελτιώσεις λειτουργιών και την γενική κατανόηση του συστήματος ολοένα και δυσκολότερη. Ακόμη και μικρές αλλαγές στον κώδικα απαιτούν την ανακατασκευή και το εκ νέου deployment ολόκληρης της εφαρμογής, οδηγώντας σε πιο αργούς ρυθμούς ανάπτυξης [7]. Αξίζει επίσης να σημειωθεί ότι τα μονολιθικά συστήματα υλοποιούνται σε μια συγκεκριμένη στοίβα τεχνολογιών (technology stack) καθιστώντας δύσκολη την ενσωμάτωση νέων τεχνολογιών. Αυτή η αυξανόμενη πολυπλοκότητα ώθησε τους μηχανικούς λογισμικού να αναζητήσουν εναλλακτικές λύσεις.

Ως απάντηση σε αυτούς τους περιορισμούς, η βιομηχανία κινήθηκε αρχικά προς την Αρχιτεκτονική Προσανατολισμένη στις Υπηρεσίες (Service-Oriented Architecture - SOA). Η SOA εισήγαγε

την έννοια της διάσπασης των εφαρμογών σε χαλαρά συζευγμένες (loosely coupled) υπηρεσίες, με στόχο την επαναχρησιμοποίηση της επιχειρησιακής λογικής (business logic). Παρ'όλ'αυτά, η υλοποίηση της SOA συχνά βασιζόταν σε πολύπλοκους κεντρικούς μηχανισμούς επικοινωνίας, όπως ο Δίαυλος Εξυπηρέτησης Επιχειρήσεων (Enterprise Service Bus - ESB). Ο ESB, αν και σχεδιάστηκε για να διευκολύνει την ενοποίηση, συχνά μετατρεπόταν σε bottleneck και εισήγαγε νέα επίπεδα πολυπλοκότητας και κόστους με αποτέλεσμα να μην προσφέρει την ευελιξία που απαιτούσαν τα σύγχρονα συστήματα.

Η Αρχιτεκτονική Μικροϋπηρεσιών (Microservices Architecture - MSA), αναδείχθηκε επίσημα το 2014, ως η σύγχρονη εξέλιξη της SOA και έχει καθιερωθεί ως το πρότυπο για κατανεμημένα συστήματα μεγάλης κλίμακας. Αξίζει να αναφερθεί πως ο όρος «microservices» χρησιμοποιήθηκε για πρώτη φορά το 2011 κατά τη διάρκεια ενός εργαστηρίου αρχιτεκτονικής και έπειτα παρουσιάστηκε πιο επίσημα στο 33ο Degree Conference στην Κρακοβία το 2012. Το 2014 όμως, οι μικροϋπηρεσίες ξεκίνησαν να κερδίζουν σημαντικά το ενδιαφέρον των μηχανικών λογισμικού χάρη στους Lewis και Fowler οι οποίοι συνέταξαν ένα άρθρο σχετικά με αυτό και στη συνέχεια ακολούθησαν μέλη της Netflix τα οποία μοιράστηκαν τις γνώσεις τους σχετικά με το θέμα. Η Netflix είχε προσεγγίσει την αρχιτεκτονική μικροϋπηρεσιών από το 2009 όταν ο όρος δεν είχε ακόμη υπάρξει επίσημα. Από το 2014 και μετέπειτα, άλλες εταιρίες όπως οι Spotify και SoundCloud έχουν στραφεί στην αρχιτεκτονική μικροϋπηρεσιών [32]. Σε αντίθεση με την μονολιθική αρχιτεκτονική, σύμφωνα με τον Newman, στην MSA, το σύστημα διασπάται σε πολλές μικρές, ανεξάρτητες υπηρεσίες που επικοινωνούν μεταξύ τους μέσω πρωτοκόλλων όπως το HTTP [26]. Αυτό το μοντέλο προσφέρει καλύτερο scalability και flexibility, καθώς επιτρέπει το ανεξάρτητο scaling συγκεκριμένων υπηρεσιών βάσει ζήτησης. Αυτό έχει ως αποτέλεσμα την βελτιστοποίηση χρήσης πόρων [7, 14, 3]. Επιπλέον, η MSA ενισχύει την ανοχή σε σφάλματα (fault tolerance), καθώς μια αποτυχία μιας υπηρεσίας δεν οδηγεί στην κατάρρευση ολόκληρου του συστήματος, ενώ επιτρέπει την τεχνολογική ετερογένεια δίνοντας τη δυνατότητα στους μηχανικούς λογισμικού να επιλέγουν τα κατάλληλα εργαλεία για κάθε πρόβλημα [3, 4, 45].

2.1.2 Οι Προκλήσεις της Αρχιτεκτονικής Μικροϋπηρεσιών

Παρότι οι μικροϋπηρεσίες προσφέρουν σημαντικά οφέλη, εισάγουν παράλληλα νέες προκλήσεις που δεν πρέπει να αγνοηθούν. Η μετάβαση σε MSA μπορεί να φέρει αρνητικό αντίκτυπο στην απόδοση του συστήματος λόγω της επιπλέον καθυστέρησης της σειριοποίησης, αποσειριοποίησης και μεταφοράς των δεδομένων. Συγκεκριμένα, έρευνα έχει δείξει ότι η καθυστέρηση (latency) μπορεί να αυξηθεί δραματικά, με μετρήσεις να δείχνουν επιβάρυνση από 201% έως και 6977% σε σύγκριση με μονολιθικές υλοποιήσεις, ανάλογα με την πολυπλοκότητα των κλήσεων [9].

Μια από τις σημαντικότερες προκλήσεις αφορά τη λειτουργική πολυπλοκότητα, καθώς η διαχείριση ενός κατανεμημένου συστήματος απαιτεί αυξημένο λειτουργικό κόστος για την παρακολούθηση (monitoring), την ασφάλεια και τη διαχείριση δεκάδων ή εκατοντάδων αυτόνομων υπηρεσιών [3, 4]. Η απλότητα της διαχείρισης μιας ενιαίας εφαρμογής αντικαθίσταται από την ανάγκη συντονισμού πολλών αυτόνομων υπηρεσιών.

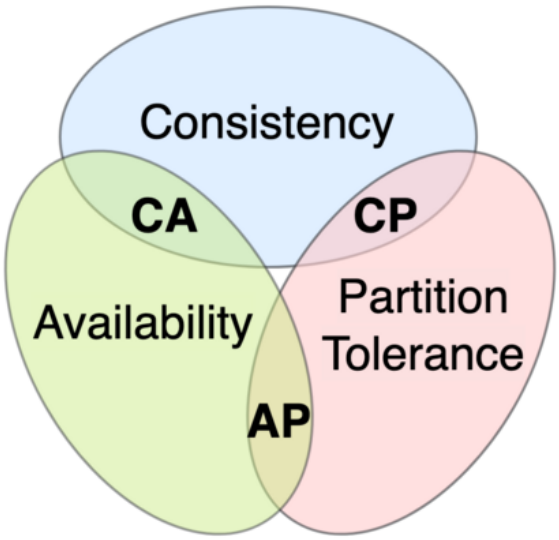
Η μεγαλύτερη όμως πρόκληση αφορά τη διατήρηση της συνέπειας των δεδομένων. Στην MSA κάθε μικροϋπηρεσία διαθέτει ιδανικά τη δική της βάση δεδομένων, γεγονός που καθιστά δύσκολη την εξασφάλιση της ακεραιότητας και της συνέπειας των δεδομένων σε ολόκληρο το σύστημα. Σε αντίθεση με τις μονολιθικές αρχιτεκτονικές που βασίζονται σε ACID συναλλαγές εντός μιας κεντρικής βάσης δεδομένων, οι μικροϋπηρεσίες απαιτούν μηχανισμούς τελικής συνέπειας (eventual consistency), οι οποίοι προσθέτουν πολυπλοκότητα στον σχεδιασμό και την υλοποίηση του συστήματος.

2.2 Η Πρόκληση της Συνέπειας Δεδομένων στα Κατανεμημένα Συστήματα

Η μετάβαση στην αρχιτεκτονική μικροϋπηρεσιών, όπως αναφέρθηκε στην προηγούμενη ενότητα, σηματοδοτεί μια ριζική αλλαγή στη διαχείριση δεδομένων, καθώς μεταβαίνουμε από την απλότητα των μονολιθικών συστημάτων, που βασίζονται σε μια ενιαία βάση και στις ιδιότητες ACID για την εγγύηση της συνέπειας [23, 8], στο αποκεντρωμένο πρότυπο Database-per-Service στο οποίο κάθε υπηρεσία έχει τη δική της βάση δεδομένων [3]. Αν και αυτή η προσέγγιση, σύμφωνα με μια μελέτη, που χρησιμοποιείται από το 58% των συμμετεχόντων, προσφέρει σημαντικά οφέλη scalability και αυτονομίας, καταργεί τη δυνατότητα εκτέλεσης ατομικών συναλλαγών. Αυτό δημιουργεί σοβαρές προκλήσεις σχεδιασμού, με το 47% να αναγκάζονται να αναδιαμορφώσουν τις συναλλαγές τους ώστε να είναι συμβατές με τη διάσπαση σε μικροϋπηρεσίες, ενώ το 19% επιλέγει τη χρήση μιας κεντρικής βάσης δεδομένων για να αποφύγει την πολυπλοκότητα, θυσιάζοντας όμως την αυτονομία των μικροϋπηρεσιών [27].

Οι θεωρητικοί περιορισμοί της αρχιτεκτονικής μικροϋπηρεσιών περιγράφονται από το θεώρημα CAP, γνωστό και ως θεώρημα του Brewer. Η αρχική διατύπωση της κεντρικής ιδέας έγινε από τον Eric Brewer το 2000 [2], ενώ η μαθηματική του απόδειξη πραγματοποιήθηκε δύο χρόνια αργότερα [11]. Το θεώρημα αυτό, ορίζει ότι σε ένα κατανεμημένο σύστημα με ανοχή στον διαμερισμό (Partition Tolerance) είναι αδύνατη η ταυτόχρονη επίτευξη ισχυρής συνέπειας (Consistency) και διαθεσιμότητας (Availability). Αυτό διατυπώνεται καλύτερα στην εικόνα 2.1. Οι μικροϋπηρεσίες τυπικά ακολουθούν τη διαθεσιμότητα, με την παραδοχή ότι τα δεδομένα ενδέχεται να μην είναι ταυτόχρονα ενημερωμένα σε όλους τους κόμβους. Αυτό γίνεται προκειμένου να διασφαλιστεί η ανθεκτικότητα του συστήματος.

Ως απάντηση, χρησιμοποιείται το μοντέλο BASE (Basically Available, Soft state, Eventual consistency), που αντικαθιστά την άμεση συνέπεια με την τελική συνέπεια (Eventual Consistency). Το μοντέλο αυτό, εγγυάται πως τα δεδομένα θα συγκλίνουν σε μια συνεπή κατάσταση με την πάροδο του χρόνου [46]. Στην πράξη, το 42% των συμμετεχόντων, βάσει της έρευνας που αναφέρθηκε και προηγουμένως, βασίζεται σε στρατηγικές τελικής συνέπειας, αν και η πολυπλοκότητα οδηγεί το 30% σε ατομικές συναλλαγές όπου είναι εφικτό [27]. Η αποδοχή αυτής της προσωρινής ασυνέπειας καθιστά αναγκαία τη χρήση κάποιων μηχανισμών διαχείρισης συναλλαγών.



Σχήμα 2.1: Το διάγραμμα του θεωρήματος CAP που απεικονίζει τις τρεις ιδιότητες. Πηγή: Wikimedia Commons.

2.3 Μηχανισμοί Διαχείρισης Κατανεμημένων Συναλλαγών

2.3.1 Two-Phase Commit (2PC) και Saga Pattern

Το πρωτόκολλο Two-Phase Commit (2PC) αποτελεί την παραδοσιακή προσέγγιση για την επίτευξη ατομικότητας σε κατανεμημένα συστήματα, δίνοντας έμφαση στην ισχυρή συνέπεια (Strong Consistency). Ωστόσο, η φύση του 2PC, λόγω του σύγχρονου συντονισμού, δημιουργεί πολλά κλειδώματα μέχρι την ολοκλήρωση μιας συναλλαγής και δεσμεύει πολλούς πόρους. Σε αρχιτεκτονικές μικροϋπηρεσιών, αυτό οδηγεί στη δημιουργία μακροχρόνιων κλειδωμάτων (long-term locks), τα οποία ακυρώνουν τα οφέλη της αρχιτεκτονικής και μετατρέπουν το σύστημα σε έναν «κατανεμημένο μονόλιθο» [17].

Η σύγχρονη φύση του 2PC επηρεάζει αρνητικά την απόδοση του συστήματος. Διάφορα δεδομένα από έρευνες δείχνουν ότι η χρήση 2PC μπορεί να οδηγήσει σε σημαντικά χαμηλότερο throughput σε σύγκριση με ασύγχρονες λύσεις (17.33 TPS έναντι 25.45 TPS), καθώς και σε υψηλότερο μέσο χρόνο απόκρισης [46]. Επιπλέον, το 2PC είναι ιδιαίτερα απαιτητικό σε πόρους, με μελέτες να καταγράφουν χρήση CPU που αγγίζει το 100% υπό φόρτο, σε αντίθεση με το 85.4% των ασύγχρονων προσεγγίσεων [22]. Σε περιπτώσεις σφαλμάτων ή μερικής μη διαθεσιμότητας υπηρεσιών, η καθυστέρηση (latency) μπορεί να αυξηθεί πάνω από 150% [12].

Ως εναλλακτική λύση, ιδιαίτερα για συναλλαγές μεγάλης διάρκειας (Long Lived Transactions - LLTs), προτάθηκε το πρότυπο Saga από τους Garcia-Molina και Salem. Σε αντίθεση με το 2PC, το

Saga διαμερίζει μια συναλλαγή σε μια σειρά από υπο-συναλλαγές, οι οποίες απελευθερώνουν τους πόρους άμεσα μετά την εκτέλεση κάθε βήματος [10]. Επειδή δεν υπάρχει αυτόματη επαναφορά (rollback) όπως στις ACID συναλλαγές, το Saga διαχειρίζεται τις αποτυχίες εκτελώντας αντισταθμιστικές συναλλαγές (compensating transactions) για να αναιρέσει τις ενέργειες που έχουν ήδη πραγματοποιηθεί [10].

Αυτή η προσέγγιση ευθυγραμμίζεται με το θεώρημα CAP, τοποθετώντας το Saga στο μοντέλο AP (Availability και Partition Tolerance). Το Saga θυσιάζει την άμεση συνέπεια υπέρ της διαθεσιμότητας, επιτυγχάνοντας τελική συνέπεια (Eventual Consistency), σε αντίθεση με το 2PC που ακολουθεί το μοντέλο CP και αδρανοποιεί το σύστημα σε περιπτώσεις διαμερισμού δικτύου [46].

2.3.2 Προσεγγίσεις Saga: Choreography και Orchestration

Η υλοποίηση ενός Saga μπορεί να γίνει με δύο βασικές προσεγγίσεις, το choreography και το orchestration. Στην περίπτωση του choreography, η διαδικασία είναι αποκεντρωμένη και οι υπηρεσίες επικοινωνούν μέσω γεγονότων (events). Αν και αυτή η προσέγγιση είναι απλή για μικρές ροές, παρουσιάζει σημαντικά μειονεκτήματα καθώς αυξάνεται η πολυπλοκότητα. Η συμπεριφορά του συστήματος γίνεται δύσκολη στην κατανόηση, ενώ η απουσία κεντρικής παρακολούθησης καθιστά δύσκολο τον εντοπισμό σφαλμάτων και μειώνει την ορατότητα (visibility) της επιχειρησιακής λογικής [1].

Αντίθετα, στην προσέγγιση orchestration, υπάρχει ένας κεντρικός συντονιστής, ο λεγόμενος orchestrator που διαχειρίζεται τη ροή της εκτέλεσης των συναλλαγών. Αυτή η προσέγγιση συγκεντρώνει την επιχειρησιακή λογική σε ένα σημείο, καθιστώντας το σύστημα πιο εύκολο στη διαχείριση και την τροποποίηση, ειδικά για πολύπλοκες ροές [1]. Παρόλο που ο orchestrator εισάγει μια επιπλέον πολυπλοκότητα υποδομής και αποτελεί εν δυνάμει μοναδικό σημείο αποτυχίας (single point of failure), προσφέρει σημαντικά πλεονεκτήματα [19].

Συγκεκριμένα, η orchestration προσέγγιση, αφαιρεί την πολυπλοκότητα του συντονισμού από τις ίδιες τις υπηρεσίες, επιτρέποντας χαλαρή σύζευξη (loose coupling) σε επίπεδο λογικής, καθώς οι υπηρεσίες δεν χρειάζεται να γνωρίζουν η μία την άλλη, παρά μόνο να εκτελούν εντολές του Orchestrator [19]. Για τους λόγους αυτούς, αυτή η προσέγγιση έχει επιλεχθεί στην παρούσα εργασία ως η στρατηγική για τη διαχείριση της ροής των συναλλαγών.

2.4 Διαχείριση των Καταστάσεων στην Orchestration προσέγγιση

2.4.1 Ο Ρόλος του Orchestrator

Ο Saga Orchestrator, γνωστός και ως Saga Execution Coordinator (SEC), λειτουργεί ως ο κεντρικός «εγκέφαλος» που διαχειρίζεται τη ροή των κατανεμημένων συναλλαγών. Σε αντίθεση με έναν stateless proxy, το οποίο απλώς προωθεί αιτήματα χωρίς να διατηρεί την κατάσταση της διαδικασίας, ο SEC διατηρεί την τρέχουσα κατάσταση της ροής για να διασφαλίσει τη συνέπεια των δεδομένων και την ανθεκτικότητα [17, 19].

Η διατήρηση της κατάστασης της συναλλαγής επιτυγχάνεται μέσω συγκεκριμένων μηχανισμών. Ο orchestrator συνήθως έχει υλοποιηθεί ως μια μηχανή καταστάσεων, για την καλύτερη παρακολούθηση των βημάτων που έχουν εκτελεστεί και αυτών που εκκρεμούν [24]. Για την παρακολούθηση του κύκλου ζωής ενός saga, ο orchestrator αποθηκεύει την κατάσταση στη δική του βάση δεδομένων, καταγράφοντας την ακολουθία των γεγονότων και των αποφάσεων που λαμβάνονται [17].

Η διατηρούμενη κατάσταση (persisted state) είναι κρίσιμη για την ανθεκτικότητα του συστήματος. Επιτρέπει την ανάκτηση σε περίπτωση κατάρρευσης του συστήματος (crash recovery), καθώς χωρίς αυτήν θα χανόταν το ιστορικό του Saga, καθιστώντας αδύνατη την ολοκλήρωση ή την αντιστάθμιση εκκρεμών συναλλαγών [19]. Είναι επίσης απαραίτητη για τη διαχείριση διεργασιών μεγάλης διάρκειας και παρέχει δυνατότητα ελέγχου (auditability), επιτρέποντας την κατανόηση του σημείου αποτυχίας και της αιτίας ενεργοποίησης μιας αντισταθμιστικής ενέργειας.

2.4.2 Μηχανές Πεπερασμένων Καταστάσεων

Οι Μηχανές Πεπερασμένων Καταστάσεων (Finite State Machines - FSM) χρησιμοποιούνται στην προσέγγιση orchestration των συναλλαγών Saga για τη μοντελοποίηση της συμπεριφοράς και της προόδου των κατανεμημένων συναλλαγών. Σε αυτό το πλαίσιο, ο orchestrator χρησιμοποιεί τη μηχανή καταστάσεων για να παρακολουθεί την κατάσταση της ροής, καθορίζοντας ποια βήματα έχουν ολοκληρωθεί και αποφασίζοντας τις επόμενες ενέργειες βάσει των αποτελεσμάτων [24].

Οι βασικές έννοιες των FSM στις συναλλαγές Saga ορίζονται ως εξής:

- **Καταστάσεις (States):** Αντιπροσωπεύουν την τρέχουσα κατάσταση της συναλλαγής. Ένα απλοποιημένο μοντέλο θεωρεί μόνο τις καταστάσεις δεδομένων ως καταστάσεις της μηχανής (π.χ. "Transaction Started", "Hotel Reserved").
- **Γεγονότα (Events):** Είναι τα εξωτερικά ερεθίσματα που πυροδοτούν μια μετάβαση της μηχανής καταστάσεων. Συνήθως πρόκειται για τις απαντήσεις από τις μικροϋπηρεσίες, που υποδεικνύουν αν μια τοπική συναλλαγή ήταν επιτυχής (commit) ή ανεπιτυχής (rollback).
- **Μεταβάσεις (Transitions):** Περιγράφουν τη λογική μετάβασης από μια κατάσταση σε μια άλλη ως απάντηση σε ένα γεγονός. Στην περίπτωση επιτυχούς γεγονότος, η μετάβαση οδηγεί στην επόμενη κατάσταση και ενεργοποιεί την επόμενη τοπική συναλλαγή. Στην περίπτωση γεγονότος αποτυχίας, η μετάβαση οδηγεί προς μια κατάσταση ακύρωσης, ενεργοποιώντας τις αντισταθμιστικές συναλλαγές.

Για πολύπλοκα Sagas, αυτές οι μηχανές καταστάσεων μπορεί να γίνουν σύνθετες, καθώς πρέπει να λαμβάνουν υπόψη διάφορους συνδυασμούς γεγονότων επιτυχίας και αποτυχίας για τη διασφάλιση της συνέπειας των δεδομένων [24]. Διάφορα frameworks, όπως το Seata, βασίζονται ρητά σε μηχανές καταστάσεων για τη διαχείριση αυτών των καταστάσεων [17].

2.5 Υπάρχοντα Saga Frameworks

Στα συστήματα μικροϋπηρεσιών που έχουν αναπτυχθεί σε περιβάλλον Spring, υπάρχουν αρκετά καθιερωμένα frameworks που προσφέρουν έτοιμες λύσεις για την υλοποίηση του προτύπου Saga. Ένα από τα πιο γνωστά είναι το Axon Framework, το οποίο υποστηρίζει τόσο την προσέγγιση orchestration όσο και την choreography. Είναι ιδιαίτερα σχεδιασμένο για συστήματα που ακολουθούν τα πρότυπα CQRS (Command Query Responsibility Segregation) και Event Sourcing. Το Axon απλοποιεί την υλοποίηση, διαχωρίζοντας την επιχειρισιακή λογική από τις τεχνικές λεπτομέρειες του Saga [17].

Το Eventuate Tram είναι μια άλλη δημοφιλής λύση που βασίζεται στο Apache Kafka για την ανταλλαγή μηνυμάτων και χρησιμοποιεί το Transactional Outbox Pattern για να εγγυηθεί την ατομικότητα μεταξύ ενημερώσεων κατάστασης και δημοσίευσης γεγονότων. Προσφέρει ένα μοντέλο, που το ονομάζει builder model, για τον ορισμό των Sagas, που χρησιμοποιείται για την δήλωση ενεργειών και αντισταθμιστικών διαδικασιών, αν και αυτό μπορεί να οδηγήσει σε αυξημένες γραμμές κώδικα. Επίσης, το Seata, το οποίο χρησιμοποιείται ευρέως από την Alibaba, βασίζεται σε μια μηχανή καταστάσεων για τη διαχείριση κατανεμημένων συναλλαγών και προσφέρει υψηλή απόδοση και scalability [17].

Άλλα αξιοσημείωτα εργαλεία περιλαμβάνουν το Netflix Conductor, μια μηχανή ενορχήστρωσης που χρησιμοποιεί JSON-based DSL για τον ορισμό της λογικής και παρέχει γραφικό περιβάλλον για την παρακολούθηση της εκτέλεσης, καθώς και το Apache Camel που υποστηρίζει το Saga pattern ως integration pattern [17]. Τέλος, βιβλιοθήκες όπως το Anser-Saga εστιάζουν στην τελική συνέπεια, παρέχοντας μηχανισμούς backup για την αποθήκευση της κατάστασης της συναλλαγής και τη δημιουργία σημείων επανεκκίνησης σε περίπτωση αποτυχίας [18].

2.6 Ο Σκοπός της Παρούσας Υλοποίησης

Παρόλο που τα παραπάνω frameworks προσφέρουν ισχυρές δυνατότητες και θεωρούνται έτοιμα για χρήση σε περιβάλλοντα παραγωγής (production environments), η χρήση τους σε μικρότερα έργα ή για εκπαιδευτικούς σκοπούς παρουσιάζει σημαντικά μειονεκτήματα. Τα frameworks αυτά συχνά επιβάλλουν συγκεκριμένα αρχιτεκτονικά πρότυπα, όπως το CQRS στην περίπτωση του Axon, τα οποία μπορεί να προσθέσουν αχρείαστη πολυπλοκότητα σε απλούστερες εφαρμογές που δεν ακολουθούν αυτά τα πρότυπα. Επιπλέον, η υλοποίηση των Sagas σε cloud-native περιβάλλοντα με χρήση τέτοιων εργαλείων μπορεί να αυξήσει την πολυπλοκότητα της υποδομής και να οδηγήσει σε vendor lock-in, καθιστώντας δύσκολη τη μελλοντική μετάβαση σε άλλα εργαλεία [17].

Ένα ακόμη σημαντικό ζήτημα είναι το λεγόμενο black-box αυτών των frameworks. Όπως επισημαίνεται στη βιβλιογραφία, στα επαγγελματικά frameworks δεν είναι ξεκάθαρες οι σημαντικές λεπτομέρειες της διαχείρισης συναλλαγών, καθιστώντας τα ακατάλληλα για την κατανόηση των μηχανισμών. Για κάποιον που θέλει να κατανοήσει την πραγματική ουσία του προτύπου, η χρήση ενός framework που αυτοματοποιεί τα περισσότερα κομμάτια μπορεί να μην είναι τόσο χρήσιμη, καθώς δεν του επιτρέπει να δει πώς ακριβώς διαχειρίζεται το σύστημα τα διάφορα γεγονότα και τις

αντισταθμιστικές ενέργειες [23].

Για τους λόγους αυτούς, στην παρούσα εργασία επιλέχθηκε η υλοποίηση ενός πολύ απλού Saga Orchestrator με τη χρήση της βιβλιοθήκης Spring Statemachine. Παρόλο που το Spring Statemachine είναι ένα διαδεδομένο εργαλείο για τη γενική διαχείριση καταστάσεων, δεν βρέθηκαν στη βιβλιογραφία απλές υλοποιήσεις που να το εφαρμόζουν σε περιβάλλον Spring Boot. Συγκεκριμένα, η παρούσα εργασία εστιάζει στον σχεδιασμό της μηχανής καταστάσεων για τον χειρισμό των μεταβάσεων και των αντισταθμιστικών ενεργειών. Τέλος, παρουσιάζονται κάποια σενάρια ως μέρος της αξιολόγησης της υλοποίησης.

3. Σχεδίαση Συστήματος

3.1 Επισκόπηση Αρχιτεκτονικής Συστήματος

Αυτή η ενότητα έχει ως σκοπό να παρουσιάσει την αρχιτεκτονική του συστήματος, το οποίο θα αποτελέσει τη βάση για την εφαρμογή του Saga Orchestrator. Αναλυτικότερα, η υλοποίηση βασίζεται σε ένα σύστημα ηλεκτρονικού εμπορίου (e-commerce) και εστιάζει στη ροή δημιουργίας παραγγελίας προϊόντων. Η δημιουργία παραγγελίας είναι μια ενέργεια που συναντάται σχεδόν σε κάθε e-commerce και αποτελείται από αρκετές επιμέρους ενέργειες. Αυτές οι ενέργειες συνήθως αποτελούνται από την δέσμευση του ποσού χρέωσης του πελάτη, την δέσμευση του αποθέματος των προϊόντων, την χρέωση του πελάτη, την δημιουργία δελτίου αποστολής κ.ά. Στην παρούσα εργασία θα εστιάσουμε στο σενάριο που αποτελείται από τα εξής βήματα:

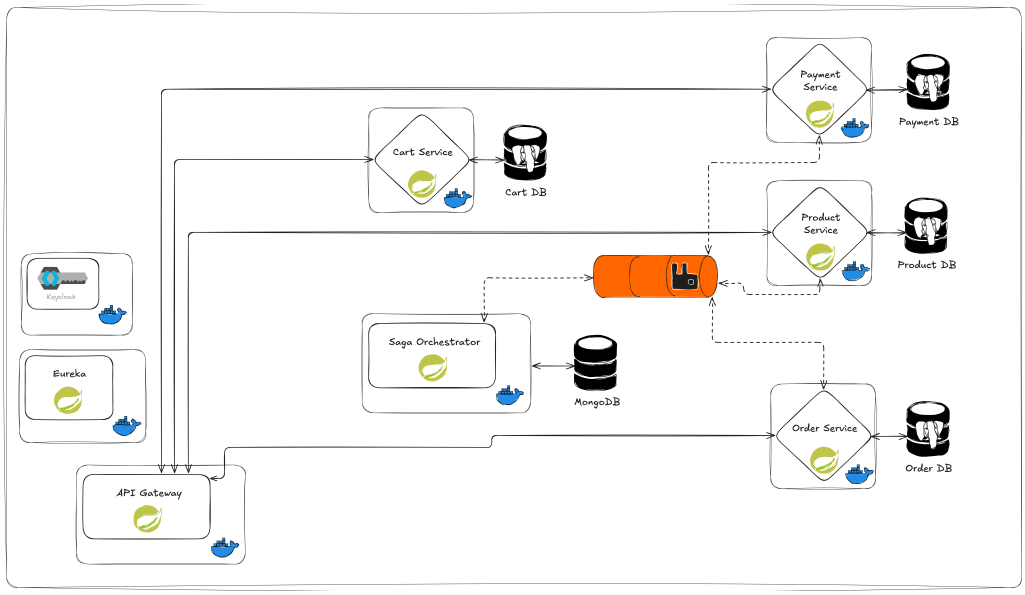
1. Δημιουργία νέας παραγγελίας
2. Δέσμευση του συνολικού ποσού χρέωσης από τον πελάτη
3. Δέσμευση του αποθέματος των προϊόντων
4. Χρέωση του πελάτη
5. Ολοκλήρωση παραγγελίας

Η αρχιτεκτονική του συστήματος είναι βασισμένη στις μικροϋπηρεσίες και κάθε υπηρεσία έχει ανεπτυχθεί σε Spring Boot [33]. Το Spring Boot είναι ένα εργαλείο του Spring framework, το οποίο παρέχει έτοιμες βιβλιοθήκες που είναι συμβατές μεταξύ τους και βοηθά στην γρήγορη ανάπτυξη εφαρμογών Java. Τα επιμέρους κομμάτια του συστήματος αποτελούνται από τα API Gateway, Eureka Server, Saga Orchestrator καθώς και τις υπηρεσίες Product, Cart, Order και Payment, οι οποίες εφεξής θα αναφέρονται συνολικά ως εσωτερικές υπηρεσίες. Επιπλέον, το σύστημα ενσωματώνει κάποιες τρίτες υπηρεσίες όπως το RabbitMQ [31], που χρησιμοποιείται ως message broker για την ασύγχρονη ανταλλαγή μηνυμάτων και το Keycloak [16] που χρησιμοποιείται ως η κεντρική υπηρεσία αυθεντικοποίησης των χρηστών.

Η επικοινωνία των υπηρεσιών πραγματοποιείται είτε συγχρονισμένα μέσω HTTP είτε ασύγχρονα μέσω queues του RabbitMQ. Η ασύγχρονη επικοινωνία χρησιμοποιείται για ενέργειες όπου η υπηρεσία που στέλνει το μήνυμα δεν χρειάζεται άμεση απάντηση. Αυτό επιτρέπει στις υπηρεσίες να επεξεργάζονται τα αιτήματα σε δικό τους χρόνο με αποτέλεσμα να μειώνεται η επιβάρυνση του συστήματος που θα προκαλούσαν οι πολλές ταυτόχρονες HTTP κλήσεις.

Κάθε εσωτερική υπηρεσία έχει τη δική της βάση δεδομένων στο ΣΔΒΔ PostgreSQL [29], ακολουθώντας το πρότυπο database per service. Ο Saga Orchestrator χρησιμοποιεί MongoDB [20] για την αποθήκευση των μηχανών καταστάσεων. Η επιλογή της PostgreSQL έγινε λόγω της αξιοπιστίας της

για την διαχείριση πολύπλοκων δεδομένων με ασφάλεια και συνέπεια ενώ της MongoDB οφείλεται στην καταλληλότητά της για την απλή και γρήγορη αποθήκευση και ανάκτηση των δεδομένων. Το διάγραμμα της αρχιτεκτονικής του συστήματος απεικονίζεται στην εικόνα 3.1. Είναι σημαντικό να σημειωθεί, πως για λόγους απλότητας, δεν έχουν συμπεριληφθεί συνδέσεις μεταξύ των υπηρεσιών για την ενδοεπικοινωνία τους, ούτε για την επικοινωνία τους με τα Eureka και Keycloak.



Σχήμα 3.1: Διάγραμμα Αρχιτεκτονικής Συστήματος

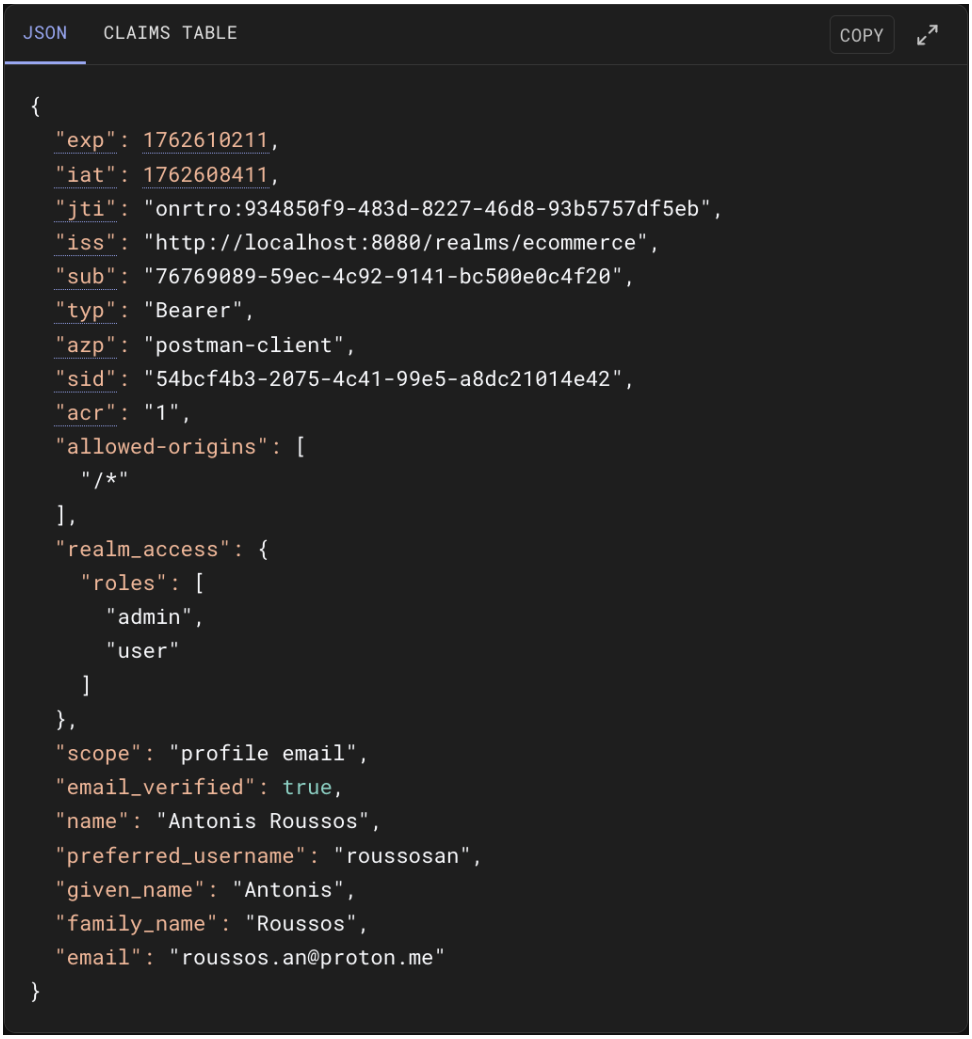
Στη συνέχεια θα αναλυθούν ένα προς ένα τα επιμέρους τμήματα του συστήματος που χρησιμοποιήθηκαν στο πλαίσιο αυτής της πτυχιακής εργασίας. Συγκεκριμένα, θα αναφερθούν αναλυτικότερα οι τεχνολογίες που χρησιμοποιήθηκαν και η λογική του διαχωρισμού των υλοποιημένων μικροϋπηρεσιών.

3.1.1 Keycloak

Το Keycloak [16] είναι ένα λογισμικό ανοιχτού κώδικα που έχει υλοποιηθεί με κύριο σκοπό την κεντρική διαχείριση της αυθεντικοποίησης και εξουσιοδότησης των χρηστών. Υποστηρίζει πολλά πρωτόκολλα όπως τα OpenID, OAuth2 και SAML. Επιπλέον, παρέχει δυνατότητες για two-factor authentication καθώς και διαχείριση χρηστών και ρόλων μέσω της web πλατφόρμας του.

Ο λόγος που το επέλεξα ήταν η ανάγκη για την συγκέντρωση όλης της λογικής της αυθεντικοποίησης σε ένα κεντρικό σημείο, αξιοποιώντας ένα εργαλείο που είναι δημοφιλές και αρκετά δοκιμασμένο σε μεγάλα συστήματα. Με αυτόν τον τρόπο, απέφυγα να υλοποιήσω από το μηδέν τη λογική αυθεντικοποίησης σε κάποιο δικό μου service. Το Keycloak παρέχει έτοιμα authentication flows για την αυθεντικοποίηση χρηστών και κάνει χρήση των JWTs ως μηχανισμό μεταφοράς της πληροφορίας των χρηστών. Τα JWT tokens περιέχουν τις απαραίτητες πληροφορίες που

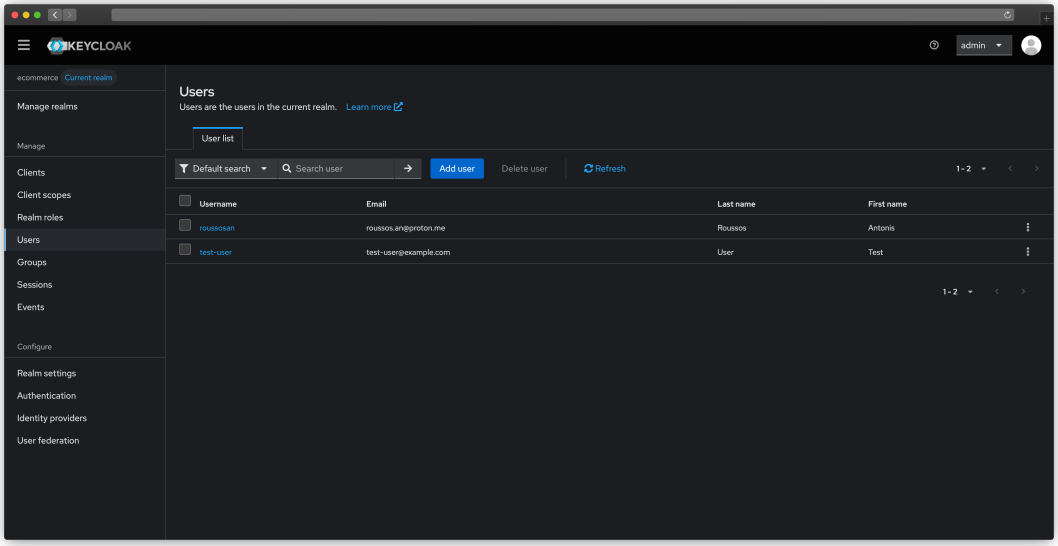
χρησιμοποιούν οι επιμέρους μικροϋπηρεσίες για να εφαρμόσουν ελέγχους εξουσιοδότησης ανά endpoint, για να επιτρέψουν ή να απορρίψουν την πρόσβαση στους χρήστες. Για παράδειγμα, ένα JWT που έχει εκδόσει το Keycloak φαίνεται στην εικόνα 3.2.



Σχήμα 3.2: Παράδειγμα JWT που έχει εκδόσει το Keycloak.

Η ενσωμάτωση του Keycloak στο σύστημα πραγματοποιήθηκε μέσω του Docker [6] και το Keycloak υπάρχει σε ένα docker container. Επίσης, το configuration ήταν αρκετά εύκολο χάρη στη χρήση του Keycloak Admin Console, όπως φαίνεται στην εικόνα 3.3. Στο πλαίσιο της υλοποίησης, δημιούργησα ένα realm με όνομα ecommerce και έναν client για να μπορούν να πραγματοποιηθούν οι απαραίτητες δοκιμές. Οι ρόλοι που ορίζονται στο σύστημα αποτελούνται από τους user και admin.

Συνοψίζοντας, το Keycloak αναλαμβάνει την κεντρική διαχείριση της αυθεντικοποίησης στο σύστημα ενώ οι μικροϋπηρεσίες αναλαμβάνουν την εξουσιοδότηση των χρηστών με χρήση των JWTs. Με αυτόν τον τρόπο, διασφαλίζεται η επεκτασιμότητα και η ασφάλεια των μικροϋπηρεσιών.



Σχήμα 3.3: Το web dashboard του Keycloak.

3.1.2 Netflix Eureka

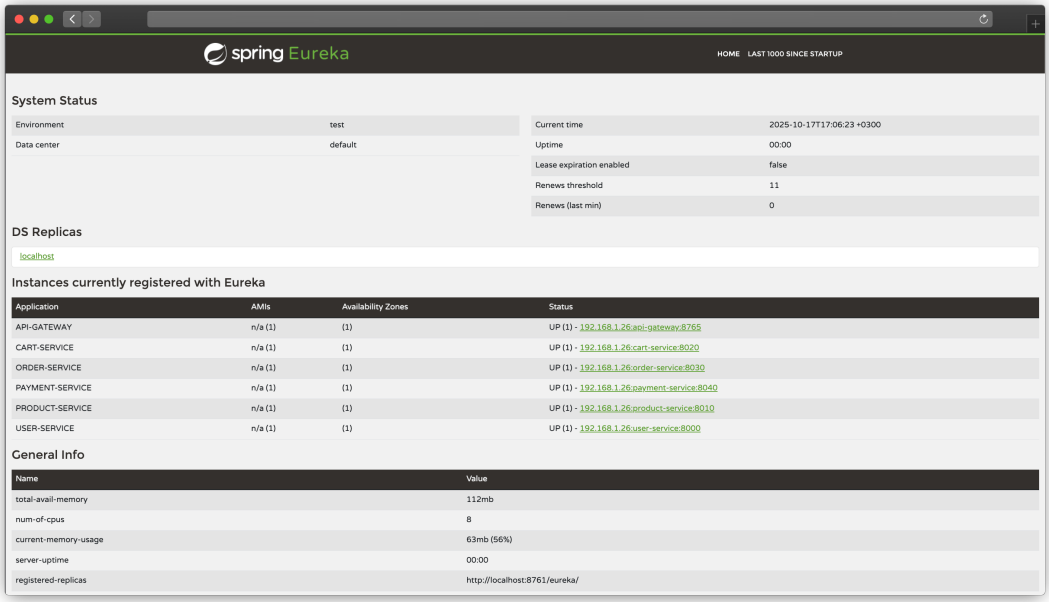
Το Netflix Eureka [25] είναι ένα λογισμικό που έχει αναπτυχθεί από την ομάδα του Netflix και παρέχει λειτουργικότητα για service discovery και load balancing σε αρχιτεκτονικές μικροϋπηρεσιών. Σε τέτοια συστήματα, υπάρχουν πολλές αυτόνομες υπηρεσίες που χρειάζεται να επικοινωνούν μεταξύ τους, συχνά μέσω HTTP. Κάθε υπηρεσία μπορεί να διαθέτει πολλαπλά instances, με αποτέλεσμα αυτό να καθιστά πρόβλημα στην επικοινωνία όταν ο αριθμός των instances αρχίζει να αυξάνεται. Το Eureka επιλύει αυτό το πρόβλημα, διατηρώντας ένα registry όλων των ενεργών υπηρεσιών που έχουν εγγραφεί σε αυτό. Κάθε υπηρεσία αποστέλλει περιοδικά heartbeats, τα οποία είναι κάποια αιτήματα που δηλώνουν ότι η υπηρεσία είναι ακόμη ενεργή. Όλες οι υπηρεσίες μπορούν να ανακτούν δυναμικά τις διευθύνσεις των διαθέσιμων instances μέσω του registry.

Το service discovery είναι πολύ σημαντικό, καθώς επιτρέπει στις υπηρεσίες να βρίσκουν η μία την άλλη και συμβάλλει στο load balancing μεταξύ των ενεργών instances. Έτσι, βελτιώνεται η επεκτασιμότητα και η ανθεκτικότητα του συστήματος.

Ο λόγος που επέλεξα το Netflix Eureka ήταν η ανάγκη για δυναμική ανακάλυψη και δρομολόγηση μεταξύ των μικροϋπηρεσιών χωρίς την χρήση στατικών διευθύνσεων. Με αυτόν τον τρόπο, αποφεύγεται η ανάγκη χειροκίνητης ενημέρωσης των διευθύνσεων κάθε φορά που μια υπηρεσία αλλάζει IP ή port.

Στο πλαίσιο της παρούσας υλοποίησης, το Eureka έχει αναπτυχθεί ως μια ανεξάρτητη εφαρμογή Spring Boot με χρήση της βιβλιοθήκης Spring Cloud Netflix Eureka Server [37]. Κάθε εσωτερική υπηρεσία ενσωματώνει την βιβλιοθήκη Eureka Client [36] και εγγράφεται αυτόματα στο registry κατά την εκκίνηση. Η αποστολή των heartbeats πραγματοποιείται αυτόματα από το framework ώστε το registry να παραμένει πάντα ενημερωμένο.

Για την παρακολούθηση των υπηρεσιών, ο Eureka Server παρέχει ένα web περιβάλλον διαχείρισης, όπου εμφανίζονται όλα τα εγγεγραμμένα instances και η κατάστασή τους (UP/DOWN), όπως φαίνεται στην εικόνα 3.4.



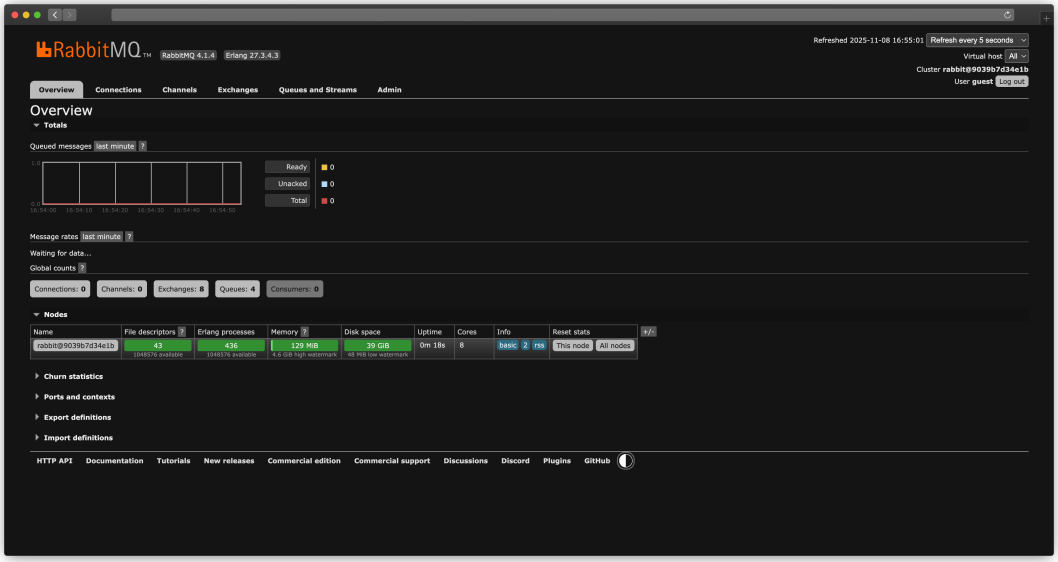
Σχήμα 3.4: Προβολή όλων των εγγεγραμμένων microservices και της κατάστασής τους στο Eureka Server dashboard.

Συνοψίζοντας, το Netflix Eureka παρέχει έναν αξιόπιστο τρόπο για το service discovery και το load balancing, τα οποία είναι πολύ σημαντικά σε συστήματα που έχουν ως σκοπό να είναι επεκτάσιμα, όπως οι μικροϋπηρεσίες.

3.1.3 RabbitMQ

Το RabbitMQ [31] είναι ένα message broker ανοιχτού κώδικα που υλοποιεί το πρωτόκολλο AMQP (Advanced Message Queuing Protocol). Η βασική του λειτουργία είναι η ασύγχρονη ανταλλαγή μηνυμάτων μεταξύ των υπηρεσιών. Τα μηνύματα αποστέλλονται σε queues μέσω exchanges, τα οποία καθορίζουν τη δρομολόγηση τους βάσει κάποιων κανόνων. Η χρήση του RabbitMQ επιλέχθηκε για να υποστηρίξει την ασύγχρονη επικοινωνία ανάμεσα στις υπηρεσίες που συμμετέχουν στη ροή του saga.

Η ενσωμάτωση πραγματοποιήθηκε μέσω Docker, ώστε το RabbitMQ να εκτελείται σε ξεχωριστό container. Η παρακολούθηση των queues και των exchanges γίνεται μέσω του web UI που παρέχει, όπως φαίνεται στην εικόνα 3.5.



Σχήμα 3.5: Προβολή του RabbitMQ dashboard.

3.1.4 PostgreSQL

Το PostgreSQL [29] είναι ένα σχεσιακό σύστημα διαχείρισης βάσεων δεδομένων που είναι πολύ αποδοτικό σε πολύπλοκα queries και εξασφαλίζει ACID ιδιότητες. Είναι αξιόπιστο και κατάλληλο για συστήματα που απαιτούν συνέπεια δεδομένων.

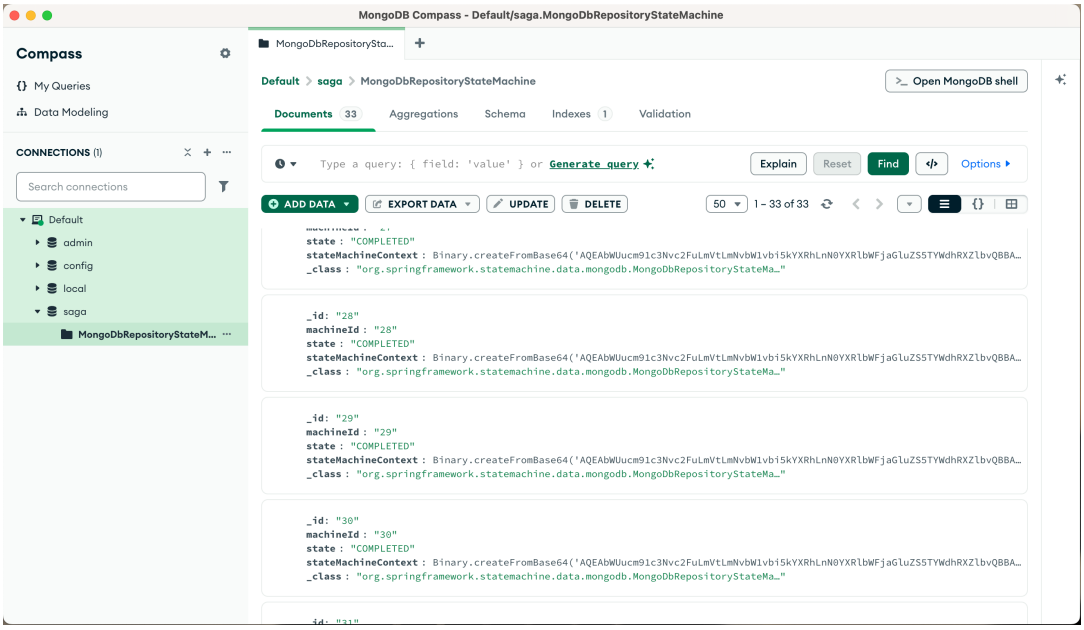
Στο σύστημά μου, κάθε εσωτερική υπηρεσία διαθέτει το δικό της schema σε PostgreSQL. Το postgreSQL τρέχει ως docker container και κάθε service συνδέεται στο δικό του schema με χρήση της βιβλιοθήκης Spring Data JPA [38] και του Hibernate ORM [13]. Για λόγους απλότητας, στη παρούσα πτυχιακή εργασία, υπάρχει ένα PostgreSQL instance με διαφορετικά schemas για κάθε μικροϋπηρεσία, αντί για ένα PostgreSQL instance ανά μικροϋπηρεσία.

3.1.5 MongoDB

Η MongoDB [20] είναι μια NoSQL βάση δεδομένων που χρησιμοποιείται για την αποθήκευση μη δομημένων ή ημι-δομημένων δεδομένων. Ο Saga Orchestrator χρησιμοποιεί τη MongoDB για την αποθήκευση των μηχανών καταστάσεων και επικοινωνεί μαζί της μέσω της βιβλιοθήκης Spring Data MongoDB [39]. Η MongoDB τρέχει επίσης μέσω Docker και παρακολουθείται με το MongoDB Compass [21], όπως φαίνεται στην εικόνα 3.6.

3.1.6 API Gateway

Το API Gateway είναι η κεντρική δημόσια διεπαφή της πλατφόρμας. Κάθε εξωτερικό HTTP αίτημα από κάποιον client περνάει αρχικά από το API Gateway. Έπειτα, το αίτημα συνήθως περνάει



Σχήμα 3.6: Το MongoDB Compass client.

από κάποια επικύρωση και διάφορους ελέγχους και μετά επαναδρομολογείται στην αντίστοιχη εσωτερική μικροϋπηρεσία. Σε αυτή την εργασία, για λόγους απλότητας, το gateway δρομολογεί όλα τα αιτήματα στην αντίστοιχη μικροϋπηρεσία χωρίς να πραγματοποιεί κάποιον έλεγχο. Ακόμα και τον έλεγχο της αυθεντικοποίησης τον επιχειρούν οι ίδιες οι μικροϋπηρεσίες κι όχι το API Gateway, καθώς αυτό δεν έχει καμία άλλη λογική πέρα από την δρομολόγηση των αιτημάτων εσωτερικά στο σύστημα και την επιστροφή της απάντησης προς τα έξω.

Το API Gateway συνεργάζεται με το Eureka Server για το κομμάτι του service discovery με αποτέλεσμα να μην αποθηκεύει στατικές διευθύνσεις ή ports. Στέλνει ένα ερώτημα στο Eureka για τα ενεργά instances κάποιας μικροϋπηρεσίας τη στιγμή που λαμβάνει κάποιο αίτημα. Αν, για παράδειγμα, κάποιο νέο instance του Product Service σταματήσει να στέλνει heartbeats τότε θα σβηστεί από το registry του Eureka και το gateway θα σταματήσει να δρομολογεί τα αιτήματα σε αυτό. Αυτή η συμπεριφορά μειώνει το ποσοστό σφαλμάτων του συστήματος και βελτιώνει την ανθεκτικότητα. Επειδή το gateway έχει κατασκευαστεί με τη βιβλιοθήκη Spring Cloud Gateway [34], χρησιμοποιεί ένα reactive μοντέλο που το βοηθά να χειρίζεται πολλές ταυτόχρονες συνδέσεις καταναλώνοντας λίγους πόρους. Επίσης, χρησιμοποιεί το Spring Cloud LoadBalancer [35] ώστε να διανέμει τα αιτήματα μεταξύ των ενεργών instances. Η δρομολόγηση των αιτημάτων γίνεται βάσει URL, δηλαδή, ένα αίτημα του τύπου `/api/cart/*` θα απευθυνθεί στο Cart Service ενώ ένα αίτημα τύπου `/api/products/*` αφορούν το Product Service.

Σχετικά με την ασφάλεια, έχει υλοποιηθεί με τρόπο που να καλύπτει την παρούσα εργασία. Το API Gateway προωθεί όλα τα αιτήματα στις εσωτερικές υπηρεσίες χωρίς να ελέγχει την ύπαρξη έγκυρου JWT ή να ελέγχει άλλες τιμές των headers. Η λογική των αποφάσεων εξουσιοδότησης για

το κάθε endpoint βρίσκεται σε κάθε μικροϋπηρεσία ξεχωριστά.

3.2 Περιγραφή Μικροϋπηρεσιών

3.2.1 Product Service

Το Product Service είναι υπεύθυνο για την διαχείριση των προϊόντων, των κατηγοριών και του αποθέματος. Έχει το δικό του PostgreSQL schema και εκθέτει κάποια RESTful API endpoints ώστε το API Gateway και οι άλλες μικροϋπηρεσίες να μπορούν να ανακτήσουν ή να τροποποιήσουν τα δεδομένα των προϊόντων και των κατηγοριών μέσω HTTP.

Η υπηρεσία εκθέτει endpoints για προϊόντα και κατηγορίες. Τα προϊόντα έχουν ξεχωριστό SKU ως μοναδικό κλειδί. Υπάρχει επίσης ένα HEAD endpoint που επιτρέπει σε άλλες υπηρεσίες να ελέγχουν γρήγορα την ύπαρξη ενός προϊόντος χωρίς να μεταφέρεται άσκοπα το περιεχόμενο. Στα endpoints εγγραφής πραγματοποιείται έλεγχος ώστε να μην δημιουργηθεί προϊόν με διπλό SKU και αντίστοιχος έλεγχος πριν την αλλαγή του SKU. Κατά την διαγραφή προϊόντος υπάρχει έλεγχος όπου η διαδικασία απορρίπτεται αν το προϊόν έχει ενεργές κρατήσεις αποθέματος ώστε να μην χαθούν τα δεσμευμένα τεμάχια που σχετίζονται με παραγγελίες σε εξέλιξη. Αντίστοιχα, για τις κατηγορίες γίνεται έλεγχος μοναδικότητας ονόματος. Τα endpoints φαίνονται στον πίνακα 3.1.

Το σχήμα της βάσης δεδομένων περιλαμβάνει τέσσερις κύριους πίνακες: product, category, inventory, inventory_reservation. Ο πίνακας product σχετίζεται με πληροφορία που αφορά τα προϊόντα και αποτελείται από τα αναγνωριστικό (id), ονομασία (name), περιγραφή (description), SKU, τιμή (price), ημερομηνία δημιουργίας (created_at), ημερομηνία επεξεργασίας (updated_at) καθώς και τη συσχέτιση με τον πίνακα category (category_id). Ο πίνακας category είναι πιο απλός και περιέχει μόνο τα πεδία id, name, created_at, updated_at. Στον πίνακα inventory αποθηκεύεται η πληροφορία του αποθέματος με τα πεδία id, ποσότητα αποθέματος (stock_quantity), πεδίο συσχέτισης με το αντίστοιχο προϊόν (product_id), version για την υποστήριξη optimistic locking ώστε να μην υπάρξουν συγκρούσεις σε ταυτόχρονες ενημερώσεις του αποθέματος, καθώς και τα created_at και updated_at. Αντί για μια στήλη reserved_quantity μέσα στον inventory πίνακα, η δέσμευση του αποθέματος γίνεται με ξεχωριστό πίνακα inventory_reservation που περιέχει τα id, id συσχέτισης με μια πραγγελία (order_id), πεδίο συσχέτισης με ένα προϊόν (product_sku), quantity, idempotency_id και created_at. Η χρήση του idempotency_id θα εξηγηθεί στην υποενότητα 4.2.3. Η συσχέτιση των προϊόντων με τις κατηγορίες είναι ένα-προς-πολλά ενώ με το απόθεμα είναι ένα-προς-ένα. Το schema της βάσης δεδομένων απεικονίζεται στην εικόνα 3.7. Η απόφαση να μοντελοποιηθεί η δέσμευση του αποθέματος ως μια ξεχωριστή οντότητα, αντί απλά να είναι ενσωματωμένο στον πίνακα inventory, επιλέγχθηκε επειδή επιτρέπει μια πιο καθαρή προσέγγιση στη διαχείριση του αποθέματος.

Η υπηρεσία συμμετέχει στη saga ροή της δημιουργίας παραγγελίας. Συγκεκριμένα, ακούει για inventory commands από τον orchestrator και έχει έναν Saga Command Listener ο οποίος είναι υπεύθυνος στο να λαμβάνει αυτά τα commands και να εκτελεί την αντίστοιχη λογική. Περισσότερα σχετικά με τον listener του Product Service θα αναφερθούν στην υποενότητα 4.2.3.

Πίνακας 3.1: Τα HTTP endpoints του Product Service

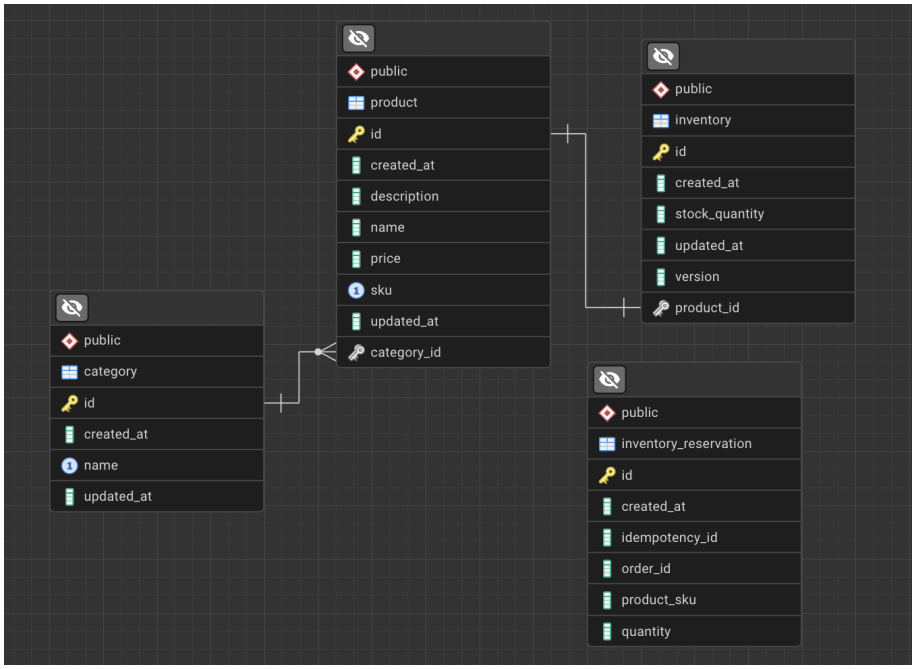
Endpoint	Σκοπός	Ρόλος
APIs Διαχείρισης Προϊόντων		
GET /api/products	Ανάκτηση λίστας προϊόντων (με pagination)	--
GET /api/products/{sku}	Ανάκτηση συγκεκριμένου προϊόντος	--
GET /api/products/bulk	Ανάκτηση συγκεκριμένων προϊόντων	--
HEAD /api/products/{sku}	Έλεγχος ύπαρξης προϊόντος	--
POST /api/products	Δημιουργία νέου προϊόντος	admin
PUT /api/products/{sku}	Ενημέρωση προϊόντος	admin
DELETE /api/products/{sku}	Διαγραφή προϊόντος	admin
APIs Διαχείρισης Κατηγοριών		
GET /api/categories	Ανάκτηση λίστας κατηγοριών	--
GET /api/categories/{id}	Ανάκτηση συγκεκριμένης κατηγορίας	--
POST /api/categories	Δημιουργία νέας κατηγορίας	admin
PUT /api/categories/{id}	Ενημέρωση κατηγορίας	admin
DELETE /api/categories/{id}	Διαγραφή κατηγορίας	admin

Σχετικά με την ασφάλεια, το service λειτουργεί ως ένας OAuth2 Resource Server [28] και χρησιμοποιεί τα JWT που εκδίδει το Keycloak για τον έλεγχο εξουσιοδότησης. Όπως φαίνεται και στον πίνακα 3.1, τα GET, HEAD endpoints είναι δημόσια και δεν χρειάζεται κάποια εξουσιοδότηση για να έχει ο χρήστης πρόσβαση. Αντιθέτως, τα POST, PUT, DELETE endpoints χρειάζονται τον ρόλο του διαχειριστή (admin) του συστήματος καθώς αφορούν ενέργειες που τροποποιούν τα δεδομένα στην βάση δεδομένων. Επίσης, πραγματοποιείται έλεγχος εγκυρότητας των δεδομένων που εισάγει ο χρήστης σε αυτές τις ενέργειες, ώστε να μην υπάρξει εισαγωγή μη έγκυρων δεδομένων.

Συνολικά, το Product Service προσφέρει έναν διαχωρισμό της λογικής σχετικά με τα προϊόντα, τις κατηγορίες και το απόθεμα των προϊόντων. Υπάρχουν διάφοροι απλοί έλεγχοι εγκυρότητας των δεδομένων στα endpoints οι οποίοι διατηρούν την εγκυρότητα των δεδομένων. Τα endpoints που τροποποιούν δεδομένα προστατεύονται με ελέγχους εξουσιοδότησης βάσει ρόλων από τα JWTs που εκδίδει το Keycloak.

3.2.2 Cart Service

Η υπηρεσία Cart Service είναι υπεύθυνη για την αποθήκευση και τη διαχείριση του ενεργού καλαθιού αγορών κάθε χρήστη και για την έναρξη της διαδικασίας δημιουργίας παραγγελίας μέσω



Σχήμα 3.7: Το database schema του Product Service.

της ενέργειας Cart Checkout. Η υπηρεσία δεν διαχειρίζεται τις παραγγελίες ούτε τις χρηματικές συναλλαγές των παραγγελιών. Αυτές οι ενέργειες πραγματοποιούνται από τις υπηρεσίες Order και Payment Service, αντίστοιχα, που θα αναφερθούν στις επόμενες υποενότητες. Το Cart Service έχει, με τη σειρά του, το δικό του PostgreSQL database schema και εκθέτει τα δικά του RESTful APIs για την αλληλεπίδραση με τα δεδομένα του. Κάθε αυθεντικοποιημένος χρήστης διαθέτει ένα μοναδικό καλάθι αγορών, το οποίο λειτουργεί ως προσωρινή αποθήκευση των προϊόντων ώστε να συνεχίσει στη συνέχεια στην βασική ροή saga που είναι η δημιουργία παραγγελίας με τα προϊόντα που περιέχει το καλάθι.

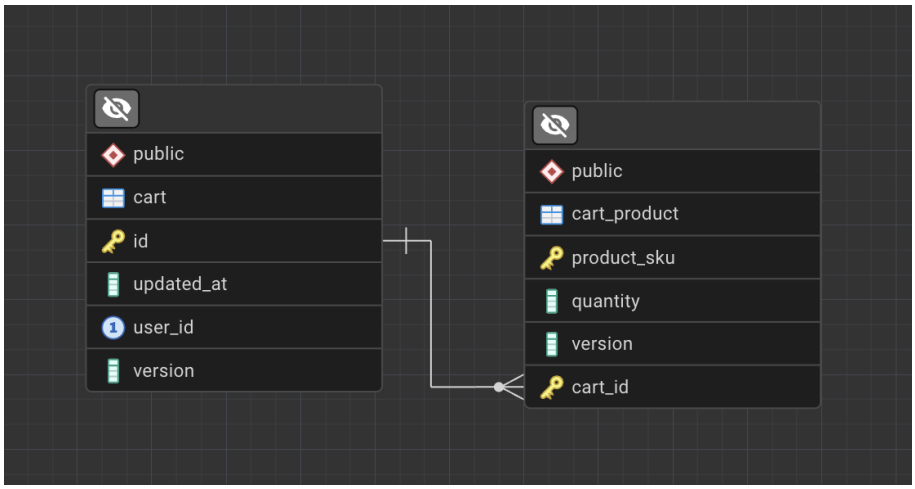
Όλες οι ενέργειες που αφορούν τη διαχείριση του καλαθιού είναι διαθέσιμες μέσω REST endpoints και επιτρέπουν στον αυθεντικοποιημένο χρήστη να αλληλεπιδράσει μαζί τους. Συγκεκριμένα, οι ενέργειες περιλαμβάνουν την προβολή των προϊόντων που βρίσκονται εντός καλαθιού, την προσθήκη προϊόντος στο καλάθι, την ενημέρωση ποσότητας ενός προϊόντος καθώς και την αφαίρεσή του από το καλάθι. Επιπλέον, υπάρχει τρόπος ο χρήστης να δημιουργήσει μια παραγγελία από τα προϊόντα του καλαθιού του (checkout). Υπάρχει έλεγχος πριν από κάθε εκτέλεση κάποιας ενέργειας με επικύρωσης του JWT που έχει εκδόσει το Keycloak. Ο πίνακας 3.2 περιέχει τα διαθέσιμα endpoints για την αλληλεπίδραση των χρηστών με το καλάθι αγορών τους.

Το σχήμα της βάσης δεδομένων αυτής της υπηρεσίας αποτελείται από τους πίνακες cart και cart_product. Ο πίνακας cart περιλαμβάνει τα πεδία id, user_id που αφορά το id του χρήστη σύμφωνα με το Keycloak, την ημερομηνία τελευταίας επεξεργασίας του καλαθιού (updated_at) και το version για τον εντοπισμό ταυτόχρονων τροποποιήσεων. Η μοναδικότητα του user_id εξασφαλίζει

Πίνακας 3.2: Τα HTTP endpoints του Cart Service

Μέθοδος	Endpoint	Σκοπός	Ρόλος
GET	/api/cart	Ανάκτηση προϊόντων καλαθιού	user
POST	/api/cart/products	Εισαγωγή προϊόντος στο καλάθι	user
POST	/api/cart/checkout	Δημιουργία παραγγελίας από τα προϊόντα του καλαθιού	user
PATCH	/api/cart/products/{sku}	Ενημέρωση ενός προϊόντος που βρίσκεται στο καλάθι	user
DELETE	/api/cart/products/sku	Αφαίρεση ενός προϊόντος από το καλάθι	user

πως κάθε χρήστης έχει ένα μόνο ενεργό καλάθι. Ο πίνακας cart_product μοντελοποιεί τα προϊόντα που υπάρχουν μέσα στο καλάθι χρησιμοποιώντας ένα σύνθετο κλειδί από τα πεδία cart_id και product_sku. Επίσης, περιέχει τα πεδία quantity και version. Η σχέση του cart με το cart_product είναι ένα-προς-πολλά (one-to-many). Το πεδίο product_sku που αποθηκεύεται στον πίνακα cart_product είναι το SKU που διαχειρίζεται το Product Service. Το σχήμα της βάσης δεδομένων απεικονίζεται στην εικόνα 3.8.



Σχήμα 3.8: Το database schema του Cart Service.

Η συνέπεια των συναλλαγών εφαρμόζεται σε όλες τις εγγραφές. Η προσθήκη προϊόντος δημιουργεί το σύνθετο κλειδί και αποθηκεύει το καλάθι με το νέο προϊόν εντός μιας συναλλαγής. Η ενημέρωση ποσότητας ελέγχει αν το προϊόν υπάρχει ήδη, ανακτά τα τρέχοντα δεδομένα του, επιβεβαιώνει ότι η ζητούμενη ποσότητα δεν υπερβαίνει το διαθέσιμο απόθεμα και αποθηκεύει την εγγραφή. Στο checkout, το Cart Service στέλνει αίτημα στο Order Service για την δημιουργία της παραγγελίας βάσει των προϊόντων του καλαθιού του πελάτη. Έπειτα, το καλάθι καθαρίζεται ανεξαρτήτως αποτελέσματος της ροής saga δημιουργίας παραγγελίας.

Η επικοινωνία του Cart Service με τις άλλες μικροϋπηρεσίες είναι περιορισμένη. Συγκεκριμένα, η υπηρεσία καλεί το Product Service για την ανάκτηση πληροφορίας σχετικά με τα δεδομένα των προϊόντων όπου χρειάζεται (π.χ. κατά την προσθήκη ενός προϊόντος στο καλάθι για τον έλεγχο αποθέματος). Το Order Service καλείται επίσης κατά το checkout, για την δημιουργία μιας παραγγελίας. Η υπηρεσία δεν συμμετέχει στο saga, απλώς ενεργοποιεί τη δημιουργία παραγγελίας. Μετά τον καθαρισμό του καλαθιού, επιστρέφει τον αριθμό παραγγελίας, το συνολικό ποσό και την αρχική κατάστασή της (PENDING) επισημαίνοντας στον πελάτη πως η διαδικασία έχει ξεκινήσει.

Συνοπτικά, το Cart Service διαχειρίζεται ένα καλάθι ανά χρήστη και παρέχει απλές λειτουργίες για την αλληλεπίδραση με αυτό. Εξαρτάται αρκετά από το Product Service για την ανάκτηση των δεδομένων των προϊόντων και από το Order Service για τη δημιουργία της παραγγελίας χωρίς όμως να έχει άμεση συσχέτιση με το saga.

3.2.3 Order Service

Το Order Service είναι υπεύθυνο για την δημιουργία παραγγελιών και την παρακολούθηση κατάστασής τους μέχρι να ολοκληρωθεί η ροή saga. Η υπηρεσία διαθέτει τη δική της βάση PostgreSQL και παρέχει RESTful APIs ώστε οι χρήστες να μπορούν να βλέπουν τις παραγγελίες τους. Κατά την δημιουργία της παραγγελίας, η υπηρεσία δημοσιεύει ένα event στον message broker, ώστε ο saga orchestrator να ξεκινήσει τη ροή saga. Το Order Service δεν ενημερώνει το απόθεμα ούτε εκτελεί πληρωμές. Το μόνο που κάνει είναι να αποθηκεύει τον χρήστη, τα προϊόντα, το συνολικό ποσό και την κατάσταση της παραγγελίας, περιμένοντας εντολές για επιτυχή ή ανεπιτυχή ολοκλήρωσή της.

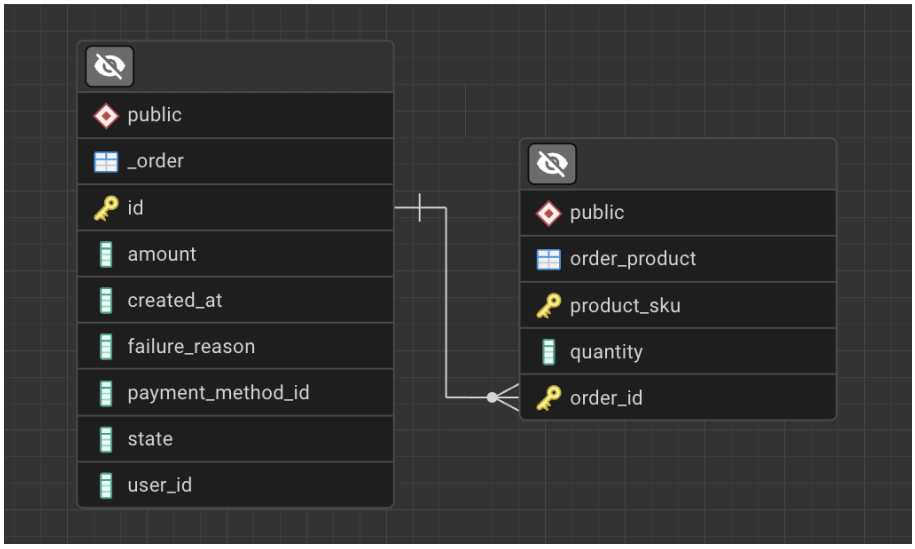
Η υπηρεσία εκθέτει κάποια endpoints που έχουν πρόσβαση οι αυθεντικοποιημένοι χρήστες. Επίσης, υπάρχει ένα POST endpoint που χρησιμοποιεί το Cart Service κατά το checkout για την δημιουργία της παραγγελίας. Επίσης, το Order Service ανακτά μαζικά δεδομένα των προϊόντων από το Product Service, ελέγχει ότι οι ζητούμενες ποσότητες δεν υπερβαίνουν το διαθέσιμο απόθεμα και υπολογίζει το συνολικό ποσό. Αν κάποιο προϊόν λείπει ή υπάρχει υπέρβαση αποθέματος, τότε το αίτημα απορρίπτεται. Διαφορετικά, δημιουργείται μια νέα παραγγελία σε κατάσταση PENDING, αποθηκεύονται τα προϊόντα της και δημοσιεύεται ένα Order Created event με όλα τα σχετικά στοιχεία. Τα endpoints ανάγνωσης επιτρέπουν αναζήτηση και προβολή των παραγγελιών του εκάστοτε συνδεδεμένου χρήστη.Ο πίνακας 3.3 περιέχει τα διαθέσιμα endpoints για την αλληλεπίδραση των χρηστών με τις παραγγελίες τους.

Πίνακας 3.3: Τα HTTP endpoints του Order Service

Μέθοδος	**Endpoint**	**Σκοπός**	**Ρόλος**
GET	/api/orders	Ανάκτηση όλων των παραγγελιών του χρήστη	user
GET	/api/orders/{id}	Ανάκτηση μιας συγκεκριμένης παραγγελίας του χρήστη	user

Το σχήμα της βάσης δεδομένων αποτελείται από τους πίνακες order και order_product, όπως

φαίνεται στην εικόνα 3.9. Ο πίνακας order έχει τα πεδία id, user_id, payment_method_id, amount, state, failure_reason και created_at. Ο πίνακας order_product περιέχει τον συνδυασμό των πεδίων order_id και product_sku ως πρωτεύον κλειδί και το quantity.



Σχήμα 3.9: Το database schema του Order Service.

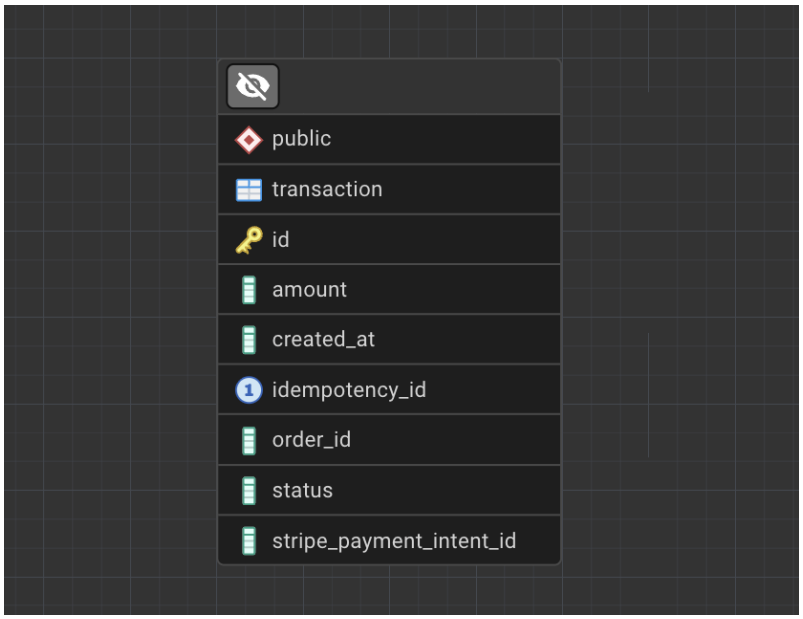
Η υπηρεσία έχει ενεργό ρόλο στο saga. Υπάρχει ένας listener που λαμβάνει τις εντολές του orchestrator και τις δρομολογεί στους αντίστοιχους handlers. Οι αλλαγές κατάστασης βασίζονται αποκλειστικά από τα μηνύματα του saga χωρίς απευθείας επικοινωνία με τις άλλες υπηρεσίες. Περισσότερα σχετικά με την λογική πίσω από τους listeners του Order Service αναφέρονται στην υποενότητα 4.2.1.

3.2.4 Payment Service

Το Payment Service είναι υπεύθυνο για τις ενέργειες εξουσιοδότησης πληρωμής (authorization), οριστικής χρέωσης (capture) και ακύρωσης της δέσμευσης του ποσού (void) σε μια παραγγελία. Η υπηρεσία λειτουργεί αποκλειστικά μέσω του message broker και δεν εκθέτει κάποιο HTTP endpoint για τους σκοπούς της παρούσας εργασίας. Όλες οι ενέργειες πραγματοποιούνται από saga commands που αποστέλλει ο orchestrator και οι απαντήσεις επιστρέφονται ως saga reply events.

Η διασύνδεση με τον πάροχο πληρωμών υλοποιείται με το επίσημο Java Stripe SDK [43] ακολουθώντας μια ροή δύο βημάτων. Αρχικά, δημιουργείται ένα Payment Intent με manual capture ώστε να δεσμεύεται το ποσό χωρίς άμεση χρέωση κι έπειτα πραγματοποιείται η οριστική χρέωση του ποσού εφόσον έχει δεσμευθεί επιτυχώς το απόθεμα. Σε αντίθετη περίπτωση, η εξουσιοδότηση ακυρώνεται (void) χωρίς χρέωση του πελάτη. Το Payment Service υπηρεσία δεν αποθηκεύει στοιχεία κάρτας, διατηρεί μόνο το αναγνωριστικό του Payment Intent και το αποτέλεσμα του.

Το σχήμα της βάσης δεδομένων είναι απλό και βασίζεται σε έναν πίνακα transaction που περιλαμβάνει id, amount, status (αλφαριθμητικό που αναπαριστά διακριτές καταστάσεις), stripe_payment_intent_id (το Stripe Payment Intent id), order_id, idempotency_id και created_at. Τα statuses καταγράφουν τον κύκλο ζωής της πληρωμής και περιλαμβάνουν τα PENDING, AUTHORIZED, PAID, FAILED και VOIDED. Η βάση είναι PostgreSQL σε ξεχωριστό schema, σύμφωνα με την αρχή database-per-service και το σχήμα της απεικονίζεται στην εικόνα 3.10.



Σχήμα 3.10: Το database schema του Payment Service.

3.3 Saga Orchestrator με χρήση Μηχανής Καταστάσεων

3.3.1 Σκοπός και Τρόπος Προσέγγισης

Σε αυτή την ενότητα, θα αναλύσουμε τον σχεδιασμό της διαδικασίας δημιουργίας παραγγελίας σε ένα σύστημα μικροϋπηρεσιών. Δεδομένου του database per service προτύπου, η αυστηρή επίτευξη των ACID ιδιοτήτων δεν είναι εφικτή, σύμφωνα με αυτά που έχουν αναφερθεί στις προηγούμενες ενότητες. Αντ' αυτού, επέλεξα να εφαρμόσω το πρότυπο Saga Orchestration για την διαχείριση αυτής της διαδικασίας. Ο κύριος στόχος είναι η διασφάλιση της τελικής συνέπειας των δεδομένων και η διατήρηση της ανεξαρτησίας των υπηρεσιών. Αυτό εξασφαλίζει την τήρηση κάποιων κανόνων, όπως η εγγύηση ότι καμία παραγγελία δεν ολοκληρώνεται χωρίς ταυτόχρονη επιτυχή δέσμευση αποθέματος και έγκυρη εξουσιοδότηση πληρωμής. Από την πλευρά του χρήστη, η διαδικασία θα πρέπει να εμφανίζεται ως μια ενιαία λειτουργία, παρόλο που αυτή θα εκτείνεται σε πολλαπλές ανεξάρτητες υπηρεσίες.

Η επιλογή της orchestration προσέγγισης αντί της choreography έγινε για να υπάρχει καλύτερος έλεγχος και μια πιο ξεκάθαρη εικόνα ολόκληρης της ροής από ένα κεντρικό σημείο. Με τη χρήση του κεντρικού orchestrator, όλη η ροή είναι πιο κατανοητή και τα σφάλματα μπορούν να εντοπιστούν γρηγορότερα, καθώς ο orchestrator καθορίζει ρητά την ακολουθία όλων των ενεργειών του saga. Παρόλο που η orchestration προσέγγιση αποτελεί ένα βαθμό εξάρτισης μεταξύ των υπηρεσιών το θεώρησα αμελητέο για τους σκοπούς αυτής της εργασίας.

Για να αναπαραστήσω το saga, το μοντελοποίησα ως μια μηχανή καταστάσεων με σαφώς ορισμένες καταστάσεις (states), γεγονότα (events) και μεταβάσεις (transitions). Αυτή η προσέγγιση προσφέρει ντετερμινιστική συμπεριφορά και καθιστά τη διαδικασία πιο εύκολη στην κατανόηση και τον εντοπισμό σφαλμάτων. Επιπλέον, η συνεχής αποθήκευση της τρέχουσας κατάστασης διασφαλίζει ότι το σύστημα μπορεί να ανακάμψει ομαλά σε περίπτωση κάποιου σφάλματος ή κάποιας επανεκκίνησης. Παράλληλα, η υλοποίηση υποστηρίζει ενέργειες που είναι idempotent, δηλαδή μπορούν να εκτελεστούν πάνω από μια φορά χωρίς να αλλάζει το αποτέλεσμα. Η λεπτομερής υλοποίηση αυτού του saga, συμπεριλαμβανομένων των ορισμών καταστάσεων, των κανόνων μετάβασης, του μηχανισμού αποθήκευσης και της ενσωμάτωσης με ένα message broker για την ανταλλαγή γεγονότων ασύγχρονα, θα παρουσιαστεί παρακάτω.

3.3.2 Κύρια Σενάρια του Saga

Το πεδίο εφαρμογής του saga για την ροή «Δημιουργία παραγγελίας» σε αυτή την πτυχιακή εργασία περιορίζεται στην αλληλεπίδραση τριών κύριων υπηρεσιών: Order, Payment και Product Service. Ο orchestrator διαχειρίζεται αυτές τις υπηρεσίες μέσω ασύγχρονης επικοινωνίας ώστε να διασφαλίσει ότι όλα τα βήματα της ροής saga προχωρούν σωστά. Το Order Service λειτουργεί ως μια υπηρεσία που αναλαμβάνει την τελική σύνθεση της παραγγελίας, διατηρώντας την τελική κατάσταση της παραγγελίας κατά την ολοκλήρωση της διαδικασίας. Το Payment Service δίνει εντολές για εξουσιοδότηση και είσπραξη χρημάτων ή τα ακυρώνει σε περίπτωση που κάτι πάει

λάθος. Τέλος, το Product Service δεσμεύει ή απελευθερώνει απόθεμα. Άλλα components του συστήματος e-commerce, όπως η λογική του καλαθιού αγορών, ο κατάλογος προϊόντων και το API Gateway, δεν συμμετέχουν στο saga και επομένως δεν αναφέρονται σε αυτόν τον σχεδιασμό. Ενώ κάποια από αυτά τα components συμμετέχουν έμμεσα στο saga, παρέχοντας πληροφορίες στις υπηρεσίες του saga, δεν έχουν ενεργό ρόλο στην λογική αποφάσεων που παίρνει ο orchestrator.

Η ροή saga ξεκινά όταν μια νέα παραγγελία δημιουργηθεί και αφού ο orchestrator δημιουργήσει ένα καινούργιο state machine instance για αυτή την παραγγελία. Αρχικά, ο orchestrator ζητά για την εξουσιοδότηση της πληρωμής, η οποία δεσμεύει προσωρινά το ποσό χρέωσης του πελάτη. Εάν η εξουσιοδότηση ήταν επιτυχής, προχωρά στην δέσμευση του απαιτούμενου αποθέματος για κάθε προϊόν της παραγγελίας. Μόνο όταν η δέσμευση προϊόντων είναι επιτυχής, ο orchestrator δίνει εντολή στο Payment Service για να ολοκληρώσει τη χρέωση (capture). Τέλος, ο orchestrator ενημερώνει το Order Service για να ορίσει την παραγγελία ως ολοκληρωμένη. Κάθε μετάβαση μεταξύ αυτών των βημάτων καταγράφεται στην μηχανή καταστάσεων, επιτρέποντας στο σύστημα να συνεχίσει τη ροή του μετά από επανεκκινήσεις ή καθυστερήσεις μηνυμάτων. Αυτή η ακολουθία αποτελεί τη βασική ιδανική διαδρομή (happy path), όπου όλες οι συμμετέχουσες υπηρεσίες ολοκληρώνουν τις ενέργειες επιτυχώς και ο χρήστης το αντιλαμβάνεται όλο αυτό ως μια ατομική ενέργεια.

Ωστόσο, υπάρχουν αρκετές διαδρομές αποτυχίας που είναι εξίσου σημαντικές για την διατήρηση της ακεραιότητας του συστήματος. Για παράδειγμα, εάν το Payment Service απορρίψει την αρχική εξουσιοδότηση, το saga τερματίζεται αμέσως και το Order Service σημάνει την παραγγελία ως αποτυχημένη χωρίς να εμπλακεί στην διαδικασία το Product Service. Επιπλέον, όταν η δέσμευση αποθέματος δεν μπορεί να εκπληρωθεί (π.χ. λόγω ανεπαρκούς αποθέματος), ο orchestrator θα ξεκινήσει τις αντισταθμιστικές ενέργειες δίνοντας εντολή στο Payment Service να ακυρώσει (void) την προηγούμενη εξουσιοδότηση για να διασφαλιστεί ότι δεν θα παραμείνουν δεσμευμένα τα χρήματα. Ένα πιο σύνθετο και πιο σπάνιο σενάριο συμβαίνει όταν η ολοκλήρωση της πληρωμής αποτυγχάνει αφού το απόθεμα έχει δεσμευθεί επιτυχώς. Σε αυτή την περίπτωση, ο orchestrator διαδοχικά απελευθερώνει το δεσμευμένο απόθεμα, ακυρώνει την εξουσιοδότηση και μόνο τότε οριστικοποιεί την παραγγελία ως αποτυχημένη. Αυτές οι αντισταθμιστικές ενέργειες διασφαλίζουν ότι κάθε υπηρεσία τελικά έχει μια συνεπή κατάσταση, ακόμη κι όταν συμβαίνουν μερικές αποτυχίες ενδιάμεσα στη διαδικασία. Με αυτό τον σχεδιασμό, το saga επιτυγχάνει την τελική συνέπεια (eventual consistency) μεταξύ των αυτόνομων υπηρεσιών χωρίς κατανεμημένες ACID συναλλαγές. Η επόμενη υποενότητα, θα εξηγήσει περαιτέρω τους λειτουργικούς ρόλους κάθε υπηρεσίας και τις αλληλεπιδράσεις τους με τον orchestrator.

3.3.3 Συμμετέχουσες Υπηρεσίες και Αρμοδιότητες

Το saga για την «Δημιουργία παραγγελίας», όπως προαναφέραμε, περιλαμβάνει τρεις κύριες υπηρεσίες που συνεργάζονται μέσω ασύγχρονης επικοινωνίας υπό την επίβλεψη του κεντρικού orchestrator. Κάθε υπηρεσία παραμένει αυτόνομη και έχει τα δικά της δεδομένα και επιχειρησιακή λογική, ενώ ο orchestrator διατηρεί τον έλεγχο της ροής της διαδικασίας μέσω μιας διατηρούμενης

μηχανής καταστάσεων. Το Order Service κατέχει το σύνολο της παραγγελίας και είναι αποκλειστικά υπεύθυνο για την ενημέρωση της τελικής κατάστασης του κύκλου ζωής της παραγγελίας, είτε ολοκληρωμένης είτε αποτυχημένης, μόλις λάβει εντολή από τον orchestrator. Το Payment Service διαχειρίζεται τις χρηματικές λειτουργίες, όπως εξουσιοδότηση, ολοκλήρωση και ακύρωση πληρωμών. Το Product Service διασφαλίζει ότι το απόθεμα δεσμεύεται ή απελευθερώνεται σωστά. Ο orchestrator δεν τροποποιεί δεδομένα άμεσα. Αντιθέτως, δίνει τις απαραίτητες οδηγίες στο saga στέλνοντας εντολές σε αυτές τις υπηρεσίες και ερμηνεύοντας τις απαντήσεις τους ως γεγονότα που ενεργοποιούν μεταβάσεις κατάστασης.

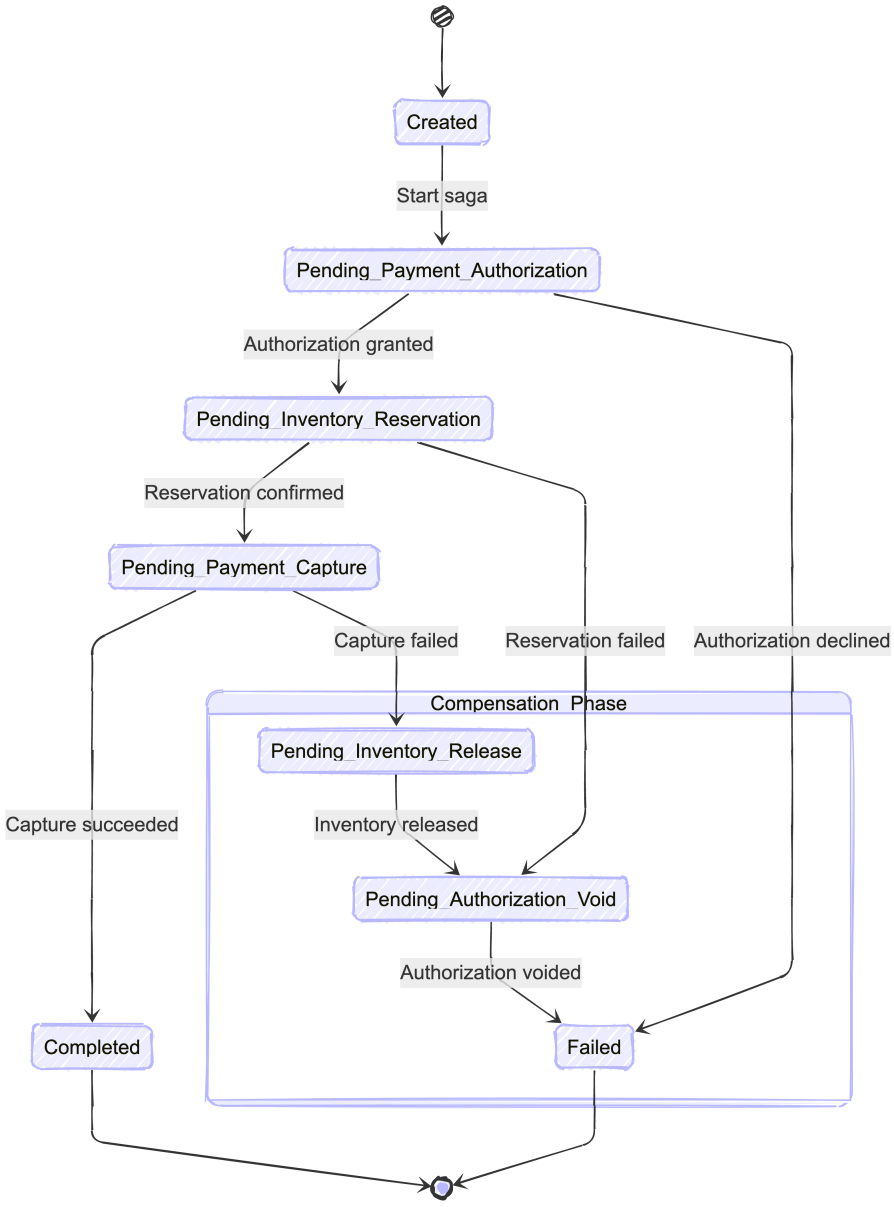
Το saga ξεκινά όταν το Order Service εκπέμψει ένα γεγονός αφού δημιουργήσει και αποθηκεύσει μια νέα παραγγελία. Ο orchestrator ακούει για αυτό το γεγονός, ξεκινά ένα νέο saga instance και αρχίζει να στέλνει εντολές στις συμμετέχουσες υπηρεσίες διαδοχικά. Κάθε εντολή, στοχεύει την αντίστοιχη υπηρεσία και κάθε υπηρεσία με τη σειρά της εκπέμπει ένα γεγονός που υποδεικνύει επιτυχία ή αποτυχία. Ο orchestrator είναι ο μόνος που ερμηνεύει αυτά τα αποτελέσματα και στη συνέχεια ενημερώνει το saga και αποφασίζει για την επόμενη ενέργεια. Εάν συμβεί κάποια αποτυχία, ο orchestrator μεταβαίνει σε κατάσταση αντιστάθμισης και ξεκινά τις διορθωτικές ενέργειες όπως η ακύρωση μιας προηγούμενης εξουσιοδότησης πληρωμής. Αυτές οι αντισταθμιστικές ενέργειες εκτελούνται από τις υπεύθυνες υπηρεσίες οι οποίες λαμβάνουν τις αντίστοιχες εντολές. Επομένως, ο orchestrator καθορίζει το πότε συμβαίνουν οι ενέργειες, ενώ οι υπηρεσίες καθορίζουν το πώς εκτελούνται εντός των δικών τους ορίων.

3.3.4 Εννοιολογικό Μοντέλο Καταστάσεων

Σε αυτή την υποενότητα, μοντελοποιείται ο κύκλος ζωής του saga χρησιμοποιώντας ένα σύνολο από καταστάσεις που περιγράφουν τις κύριες φάσεις της ροής δημιουργίας παραγγελίας από την έναρξη μέχρι κάποιο τελικό αποτέλεσμα, είτε επιτυχές είτε ανεπιτυχές. Αυτές οι καταστάσεις δεν συμπεριλαμβάνουν τις τεχνικές λεπτομέρειες της υλοποίησης αλλά περιγράφουν τη λογική του orchestrator σε ένα πιο αφαιρετικό επίπεδο. Ο κύκλος ζωής ξεκινά σε μια κατάσταση Created όταν ο orchestrator λάβει ένα event νέας παραγγελίας. Από εκεί, το saga προχωρά μέσω διαδοχικών καταστάσεων, πριν φτάσει σε ένα τελικό αποτέλεσμα. Οι δύο πιθανές τελικές καταστάσεις είναι Completed και Failed, για την ολοκληρωμένη ή αποτυχημένη παραγγελία αντίστοιχα.

Οι μεταβάσεις μεταξύ των καταστάσεων ακολουθούν την ροή που απεικονίζεται στο διάγραμμα καταστάσεων στην Εικόνα 3.11. Αφού το saga ξεκινήσει σε κατάσταση Created, μετακινείται στην κατάσταση Pending_Payment_Authorization, όπου με μια επιτυχή εξουσιοδότηση θα μεταβεί στην κατάσταση Pending_Inventory_Reservation ενώ με μια απόρριψη θα μεταβεί στην τερματική κατάσταση Failed. Μόλις το απόθεμα δεσμευθεί, το saga προχωρά στην κατάσταση Pending_Payment_Capture. Μια επιτυχής ολοκλήρωση πληρωμής, οδηγεί στην τερματική κατάσταση Completed, ενώ μια αποτυχία ενεργοποιεί την διαδικασία αντιστάθμισης. Εντός αυτής της διαδικασίας, ο orchestrator δίνει εντολή πρώτα για την απελευθέρωση του δεσμευμένου αποθέματος και στη συνέχεια ακυρώνει την προηγούμενη εξουσιοδοτημένη πληρωμή με αποτέλεσμα το saga να φτάσει στην τελική κατάσταση Failed. Κάθε μετάβαση ενεργοποιείται από το αντίστοιχο

αποτέλεσμα που επιστρέφει η υπεύθυνη υπηρεσία που έχει αναλάβει να εκτελέσει την αντίστοιχη ενέργεια. Η λεπτομερής ανάλυση των καταστάσεων, των γεγονότων και όλων των μηχανισμών υλοποίησης θα παρουσιαστεί στο κεφάλαιο 4.1.

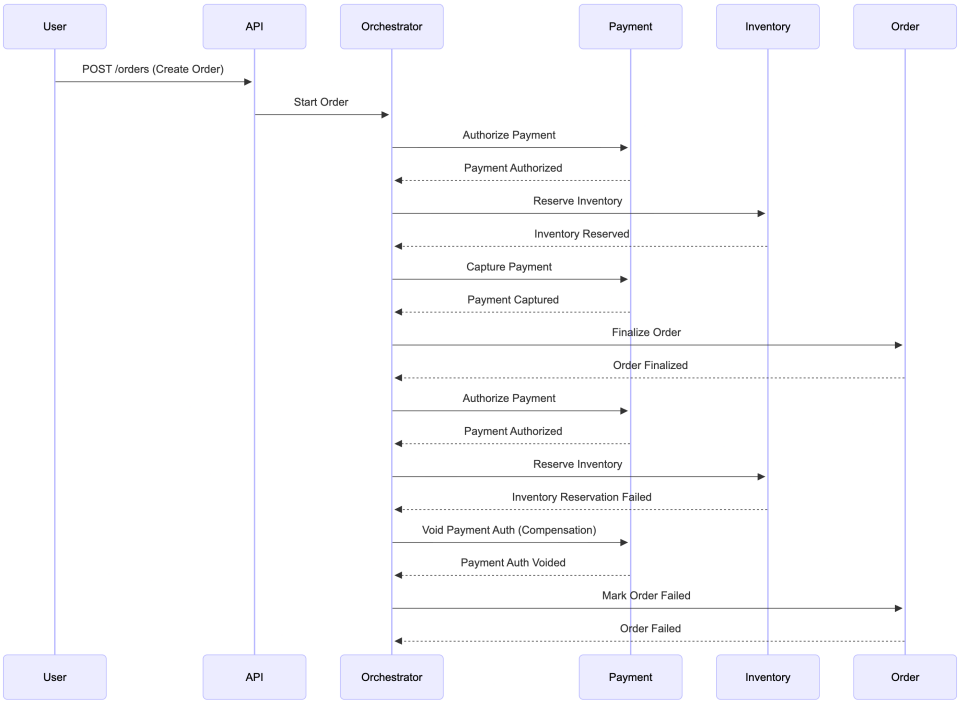


Σχήμα 3.11: Διάγραμμα καταστάσεων του Saga.

3.3.5 Τρόπος Αλληλεπίδρασης

Στην παρούσα πτυχιακή εργασία, όλες οι αλληλεπιδράσεις του saga πραγματοποιούνται μέσω ασύγχρονης ανταλλαγής μηνυμάτων κι όχι μέσω σύγχρονων κλήσεων HTTP. Κάθε βήμα του saga, επικοινωνεί με χρήση μηνυμάτων μέσω ενός message broker. Ο orchestrator στέλνει εντολές (commands) στις υπηρεσίες και ακούει για τα αντίστοιχα γεγονότα αποτελέσματος (outcome events). Έτσι, επιτρέπει σε κάθε υπηρεσία να επεξεργάζεται τα αιτήματα σε δικό της χρόνο. Αποφεύγω συνειδητά, όσο είναι εφικτό, τη χρήση σύγχρονων κλήσεων HTTP στη ροή του saga, αν και κάποιες επιμέρους υπηρεσίες καλούν κάποιες άλλες μέσω των REST endpoints τους.

Όπως φαίνεται στην εικόνα 3.12, ο orchestrator ακολουθεί ένα μοτίβο command-event για κάθε βήμα στη διαδικασία δημιουργίας παραγγελίας. Στέλνει ένα command, όπως το Authorize Payment και περιμένει μέχρι ένα σχετικό event να επιστραφεί σηματοδοτώντας είτε την επιτυχία είτε την αποτυχία. Κάθε event θα πυροδοτήσει μια ντετερμινιστική μετάβαση στη μηχανή καταστάσεων του orchestrator.



Σχήμα 3.12: Διάγραμμα ροής Saga.

4. Υλοποίηση

4.1 Saga Orchestrator με χρήση Spring Statemachine

Σύμφωνα με τον σχεδιασμό που αναφέρθηκε στην ενότητα 3.3, υλοποίησα τον Saga Orchestrator χρησιμοποιώντας τη βιβλιοθήκη Spring Statemachine [41] που παρέχει το Spring framework. Οι επόμενες υποενότητες εξηγούν με λεπτομέρεια την υλοποίηση, χρησιμοποιώντας αποσπάσματα κώδικα όπου χρειάζεται.

4.1.1 Παραμετροποίηση Μηχανής Καταστάσεων

Η μηχανή καταστάσεων του saga ορίζεται από ένα σύνολο διακριτών καταστάσεων (states) και γεγονότων (events) που προκαλούν τις μεταβάσεις (transitions) μεταξύ αυτών. Τα states αντιπροσωπεύουν τα βήματα του saga (π.χ. αναμονή έγκρισης πληρωμής), ενώ τα events αντιπροσωπεύουν τις απαντήσεις από τις άλλες υπηρεσίες (π.χ. η πληρωμή εγκρίθηκε). Τα states και τα events μοντελοποιούνται ως Java enums όπως φαίνεται στα αποσπάσματα κώδικα 4.1 και 4.2 αντίστοιχα.

```
1  public enum SagaState {
2      CREATED,
3      PENDING_PAYMENT_AUTHORIZATION,
4      PENDING_INVENTORY_RESERVATION,
5      PENDING_PAYMENT_CAPTURE,
6      PENDING_INVENTORY_RELEASE,
7      PENDING_AUTHORIZATION_VOID,
8      COMPLETED,
9      FAILED,
10 }
```

Απόσπασμα 4.1: OrderState enum

```
1  public enum SagaEvent {
2      START_SAGA,
3      PAYMENT_AUTHORIZED,
4      INVENTORY_RESERVED,
5      PAYMENT_CAPTURED,
6
7      PAYMENT_AUTHORIZATION_FAILED,
8      INVENTORY_RESERVATION_FAILED,
9      PAYMENT_CAPTURE_FAILED,
10
11     AUTHORIZATION_VOIDED,
12     INVENTORY_RELEASED,
13 }
```

Απόσπασμα 4.2: OrderEvent enum

Το configuration του state machine γίνεται στο αρχείο StateMachineConfig.java. Η κλάση StateMachineConfig κληρονομεί από την EnumStateMachineConfigurerAdapter που προέρχεται από τη Spring Statemachine βιβλιοθήκη. Σε αυτή τη κλάση υπάρχει το μεγαλύτερο κομμάτι του configuration του state machine. Αρχικά, χρειάστηκε να ορίσω τα states του state machine, όπως φαίνεται στο απόσπασμα 4.3. Η αρχική κατάσταση (initial state) ορίστηκε ως η CREATED και οι τερματικές καταστάσεις (terminal states) ως οι COMPLETED και FAILED. Στο παράρτημα, φαίνονται και κάποια actions που δίνονται σαν δεύτερα ορίσματα, όπως το authorizePaymentAction. Αυτά θα αναλυθούν στην επόμενη υποενότητα.

```
@Override
public void configure(StateMachineStateConfigurer<SagaState, SagaEvent> states) throws Exception {
    states.withStates()
            .initial(CREATED)
            .states(EnumSet.allOf(SagaState.class))
            .stateEntry(PENDING_PAYMENT_AUTHORIZATION, authorizePaymentAction)
            .stateEntry(PENDING_INVENTORY_RESERVATION, reserveInventoryAction)
            .stateEntry(PENDING_PAYMENT_CAPTURE, capturePaymentAction)
            .stateEntry(PENDING_AUTHORIZATION_VOID, voidAuthorizationAction)
            .stateEntry(PENDING_INVENTORY_RELEASE, releaseInventoryAction)
            .stateEntry(COMPLETED, completeOrderAction)
            .stateEntry(FAILED, failOrderAction)
            .end(COMPLETED)
            .end(FAILED);
}
```

Απόσπασμα 4.3: States configuration

Το αμέσως επόμενο πιο σημαντικό κομμάτι του configuration είναι ο ορισμός των transitions του state machine. Ακολουθόντας παρόμοιο τρόπο με αυτόν του ορισμού των states, όρισα και τα transitions όπως φαίνεται στο απόσπασμα 4.4. Ουσιαστικά, η μέθοδος configure λαμβάνει ως όρισμα ένα αντικείμενο τύπου StateMachineTransitionConfigurer το οποίο χρησιμοποιείται μετά για τον ορισμό των μεταβάσεων. Κάθε ορισμός ενός transition ξεκινά με τη μέθοδο withExternal για να δηλώσει μια τυπική αλλαγή κατάστασης. Μετά ορίζεται η κατάσταση προέλευσης με τη μέθοδο source και η κατάσταση προορισμού με τη μέθοδο target. Η αλλαγή αυτή πυροδοτείται από ένα συγκεκριμένο event που ορίζεται στη μέθοδο event. Τέλος, η μέθοδος and κλείνει τον ορισμό του τρέχοντος transaction και μας επιτρέπει να ξεκινήσουμε τον ορισμό του επόμενου κάνοντας όλο το configuration μια ευανάγνωστη αλυσίδα.

```
@Override
public void configure(StateMachineTransitionConfigurer<SagaState, SagaEvent> transitions) throws
     Exception {
    transitions
            // CREATED
            .withExternal()
            .source(CREATED)
            .target(PENDING_PAYMENT_AUTHORIZATION)
            .event(START_SAGA)
            .and()
```

```
10
11              // PENDING_PAYMENT_AUTHORIZATION
12              .withExternal()
13              .source(PENDING_PAYMENT_AUTHORIZATION)
14              .target(PENDING_INVENTORY_RESERVATION)
15              .event(PAYMENT_AUTHORIZED)
16              .and()
17
18              .withExternal()
19              .source(PENDING_PAYMENT_AUTHORIZATION)
20              .target(FAILED)
21              .event(PAYMENT_AUTHORIZATION_FAILED)
22              .and()
23
24              // [...]
25
26              // PENDING_INVENTORY_RELEASE
27              .withExternal()
28              .source(PENDING_INVENTORY_RELEASE)
29              .target(PENDING_AUTHORIZATION_VOID)
30              .event(INVENTORY_RELEASED)
31              .and()
32
33              // PENDING_AUTHORIZATION_VOID
34              .withExternal()
35              .source(PENDING_AUTHORIZATION_VOID)
36              .target(FAILED)
37              .event(AUTHORIZATION_VOIDED);
38  }
```

Απόσπασμα 4.4: Transitions configuration

4.1.2 Υλοποίηση Λογικής

Τα actions είναι κάποιες μέθοδοι που κάνουν implement το action functional interface που παρέχει η βιβλιοθήκη Spring Statemachine. Αυτές οι μέθοδοι μπορούν να εκτελούνται από το state machine κατά την είσοδο ή την έξοδό του από συγκεκριμένες καταστάσεις. Στην συγκεκριμένη υλοποίηση, τα actions χρησιμοποιούνται κατά την είσοδο του state machine στις καταστάσεις για την αποστολή των commands προς τις άλλες υπηρεσίες μέσω RabbitMQ. Τα actions στη συγκεκριμένη υλοποίηση κάνουν implement το SagaAction interface το οποίο έχει μια προκαθορισμένη βοηθητική μέθοδο, που όλα τα actions χρειάζονται, για την ανάκτηση των πληροφοριών της παραγγελίας που αφορούν το state machine. Το SagaAction κάνει extend το Action interface που παρέχει η βιβλιοθήκη όπως φαίνεται το απόσπασμα 4.5.

```
1  public interface SagaAction extends Action<SagaState, SagaEvent> {
2      default OrderCreatedEvent getSagaContext(StateContext<SagaState, SagaEvent> context) {
3          return context.getExtendedState().get(SAGA_CONTEXT_HEADER, OrderCreatedEvent.class);
4      }
```

```
}
```

Απόσπασμα 4.5: SagaAction interface

Στο απόσπασμα 4.6 φαίνεται ένα παράδειγμα υλοποίησης του SagaAction για την ενέργεια Authorize Payment. Κατά την κλήση της μεθόδου execute, το action ανακτά τις πληροφορίες τις παραγγελίας και στη συνέχεια φτιάχνει το AuthorizePaymentCommand DTO και το στέλνει στο RabbitMQ queue με τη χρήση του RabbitTemplate και της μεθόδου convertAndSend, τα οποία θα αναλυθούν στην υποενότητα 4.1.4. Επίσης, το action καταγράφει και logs που βοηθούν στην καλύτερη κατανόηση της ροής και στην διευκόλυνση του εντοπισμού σφαλμάτων. Επιπλέον, η κλάση έχει 3 annotations, όπου τα 2 από αυτά (Log4j2 και RequiredArgsConstructor) τα παρέχει η βιβλιοθήκη Lombok. Το Component annotation το παρέχει το Spring.

```
@Log4j2
@RequiredArgsConstructor
@Component
public class AuthorizePaymentAction implements SagaAction {

    private final RabbitTemplate rabbitTemplate;

    @Override
    public void execute(StateContext<SagaState, SagaEvent> context) {
        OrderCreatedEvent sagaContext = getSagaContext(context);
        log.info("Dispatching AuthorizePaymentCommand for orderId: {}", sagaContext.getOrderId());
        var command = new AuthorizePaymentCommand(sagaContext.getOrderId(), sagaContext.
            getTotalAmount(), sagaContext.getPaymentToken(), UUID.randomUUID().toString());
        rabbitTemplate.convertAndSend(SAGA_EXCHANGE, PAYMENT_AUTHORIZE_KEY, command);
    }
}
```

Απόσπασμα 4.6: Authorize Payment action class

Όλα τα actions ακολουθούν παρόμοια λογική, δηλαδή, υλοποιούν το interface SagaAction, ανακτούν τα δεδομένα της παραγγελίας, κάνουν τα απαραίτητα logs, φτιάχνουν το command DTO και έπειτα στέλνουν το μήνυμα μέσω RabbitMQ. Επιστρέφοντας στο απόσπασμα 4.3, παρατηρούμε πως έχω ορίσει ένα action για κάθε state όπου χρειάζεται να σταλεί κάποιο command. Αυτό το action θα κληθεί αυτόματα κατά την εισαγωγή του state machine στο εκάστοτε state.

Σχετικά με τα guards και τους interceptors, δεν χρειάστηκε να υλοποιηθεί κάτι από αυτά για τους σκοπούς αυτής της πτυχιακής εργασίας.

4.1.3 Αποθήκευση Κατάστασης

Η αποθήκευση του context των state machine είναι σημαντική ώστε το σύστημα να ανακτά την πρόοδό του σε περίπτωση κάποιας επανεκκίνησης. Παράλληλα, η βάση δεδομένων λειτουργεί και ως κεντρικό σημείο αλήθειας (source of truth) σε περίπτωση που υπάρχουν πάνω από ένα instances του orchestrator.

Σε αυτό το σύστημα, έχει γίνει η επιλογή της MongoDB ως βάση δεδομένων. Η αποθήκευση υλοποιείται με το StateMachineRuntimePersister της βιβλιοθήκης Spring Statemachine το οποίο συνδέεται με τη MongoDB, όπως φαίνεται στο απόσπασμα 4.7.

```
@Bean
public StateMachineRuntimePersister<SagaState, SagaEvent, String> stateMachinePersister(
        MongoDbStateMachineRepository mongoRepository
) {
    return new MongoDbPersistingStateMachineInterceptor<>(mongoRepository);
}
```

Απόσπασμα 4.7: StateMachine persister configuration

Για την ολοκλήρωση του configuration του persister, χρειάστηκε να το δώσω σαν όρισμα στον state machine configuration configurer όπως φαίνεται στο απόσπασμα 4.8.

```
@Override
public void configure(StateMachineConfigurationConfigurer<SagaState, SagaEvent> config) throws
    Exception {
    config
            .withPersistence()
            .runtimePersister(persister);
}
```

Απόσπασμα 4.8: Persistence configuration

4.1.4 Ασύγχρονη Επικοινωνία

Σε αυτή την υποενότητα θα αναλυθεί η λογική της επικοινωνίας μεταξύ του Saga Orchestrator και των υπηρεσιών μέσω του RabbitMQ. Η λογική είναι πως υπάρχει ένα queue για κάθε υπηρεσία που υπάρχει στο saga κι ακόμη ένα για τον orchestrator. Συγκεκριμένα, τα queues είναι τα saga_orchestrator_events_queue, payment_commands_queue, inventory_commands_queue και order_events_queue για τον orchestrator, το Payment Service, το Product Service και το Order Service αντίστοιχα.

Το saga_orchestrator_events_queue χρησιμοποείται από τις υπηρεσίες για να ανακοινώνουν μέσω αυτού events που αφορούν τον saga orchestrator. Ο orchestrator με τη σειρά του ακούει γι' αυτά τα events και πράττει ανάλογα. Τα άλλα queues είναι command queues και τα χρησιμοποιεί ο orchestrator για να στείλει τα command μηνύματα στις υπηρεσίες.

Στο RabbitMQ έχει οριστεί κι ένα topic exchange, με όνομα saga_exchange, για να γίνεται πιο απλή η μετάδοση των μηνυμάτων από τους συμμετέχοντες. Το exchange λειτουργεί ως ένα κεντρικό hub επικοινωνίας και όλα τα commands από τον orchestrator και τα events από τις υπηρεσίες δημοσιεύονται σε αυτό το exchange αντί να δημοσιεύονται απευθείας στο queue. Η δημοσίευση σε ένα exchange συνοδεύεται από ένα κλειδί δρομολόγησης (routing key). Τα routing keys είναι πολύ σημαντικά σε αυτή την υλοποίηση καθώς βοηθούν το exchange στο να γνωρίζει πού πρέπει να δρομολογήσει το κάθε μήνυμα.

Αναλυτικότερα, τα routing keys ακολουθούν το μοτίβο `{service}.{type}.{action}`. Το type ορίζει τι είδους είναι το μήνυμα. Σε αυτή την υλοποίηση, το type θα είναι είτε command είτε event. Στην περίπτωση που είναι command, το service placeholder υποδεικνύει τον αποδέκτη του command. Για παράδειγμα, ένα routing key της μορφής `payment.command.*` θα απευθυνόταν στο Payment Service. Τα command type μηνύματα τα στέλνει μόνο ο orchestrator στο saga. Στην αντίθετη περίπτωση, δηλαδή όταν το type έχει την τιμή event, το service placeholder υποδεικνύει τον αποστολέα του event. Δηλαδή, το κλειδί `payment.event.*` σημαίνει πως το μήνυμα πρόκειται για ένα event που έχει εκπέμψει το Payment Service. Τα event type μηνύματα τα στέλνουν οι υπηρεσίες και απευθύνονται στον orchestrator. Τέλος, το action πρόκειται για μια συγκεκριμένη ενέργεια. Δηλαδή, αν πρόκειται για ένα command action, η τιμή θα μπορούσε να ήταν authorize και το τελικό routing key να ήταν `payment.command.authorize`. Αντίστοιχα, το event που θα χρησιμοποιούσε το Payment Service μετά από επιτυχή δέσμευση του ποσού θα ήταν το `payment.event.authorized`.

Για την σωστή δρομολόγηση των μηνυμάτων μέσω του exchange χρειάστηκε να οριστούν τα λεγόμενα bindings στο configuration. Η ενσωμάτωση της Spring Boot Starter AMQP βιβλιοθήκης [40] βοήθησε πολύ στην όλη διαδικασία. Στο απόσπασμα 4.9, φαίνεται ένα μεγάλο μέρος του configuration για τον τρόπο δήλωσης των exchange, queues και bindings. Όπως φαίνεται, το orchestratorBinding κάνει bind όλα τα κλειδιά που έχουν σαν type το event (`*.event.*`) να δρομολογούνται στο orchestrator queue, ενώ οι άλλες τιμές είναι wildcard (`*`). Αντίστοιχα, τα payment commands να δρομολογούνται στο payment queue. Οι ονομασίες είναι σταθερές που προέρχονται από την κλάση SagaConstants.

```
@Bean
TopicExchange exchange() {
    return new TopicExchange(SAGA_EXCHANGE);
}

@Bean
Queue orchestratorQueue() {
    return new Queue(ORCHESTRATOR_QUEUE, true);
}

@Bean
Queue paymentQueue() {
    return new Queue(PAYMENT_QUEUE, true);
}

@Bean
Binding orchestratorBinding(Queue orchestratorQueue, TopicExchange exchange) {
    return BindingBuilder.bind(orchestratorQueue)
            .to(exchange)
            .with("*.event.*");
}

@Bean
Binding paymentBinding(Queue paymentQueue, TopicExchange exchange) {
    return BindingBuilder.bind(paymentQueue)
```

```
26              .to(exchange)
27              .with("payment.command.*");
28  }
```

Απόσπασμα 4.9: RabbitMQ configuration

Για κάθε μήνυμα έχω δημιουργήσει ένα DTO που περιέχει την πληροφορία που χρειάζεται να σταλεί. Όλα τα μηνύματα του saga μεταφέρουν το id της παραγγελίας, ώστε οι αποδέκτες να γνωρίζουν σε ποια παραγγελία αναφέρεται το μήνυμα. Γι' αυτόν τον λόγο όλα τα DTO κάνουν implement το SagaMessage, το οποίο εκθέτει τη μέθοδο orderId (βλ. 4.10).

```
1  public interface SagaMessage {
2      Integer orderId();
3  }
```

Απόσπασμα 4.10: SagaMessage interface

Για τα μηνύματα που αντιστοιχούν σε events, χρησιμοποιώ το SagaEventMessage το οποίο επεκτείνει το SagaMessage και προσθέτει τη μέθοδο sagaEvent, όπως φαίνεται στο απόσπασμα 4.11.

```
1  public interface SagaEventMessage extends SagaMessage {
2      SagaEvent sagaEvent();
3  }
```

Απόσπασμα 4.11: SagaEventMessage interface

Ένα παράδειγμα ενός command είναι το AuthorizePaymentCommand, το οποίο κάνει implement το SagaMessage με τα απαιτούμενα δεδομένα της εξουσιοδότησης πληρωμής (βλ. 4.12). Αντίστοιχα, το InventoryReservedEvent είναι ένα event DTO και υλοποιεί το SagaEventMessage, δηλώνοντας το event που αντιπροσωπεύει (βλ. 4.13).

```
1  public record AuthorizePaymentCommand(Integer orderId, BigDecimal amount, String paymentMethod,
2                                        String idempotencyId) implements SagaMessage {
3  }
```

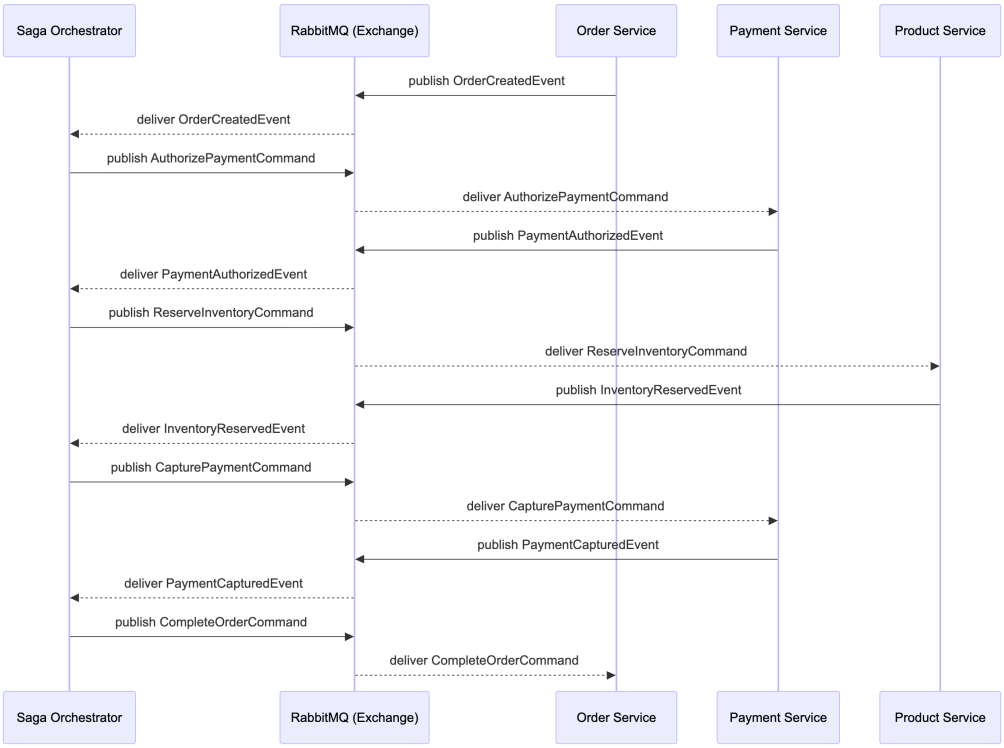
Απόσπασμα 4.12: Authorize Payment Command record

```
1  public record InventoryReservedEvent(Integer orderId) implements SagaEventMessage {
2      @Override
3      public SagaEvent sagaEvent() {
4          return SagaEvent.INVENTORY_RESERVED;
5      }
6  }
```

Απόσπασμα 4.13: Inventory Reserved Event record

Για να γίνει πιο κατανοητή η ασύγχρονη αλληλεπίδραση των υπηρεσιών μέσω των queues και του exchange, παρακάτω απεικονίζεται ένα διάγραμμα ακολουθίας (sequence diagram) που

δείχνει τη ροή των μηνυμάτων σε ένα επιτυχημένο σενάριο δημιουργίας παραγγελίας (βλ. 4.1). Ο Saga Orchestrator διαχειρίζεται τη συνολική ροή και επικοινωνεί με τις υπηρεσίες Order, Payment και Product μέσω του RabbitMQ.



Σχήμα 4.1: Διάγραμμα ακολουθίας της ασύγχρονης επικοινωνίας μεταξύ του Saga Orchestrator και των υπηρεσιών κατά τη διαδικασία δημιουργίας παραγγελίας

Για την επεξεργασία των εισερχόμενων μηνυμάτων, κάθε υπηρεσία του συστήματος διαθέτει έναν listener ο οποίος είναι υπεύθυνος στο να ακούει τα μηνύματα του queue που της αντιστοιχεί. Οι listeners υλοποιούνται με τη χρήση του RabbitListener annotation που παρέχει το Spring AMQP. Με αυτόν τον τρόπο, κάθε φορά που δημοσιεύεται ένα μήνυμα στο exchange και αυτό δρομολογείται στο queue της υπηρεσίας, ο listener λαμβάνει το μήνυμα και εκτελεί το κατάλληλο business logic.

Ο Saga Orchestrator διαθέτει έναν listener για το saga_orchestrator_events_queue μέσω του οποίου λαμβάνει όλα τα events που αποστέλλουν οι επιμέρους υπηρεσίες. Κάθε εισερχόμενο μήνυμα προωθείται στη μέθοδο handleSagaEvent η οποία με τη σειρά της το περνάει στον SagaEventDispatcher μέσω της μεθόδου dispatch. Η κλήση της σωστής μεθόδου του listener βάσει του τύπου του μηνύματος επιτυγχάνεται αυτόματα με τη χρήση του annotation RabbitHandler, που παρέχει η βιβλιοθήκη Spring AMQP. Σε περίπτωση που ληφθεί ένα μήνυμα που δεν είναι τύπου SagaEventMessage, τότε θα κληθεί η μέθοδος handleUnknown η οποία θα κάνει τα κατάλληλα logs. Στο απόσπασμα 4.14, παρουσιάζεται η κλάση SagaEventListener με τις δύο αυτές μεθόδους και το dependency SagaEventDispatcher.

```java
@RabbitListener(queues = ORCHESTRATOR_QUEUE)
public class SagaEventListener {

    private final SagaEventDispatcher dispatcher;

    @RabbitHandler
    public void handleSagaEvent(SagaEventMessage event) {
        log.info("Received saga event in listener: {}", event.getClass().getSimpleName());
        dispatcher.dispatch(event);
    }

    @RabbitHandler(isDefault = true)
    public void handleUnknown(Object payload) {
        if (payload == null) {
            log.warn("Received null payload on orchestrator queue. Ignoring.");
        } else {
            log.warn("Ignoring non-saga payload of type {} on orchestrator queue", payload.
                getClass().getName());
        }
    }
}
```

Απόσπασμα 4.14: Saga Orchestrator listener

Ο SagaEventDispatcher είναι μια κλάση που αναλαμβάνει να δρομολογεί κάθε event στον κατάλληλο handler. Όπως φαίνεται στο απόσπασμα 4.15, ο handler ορίζει μια λίστα από διαθέσιμους handlers και ένα map που χρησιμοποιείται για την αντιστοίχιση κάθε τύπου event με τον αντίστοιχο handler.

Κατά την εκκίνηση της εφαρμογής, εκτελείται η μέθοδος init, η οποία αποθηκέυει κάθε διαθέσιμο handler στο map. Οι handlers, όπως θα δούμε στη συνέχεια, έχουν δηλωθεί ως Spring components επομένως το Spring framework αναλαμβάνει αυτόματα το injection τους στη λίστα.

Όταν ο listener καλέσει τη μέθοδο dispatch, πραγματοποιείται μια αναζήτηση στο handlerMap για να βρεθεί ο handler που αντιστοιχεί στον τύπο του event. Αν δεν βρεθεί κατάλληλος handler, ο dispatcher πετάει ένα exception. Διαφορετικά, καλείται η μέθοδος handle του αντίστοιχου handler, η οποία αναλαμβάνει να επεξεργαστεί το event.

```java
@Log4j2
@RequiredArgsConstructor
@Component
public class SagaEventDispatcher {

    private final List<SagaEventHandler<SagaEventMessage>> handlers;
    private final Map<Class<SagaEventMessage>, SagaEventHandler<SagaEventMessage>> handlerMap =
        new HashMap<>();

    @PostConstruct
    public void init() {
        handlers.forEach(handler -> handlerMap.put(handler.getEventType(), handler));
```

```
12          log.info("Registered {} saga event handlers", handlerMap.size());
13      }
14
15      public void dispatch(SagaEventMessage event) {
16          if (event == null) {
17              log.warn("Received null event. Ignoring.");
18              return;
19          }
20          log.info("Received saga event of type: {}", event.getClass().getSimpleName());
21          var handler = handlerMap.get(event.getClass());
22          if (handler == null) {
23              throw new RuntimeException("No handler found for event type: " + event.getClass());
24          }
25
26          try {
27              handler.handle(event);
28          } catch (Exception e) {
29              log.error("Error handling event of type: {}", event.getClass(), e);
30          }
31      }
32  }
```

Απόσπασμα 4.15: Saga Orchestrator Event Dispatcher

Η διαχείριση των events στον Saga Orchestrator ακολουθεί το Strategy Pattern [42]. Σύμφωνα με αυτό το πρότυπο, κάθε διαφορετικός τύπος event handler υλοποιείται από μια ξεχωριστή κλάση και κάνει implement ένα κοινό interface. Με αυτόν τον τρόπο, ο κώδικας παραμένει πιο καθαρός, κατανοητός και επεκτάσιμος.

Στο απόσπασμα 4.16, φαίνεται το interface, το οποίο ορίζει τη λογική επεξεργασίας ενός event μέσω της μεθόδου handle. Επιπλέον, η μέθοδος getEventType, επιστρέφει τον τύπο του event το οποίο διαχειρίζεται ο εκάστοτε handler. Το interface χρησιμοποιεί generics, ώστε να ενισχύεται η επεναχρησιμοποίηση και η επεκτασιμότητα του κώδικα.

```
1  public interface SagaEventHandler<T extends SagaEventMessage> {
2      void handle(T event);
3      Class<T> getEventType();
4  }
```

Απόσπασμα 4.16: Saga Orchestrator Abstract Event Handler

Η κλάση AbstractSagaEventHandler, που φαίνεται στο απόσπασμα 4.17, χρησιμοποιεί επίσης generics, αποτελεί την βασική υλοποίηση του SagaEventHandler και υλοποιεί κοινή συμπεριφορά για όλους τους handlers. Μέσω της μεθόδου handle κάθε event προωθείται στη μέθοδο processSagaEvent του OrchestratorService.

```
1  @RequiredArgsConstructor
2  public abstract class AbstractSagaEventHandler<T extends SagaEventMessage> implements
       SagaEventHandler<T> {
3
```

```
    private final OrchestratorService orchestratorService;

    @Override
    public void handle(SagaEventMessage event) {
        try {
            orchestratorService.processSagaEvent(event);
        } catch (Exception e) {
            log.error("Error processing {}: {}", event.getClass().getSimpleName(), e.getMessage(),
                e);
        }
    }
}
```

Απόσπασμα 4.17: Saga Orchestrator Abstract Event Handler

Τέλος, στο απόσπασμα 4.18, φαίνεται ένα συγκεκριμένο παράδειγμα υλοποίησης ενός handler για το PaymentAuthorizedEvent, ο οποίος κάνει extend την κλάση AbstractSagaEventHandler. Αυτός ο τρόπος υλοποίησης των handlers, ο οποίος βασίζεται στο interface SagaEventHandler και την abstract κλάση AbstractSagaEventHandler, επιτρέπει την εύκολη προσθήκη νέων event handlers χωρίς να απαιτείται τροποποίηση του SagaEventDispatcher ή του SagaEventListener. Αρκεί να δημιουργηθεί ένας νέος handler για το νέο event, ο οποίος υλοποιεί το interface ή επεκτείνει την abstract κλάση. Με αυτόν τον τρόπο, το σύστημα παραμένει εύκολα επεκτάσιμο.

```
@Component
public class PaymentAuthorizedEventHandler extends AbstractSagaEventHandler<PaymentAuthorizedEvent
    > {
    public PaymentAuthorizedEventHandler(OrchestratorService orchestratorService) {
        super(orchestratorService);
    }

    @Override
    public Class<PaymentAuthorizedEvent> getEventType() {
        return PaymentAuthorizedEvent.class;
    }
}
```

Απόσπασμα 4.18: Saga Orchestrator Payment Event Handler

4.1.5 Λογική Orchestrator Service

Ο OrchestratorService αποτελεί τον βασικό μηχανισμό του Saga Orchestrator και είναι υπεύθυνος για την επεξεργασία των εισερχόμενων events που λαμβάνει ο SagaEventDispatcher. Όπως φαίνεται στο απόσπασμα 4.19, η μέθοδος processSagaEvent είναι το σημείο επεξεργασίας κάθε event. Αρχικά, η μέθοδος εξάγει το orderId και χρησιμοποιεί το StateMachineFactory για να δημιουργήσει ή να ανακτήσει το state machine που αντιστοιχεί στην παραγγελία του event. Αν δεν υπάρχει αποθηκευμένο context, δηλαδή το saga δεν έχει ξεκινήσει ακόμη, τότε το saga ξεκινά μόνο στην περίπτωση που το event είναι OrderCreatedEvent.

Αν υπάρχει αποθηκευμένο context, τότε το state machine επαναφέρεται στην προηγούμενη κατάσταση μέσω του persister ώστε να συνεχίσει η εκτέλεση του saga. Στη συνέχεια, το service στέλνει το event στο state machine με τη βοήθεια της μεθόδου sendEvent.

Η μέθοδος mapToSagaMessage μετατρέπει τα events σε κατάλληλα μηνύματα για να σταλούν στο state machine, προσθέτοντας επιπλέον πληροφορίες στο header όταν χρειάζεται (π.χ. το transactionId αν πρόκειται για PaymentAuthorizedEvent). Συνολικά, το OrchestratorService διαχειρίζεται την εκτέλεση του saga και εξασφαλίζει ότι κάθε event επεξεργάζεται με σωστό τρόπο και ότι η ροή της διαδικασίας παραμένει συνεπής σε όλη τη διάρκεια του κύκλου ζωής της παραγγελίας.

```
@Log4j2
@Service
@RequiredArgsConstructor
public class OrchestratorService {

    private final StateMachineFactory<SagaState, SagaEvent> stateMachineFactory;
    private final StateMachineRuntimePersister<SagaState, SagaEvent, String> persister;

    public void processSagaEvent(SagaEventMessage event) throws Exception {
        Integer orderId = event.orderId();
        if (orderId == null) {
            log.error("Could not extract orderId from event: {}", event);
            return;
        }

        String orderIdStr = String.valueOf(orderId);
        StateMachine<SagaState, SagaEvent> stateMachine = stateMachineFactory.getStateMachine(
             orderIdStr);
        StateMachineContext<SagaState, SagaEvent> context = persister.read(orderIdStr);

        if (context == null) {
            // No persisted state. Only start a new saga if this is the START_SAGA event
            if (event.sagaEvent().compareTo(SagaEvent.START_SAGA) != 0 || !(event instanceof
                 OrderCreatedEvent)) {
                log.warn("No persisted state for orderId {} and event {} is not a START event.
                     Ignoring.",
                        orderIdStr, event.getClass().getSimpleName());
                return;
            }

            log.info("No persisted state for orderId: {}. Starting new saga.", orderIdStr);
            startSaga((OrderCreatedEvent) event, stateMachine);
            return;
        }

        // Persisted context exists. Restore and process the event
        stateMachine.getStateMachineAccessor()
                .doWithAllRegions(access -> access.resetStateMachineReactively(context).block());

        log.info("Restored state machine for orderId: {} in state: {}", orderIdStr, stateMachine.
```

```
            getState().getId());

        Mono.defer(stateMachine::startReactively)
                .then(sendEvent(stateMachine, mapToSagaMessage(event)))
                .doOnError(e -> log.error("Error processing event for orderId: {}", orderIdStr, e)
                    )
                .onErrorResume(e -> stateMachine.stopReactively().then(Mono.error(e)))
                .then(stateMachine.stopReactively())
                .subscribe();
    }

    private void startSaga(OrderCreatedEvent event, StateMachine<SagaState, SagaEvent>
         stateMachine) {
        Mono.defer(() -> {
                    stateMachine.getExtendedState().getVariables().put(SAGA_CONTEXT_HEADER, event)
                        ;
                    return stateMachine.startReactively();
                })
                .then(sendEvent(stateMachine, mapToSagaMessage(event)))
                .doOnError(error -> log.error("Failed to start saga state machine for order id: {}
                     ", event.getOrderId().toString(), error))
                .onErrorResume(e -> stateMachine.stopReactively().then(Mono.error(e)))
                .then(stateMachine.stopReactively())
                .subscribe();
    }

    private Mono<Void> sendEvent(StateMachine<SagaState, SagaEvent> stateMachine, Message<
         SagaEvent> message) {
        return stateMachine.sendEvent(Mono.just(message)).then();
    }

    private Message<SagaEvent> mapToSagaMessage(SagaEventMessage message) {
        SagaEvent event = message.sagaEvent();
        MessageBuilder<SagaEvent> builder = MessageBuilder.withPayload(event);
        if (message instanceof PaymentAuthorizedEvent eventDto) {
            builder.setHeader("transactionId", eventDto.transactionId());
        }

        return builder.build();
    }
}
```

Απόσπασμα 4.19: Saga Orchestrator Service

4.2 Διαχείριση μηνυμάτων από τις Συμμετέχουσες Υπηρεσίες στο Saga

Σε αυτή την ενότητα περιγράφονται οι listeners των services που επεξεργάζονται τα commands από τον Orchestrator και εκπέμπουν τα αντίστοιχα events μέσω του RabbitMQ.

4.2.1 Order Service Saga Listener

Το Order Service εκπέμπει το αρχικό event δημιουργίας παραγγελίας που ξεκινά την εκτέλεση του saga στον Orchestrator και λαμβάνει saga commands από τον Orchestrator για να ενημερώσει την κατάσταση μιας παραγγελίας.

Το service λαμβάνει messages από το queue order_events_queue. Στο απόσπασμα κώδικα 4.20, φαίνεται ο listener ο οποίος χρησιμοποιεί το annotation `@RabbitListener(queues = ORDER_QUEUE)`, όπου η σταθερά `ORDER_QUEUE` προέρχεται από την κλάση SagaConstants (βλ. απόσπασμα κώδικα 4.22). Το binding του queue, όπως φαίνεται στο απόσπασμα 4.21, γίνεται στο topic exchange `saga.exchange` με routing pattern `order.command.*`, επομένως το Order Service θα παραλάβει οποιοδήποτε message δημοσιευτεί σε αυτό το exchange με routing key αυτής της μορφής.

```java
@Log4j2
@Component
@RabbitListener(queues = ORDER_QUEUE)
@RequiredArgsConstructor
public class SagaCommandListener {

    private final List<SagaCommandHandler<?>> handlers;
    private final Map<Class<?>, SagaCommandHandler<?>> handlerMap = new HashMap<>();

    @PostConstruct
    void init() {
        handlers.forEach(h -> handlerMap.put(h.getCommandType(), h));
        log.info("Registered {} command handlers", handlerMap.size());
    }

    @RabbitHandler(isDefault = true)
    public <T extends SagaMessage> void handleCommand(T command) {
        log.info("Received command: {}", command.getClass().getSimpleName());
        SagaCommandHandler<T> handler = (SagaCommandHandler<T>) handlerMap.get(command.getClass())
            ;
        if (handler == null) {
            log.warn("No handler for command type: {}", command.getClass().getSimpleName());
            return;
        }
        handler.handle(command);
    }
}
```

Απόσπασμα 4.20: Order Service Saga Command Listener

```java
@Configuration
public class RabbitMQConfig {

    @Bean
    public MessageConverter jsonMessageConverter() {
        return new Jackson2JsonMessageConverter();
```

```
    }

    @Bean
    TopicExchange exchange() {
        return new TopicExchange(SAGA_EXCHANGE);
    }

    @Bean
    Queue orderQueue() {
        return new Queue(ORDER_QUEUE, true);
    }

    @Bean
    Binding orderBinding(Queue orderQueue, TopicExchange exchange) {
        return BindingBuilder.bind(orderQueue)
                .to(exchange)
                .with("order.command.*");
    }
}
```

Απόσπασμα 4.21: Order Service RabbitMQ Configuration

```
public final class SagaConstants {

    private SagaConstants() {
    }

    // Exchange
    public static final String SAGA_EXCHANGE = "saga.exchange";

    // Queues
    public static final String ORCHESTRATOR_QUEUE = "saga_orchestrator_events_queue";
    public static final String PAYMENT_QUEUE = "payment_commands_queue";
    public static final String INVENTORY_QUEUE = "inventory_commands_queue";
    public static final String ORDER_QUEUE = "order_events_queue";

    // Routing Keys
    public static final String PAYMENT_AUTHORIZE_KEY = "payment.command.authorize";
    public static final String PAYMENT_CAPTURE_KEY = "payment.command.capture";
    public static final String PAYMENT_VOID_KEY = "payment.command.void";

    public static final String INVENTORY_RESERVE_KEY = "inventory.command.reserve";
    public static final String INVENTORY_RELEASE_KEY = "inventory.command.release";

    public static final String COMPLETE_ORDER_KEY = "order.command.complete";
    public static final String FAIL_ORDER_KEY = "order.command.fail";

    public static final String ORDER_CREATED_EVENT_KEY = "order.event.created";
    public static final String PAYMENT_AUTHORIZED_EVENT_KEY = "payment.event.authorized";
    public static final String PAYMENT_AUTHORIZATION_FAILED_EVENT_KEY = "payment.event.failed";
    public static final String PAYMENT_CAPTURED_EVENT_KEY = "payment.event.captured";
```

```
30      public static final String PAYMENT_CAPTURE_FAILED_EVENT_KEY = "payment.event.capture_failed";
31      public static final String PAYMENT_VOIDED_EVENT_KEY = "payment.event.voided";
32
33      public static final String INVENTORY_RESERVED_EVENT_KEY = "inventory.event.reserved";
34      public static final String INVENTORY_RESERVATION_FAILED_EVENT_KEY = "inventory.event.
            reservation_failed";
35      public static final String INVENTORY_RELEASED_EVENT_KEY = "inventory.event.released";
36  }
```

Απόσπασμα 4.22: Saga Constants

Η διαχείριση των εισερχόμενων commands ακολουθεί παρόμοια λογική με αυτή που χρησιμοποιείται και στον Orchestrator, δηλαδή αυτή των dispatcher και handlers. Συγκεκριμένα, κατά την εκκίνηση της εφαρμογής, ο SagaCommandListener συλλέγει όλα τα διαθέσιμα Spring beans που υλοποιούν το SagaCommandHandler και κατασκευάζει ένα map που αντιστοιχίζει τον τύπο του command με τον αντίστοιχο handler. Όταν ληφθεί ένα μήνυμα, εκτελείται ο κατάλληλος handler βάσει της κλάσης του DTO.

Στην παρούσα υλοποίηση, υπάρχουν δύο command handlers που εκτελούν τις απαραίτητες αλλαγές κατάστασης. Ο Complete Order Command Handler (βλ. 4.23) και ο Fail Order Command Handler (βλ. 4.24). Και οι δύο handlers λαμβάνουν commands που περιέχουν μόνο το orderId, και καλούν τη μέθοδο `updateOrderState()` για να αλλάξουν την κατάσταση της παραγγελίας σε COMPLETED ή FAILED αντίστοιχα. Τα διαθέσιμα order states είναι PENDING, COMPLETED και FAILED, όπως φαίνεται στο απόσπασμα 4.27.

```
1   @Log4j2
2   @RequiredArgsConstructor
3   @Component
4   public class CompleteOrderCommandHandler implements SagaCommandHandler<CompleteOrderCommand> {
5
6       private final OrderService orderService;
7
8       @Override
9       public void handle(CompleteOrderCommand command) {
10          log.info("Handling CompleteOrderCommand for orderId: {}", command.orderId());
11          try {
12              orderService.updateOrderState(command.orderId(), OrderState.COMPLETED);
13          } catch (Exception e) {
14              log.error("Error while completing order with id {}: {}", command.orderId(), e.
                    getMessage());
15          }
16      }
17
18      @Override
19      public Class<CompleteOrderCommand> getCommandType() {
20          return CompleteOrderCommand.class;
21      }
22  }
```

Απόσπασμα 4.23: Complete Order Command Handler

```java
@Log4j2
@RequiredArgsConstructor
@Component
public class FailOrderCommandHandler implements SagaCommandHandler<FailOrderCommand> {

    private final OrderService orderService;

    @Override
    public void handle(FailOrderCommand command) {
        log.info("Handling FailOrderCommand for orderId: {}", command.orderId());
        try {
            orderService.updateOrderState(command.orderId(), OrderState.FAILED);
        } catch (Exception e) {
            log.error("Error while failing order with id {}: {}", command.orderId(), e.getMessage
                ());
        }
    }

    @Override
    public Class<FailOrderCommand> getCommandType() {
        return FailOrderCommand.class;
    }
}
```

Απόσπασμα 4.24: Fail Order Command Handler

```java
public record CompleteOrderCommand(Integer orderId)
    implements SagaMessage {}
```

Απόσπασμα 4.25: Complete Order Command

```java
public record FailOrderCommand(Integer orderId)
    implements SagaMessage {}
```

Απόσπασμα 4.26: Fail Order Command

```java
public enum OrderState {
    PENDING,
    COMPLETED,
    FAILED
}
```

Απόσπασμα 4.27: Order State

Η λογική της ενημέρωσης κατάστασης της παραγγελίας βρίσκεται σε επίπεδο service. Η μέθοδος `updateOrderState()` φορτώνει την παραγγελία από τη βάση, ενημερώνει το πεδίο state και

αποθηκεύει την αλλαγή. Με αυτόν τον τρόπο, η διαχείριση των messages παραμένει ανεξάρτητη, ενώ η λογική της ενημέρωσης του μοντέλου στη βάση πραγματοποιείται εντός ενός transaction στο service layer. Στο απόσπασμα 4.28 φαίνεται η μέθοδος `updateOrderState()`.

```
@Override
@Transactional
public OrderDto updateOrderState(Integer orderId, OrderState state) {
    Order order = orderRepository.findById(orderId).orElseThrow(() -> new RuntimeException("Order
        not found"));
    order.setState(state);
    Order updatedOrder = orderRepository.save(order);
    return mapper.toOrderDto(updatedOrder);
}
```

Απόσπασμα 4.28: Update Order State Method

Όσον αφορά τα events προς τον Orchestrator, το Order Service εκπέμπει μόνο το event εκκίνησης του saga κατά τη δημιουργία μιας παραγγελίας. Συγκεκριμένα, εφόσον αποθηκεύσει την παραγγελία, εκπέμπει ένα Order Created Event στο exchange `saga.exchange` με routing key `order.event.created`. Το συγκεκριμένο event DTO (βλ. 4.29) περιέχει τα δεδομένα που χρειάζεται ο Orchestrator για τα επόμενα βήματα.

```
public final class OrderCreatedEvent implements SagaEventMessage, Serializable {
    private Integer orderId;
    private UUID customerId;
    private BigDecimal totalAmount;
    private String paymentToken;
    private List<OrderProductDto> products;
}
```

Απόσπασμα 4.29: Order Created Event

Τέλος, η υπηρεσία διαθέτει logs, τα οποία θα είναι χρήσιμα για την τεκμηρίωση της ορθής λειτουργίας στο κεφάλαιο αξιολόγησης.

4.2.2 Payment Service Saga Listener

Το Payment Service συμμετέχει στο saga λαμβάνοντας commands που στέλνει ο Saga Orchestrator και ανάλογα με το αποτέλεσμα κάθε βήματος πληρωμής, εκπέμπει τα αντίστοιχα events πίσω στον Orchestrator. Στο παράρτημα 4.30, φαίνεται η λογική του listener που είναι παρόμοια με αυτή του Order Service.

```
@Log4j2
@Component
@RabbitListener(queues = PAYMENT_QUEUE)
@RequiredArgsConstructor
public class SagaCommandListener {

    private final List<SagaCommandHandler<?>> handlers;
```

```
    private final Map<Class<?>, SagaCommandHandler<?>> handlerMap = new HashMap<>();

    @PostConstruct
    void init() {
        handlers.forEach(h -> handlerMap.put(h.getCommandType(), h));
        log.info("Registered {} command handlers", handlerMap.size());
    }

    @RabbitHandler(isDefault = true)
    public <T extends SagaMessage> void handleCommand(T command) {
        log.info("Received command: {}", command.getClass().getSimpleName());
        SagaCommandHandler<T> handler = (SagaCommandHandler<T>) handlerMap.get(command.getClass())
            ;
        if (handler == null) {
            log.warn("No handler for command type: {}", command.getClass().getSimpleName());
            return;
        }
        handler.handle(command);
    }
}
```

Απόσπασμα 4.30: Payment Service Saga Command Listener

```
@Configuration
public class RabbitMQConfig {

    @Bean
    public MessageConverter jsonMessageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    @Bean
    TopicExchange exchange() {
        return new TopicExchange(SAGA_EXCHANGE);
    }

    @Bean
    Queue paymentQueue() {
        return new Queue(PAYMENT_QUEUE, true);
    }

    @Bean
    Binding paymentBinding(Queue paymentQueue, TopicExchange exchange) {
        return BindingBuilder.bind(paymentQueue)
                .to(exchange)
                .with("payment.command.*");
    }
}
```

Απόσπασμα 4.31: Payment Service RabbitMQ Configuration

Η λογική επεξεργασίας των εισερχόμενων commands ακολουθεί το ίδιο μοτίβο με το Order Service. Συγκεκριμένα, ο listener λειτουργεί ως μηχανισμός δρομολόγησης στους αντίστοιχους handlers. Σε αυτό το service υπάρχουν τρεις saga command handlers. Ο Authorize Payment Command Handler (βλ. 4.32), ο οποίος λαμβάνει ένα Authorize Payment Command και εκκινεί την λογική της εξουσιοδότησης πληρωμής. Αν η εξουσιοδότηση ολοκληρωθεί επιτυχώς, εκπέμπεται ένα Payment Authorized Event πίσω στον Orchestrator. Σε αντίθετη περίπτωση, εκπέμπεται ένα Payment Authorization Failed Event. Αντίστοιχα, ο Capture Payment Command Handler (βλ. 4.33), λαμβάνει ένα Capture Payment Command και καλεί την μέθοδο `capturePayment()`. Ανάλογα το αποτέλεσμα, δημοσιεύει ένα Payment Captured Event ή ένα Payment Capture Failed Event. Τέλος, ο Void Payment Command Handler (βλ. 4.34) λαμβάνει ένα Void Payment Command και καλεί την μέθοδο `voidPayment()`. Σε περίπτωση επιτυχίας, εκπέμπει ένα Payment Voided Event. Σε περίπτωση αποτυχίας, όμως, δεν εκπέμπει ξεχωριστό failure event στην τρέχουσα υλοποίηση αλλά γίνεται καταγραφή του σφάλματος στα logs του συστήματος.

```
@Log4j2
@RequiredArgsConstructor
@Component
public class AuthorizePaymentCommandHandler implements SagaCommandHandler<AuthorizePaymentCommand>
      {

    private final PaymentService paymentService;
    private final RabbitTemplate rabbitTemplate;

    @Override
    public void handle(AuthorizePaymentCommand command) {
        try {
            TransactionDto transaction = paymentService.authorizePayment(command);
            log.info("Dispatching PaymentAuthorizedEvent for orderId: {}", command.orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    PAYMENT_AUTHORIZED_EVENT_KEY,
                    new PaymentAuthorizedEvent(
                            command.orderId(),
                            transaction.id().toString()
                    )
            );
        } catch (PaymentException e) {
            log.error("Payment authorization failed for orderId: {}", command.orderId(), e);
            log.info("Dispatching PaymentAuthorizationFailedEvent for orderId: {}", command.
                orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    PAYMENT_AUTHORIZATION_FAILED_EVENT_KEY,
                    new PaymentAuthorizationFailedEvent(
                            command.orderId(),
                            e.getMessage()
```

```
                    )
            );
        }
    }

    @Override
    public Class<AuthorizePaymentCommand> getCommandType() {
        return AuthorizePaymentCommand.class;
    }
}
```

Απόσπασμα 4.32: Authorize Payment Command Handler

```
@Log4j2
@RequiredArgsConstructor
@Component
public class CapturePaymentCommandHandler implements SagaCommandHandler<CapturePaymentCommand> {

    private final PaymentService paymentService;
    private final RabbitTemplate rabbitTemplate;

    @Override
    public void handle(CapturePaymentCommand command) {
        try {
            paymentService.capturePayment(command);
            log.info("Dispatching PaymentCapturedEvent for orderId: {}", command.orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    PAYMENT_CAPTURED_EVENT_KEY,
                    new PaymentCapturedEvent(command.orderId())
            );
        } catch (PaymentException e) {
            log.error("Payment capture failed for orderId: {}", command.orderId(), e);
            log.info("Dispatching PaymentCaptureFailedEvent for orderId: {}", command.orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    PAYMENT_CAPTURE_FAILED_EVENT_KEY,
                    new PaymentCaptureFailedEvent(
                            command.orderId(),
                            e.getMessage()
                    )
            );
        }
    }

    @Override
    public Class<CapturePaymentCommand> getCommandType() {
        return CapturePaymentCommand.class;
    }
```

```
}
```

Απόσπασμα 4.33: Capture Payment Command Handler

```
@Log4j2
@RequiredArgsConstructor
@Component
public class VoidPaymentCommandHandler implements SagaCommandHandler<VoidPaymentCommand> {

    private final PaymentService paymentService;
    private final RabbitTemplate rabbitTemplate;

    @Override
    public void handle(VoidPaymentCommand command) {
        try {
            paymentService.voidPayment(command);
            log.info("Dispatching PaymentVoidedEvent for orderId: {}", command.orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    PAYMENT_VOIDED_EVENT_KEY,
                    new PaymentVoidedEvent(command.orderId())
            );
        } catch (PaymentException e) {
            log.error("Payment voiding failed for orderId: {}", command.orderId(), e);
        }
    }

    @Override
    public Class<VoidPaymentCommand> getCommandType() {
        return VoidPaymentCommand.class;
    }
}
```

Απόσπασμα 4.34: Void Payment Command Handler

Τα DTOs των commands περιέχουν τα ελάχιστα απαιτούμενα δεδομένα για κάθε βήμα. Το Authorize Payment Command μεταφέρει τα orderId, amount, paymentMethod και idempotencyId. Τα Capture Payment Command και Void Payment Command μεταφέρουν τα orderId, transactionId και idempotencyId. Τα events που εκπέμπονται περιγράφουν καθαρά το αποτέλεσμα κάθε βήματος και αντιστοιχούν σε συγκεκριμένα saga events, ώστε ο Orchestrator να προωθεί το state machine στο επόμενο state. Ένα παράδειγμα ενός event DTO φαίνεται στο απόσπασμα κώδικα 4.35.

```
public record PaymentAuthorizedEvent(Integer orderId, String transactionId) implements
     SagaEventMessage {
    @Override
    public SagaEvent sagaEvent() {
        return SagaEvent.PAYMENT_AUTHORIZED;
    }
}
```

Απόσπασμα 4.35: Payment Authorized Event

Η επιχειρισιακή λογική των μεθόδων που αναφέρθηκαν παραπάνω βρίσκεται στο GitHub repository του συστήματος [44] και δεν αναφέρονται εδώ επειδή είναι εκτός του στόχου της παρούσας πτυχιακής εργασίας.

Σε αντίθεση με το Order Service, στο Payment Service έχει υλοποιηθεί μηχανισμός idempotency για την ασφαλή επανάληψη των saga commands. Συγκεκριμένα, στην οντότητα Transaction υπάρχει πεδίο idempotencyId το οποίο είναι μοναδικό και υποχρεωτικό. Πριν από την εκτέλεση κάθε ενέργειας, το Payment Service ελέγχει αν το command έχει ήδη εκτελεστεί. Έτσι, πιθανές επαναποστολές του ίδιου command δεν οδηγούν σε διπλές χρεώσεις ή σε ασυνεπή ενημέρωση της κατάστασης. Επιπλέον, οι αποτυχίες των πληρωμών αποτυπώνονται και σε επίπεδο persistence μέσω του status στο Transaction μοντέλο. Οι καταστάσεις που μπορεί να λάβει ένα Payment φαίνονται στο απόσπασμα 4.36.

```
public enum PaymentStatus {
    PENDING,
    AUTHORIZED,
    PAID,
    FAILED,
    VOIDED
}
```

Απόσπασμα 4.36: Payment Status

Τέλος, όπως και στο Order Service, υπάρχουν logs σε όλα τα σημαντικά σημεία της ροής ώστε να αξιοποιηθούν για την τεκμηρίωση της συμπεριφοράς στο κεφάλαιο της αξιολόγησης.

4.2.3 Product Service Saga Listener

Το Product Service διαχειρίζεται τα commands που στέλνει ο Orchestrator σχετικά με το απόθεμα των προϊόντων. Όπως και στα προηγούμενα services, λαμβάνει τα messages μέσω του δικού του listener (βλ. 4.37).

```
@Log4j2
@Component
@RabbitListener(queues = INVENTORY_QUEUE)
@RequiredArgsConstructor
public class SagaCommandListener {

    private final List<SagaCommandHandler<?>> handlers;
    private final Map<Class<?>, SagaCommandHandler<?>> handlerMap = new HashMap<>();

    @PostConstruct
    void init() {
        handlers.forEach(h -> handlerMap.put(h.getCommandType(), h));
```

```
13          log.info("Registered {} command handlers", handlerMap.size());
14      }
15
16      @RabbitHandler(isDefault = true)
17      public <T extends SagaMessage> void handleCommand(T command) {
18          log.info("Received command: {}", command.getClass().getSimpleName());
19          SagaCommandHandler<T> handler = (SagaCommandHandler<T>) handlerMap.get(command.getClass())
                ;
20          if (handler == null) {
21              log.warn("No handler for command type: {}", command.getClass().getSimpleName());
22              return;
23          }
24          handler.handle(command);
25      }
26  }
```

Απόσπασμα 4.37: Product Service Saga Command Listener

```
1   @Configuration
2   public class RabbitMQConfig {
3
4       @Bean
5       public MessageConverter jsonMessageConverter() {
6           return new Jackson2JsonMessageConverter();
7       }
8
9       @Bean
10      TopicExchange exchange() {
11          return new TopicExchange(SAGA_EXCHANGE);
12      }
13
14      @Bean
15      Queue inventoryQueue() {
16          return new Queue(INVENTORY_QUEUE, true);
17      }
18
19      @Bean
20      Binding inventoryBinding(Queue inventoryQueue, TopicExchange exchange) {
21          return BindingBuilder.bind(inventoryQueue)
22                  .to(exchange)
23                  .with("inventory.command.*");
24      }
25  }
```

Απόσπασμα 4.38: Product Service RabbitMQ Configuration

Για τη διαχείριση των ενεργειών του αποθέματος υπάρχουν δύο βασικοί handlers. Ο Reserve Inventory Command Handler, ο οποίος λαμβάνει ένα Reserve Inventory Command και καλεί τη μέθοδο `reserveProductStock()` και τέλος στέλνει ένα Inventory Reserved Event ή ένα Inventory Reservation Failed Event, ανάλογα το αποτέλεσμα.

```java
@Log4j2
@RequiredArgsConstructor
@Component
public class ReserveInventoryCommandHandler implements SagaCommandHandler<ReserveInventoryCommand>
      {

    private final InventoryService inventoryService;
    private final RabbitTemplate rabbitTemplate;

    @Override
    public void handle(ReserveInventoryCommand command) {
        try {
            inventoryService.reserveProductStock(command.orderId(), command.idempotencyId(),
                 command.products());
            log.info("Dispatching InventoryReservedEvent for orderId: {}", command.orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    INVENTORY_RESERVED_EVENT_KEY,
                    new InventoryReservedEvent(
                            command.orderId()
                    )
            );
        } catch (Exception e) {
            log.error("Inventory reservation failed for orderId: {}", command.orderId(), e);
            log.info("Dispatching InventoryReservationFailedEvent for orderId: {}", command.
                 orderId());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    INVENTORY_RESERVATION_FAILED_EVENT_KEY,
                    new InventoryReservationFailedEvent(
                            command.orderId(),
                            e.getMessage()
                    )
            );
        }
    }

    @Override
    public Class<ReserveInventoryCommand> getCommandType() {
        return ReserveInventoryCommand.class;
    }
}
```

Απόσπασμα 4.39: Reserve Inventory Command Handler

Ο Release Inventory Command Handler υλοποιεί την αντισταθμιστική ενέργεια και λαμβάνει ένα Release Inventory Command. Ο handler καλεί την μέθοδο `releaseReservedStock()` και σε περίπτωση επιτυχίας, εκπέμπει ένα Inventory Released Event. Σε περίπτωση αποτυχίας, γίνεται

καταγραφή του σφάλματος, χωρίς να εκπέμπεται ξεχωριστό event αποτυχίας.

```
@Log4j2
@RequiredArgsConstructor
@Component
public class ReleaseInventoryCommandHandler implements SagaCommandHandler<ReleaseInventoryCommand>
      {

    private final InventoryService inventoryService;
    private final RabbitTemplate rabbitTemplate;

    @Override
    public void handle(ReleaseInventoryCommand command) {
        try {
            inventoryService.releaseReservedStock(command.orderId(), command.idempotencyId(),
                 command.products());

            rabbitTemplate.convertAndSend(
                    SAGA_EXCHANGE,
                    INVENTORY_RELEASED_EVENT_KEY,
                    new InventoryReleasedEvent(
                            command.orderId()
                    )
            );
        } catch (Exception e) {
            log.error("Inventory release failed for orderId: {}", command.orderId(), e);
        }
    }

    @Override
    public Class<ReleaseInventoryCommand> getCommandType() {
        return ReleaseInventoryCommand.class;
    }
}
```

Απόσπασμα 4.40: Release Inventory Command Handler

Όπως και στα προηγούμενα services, τα αντίστοιχα command DTOs μεταφέρουν τα απαραίτητα δεδομένα που θα χρειαστεί ο Orchestrator για να περάσει στο επόμενο βήμα του saga. Επίσης, η επιχειρησιακή λογική βρίσκεται σε επίπεδο service. Η κλάση InventoryServiceImpl έχει τις μεθόδους `reserveProductStock()` και `releaseReservedStock()`, οι οποίες πραγματοποιούν τις ενέργειες που χρειάζεται το saga. Ο κώδικας για αυτές τις μεθόδους δεν έχει συμπεριληφθεί σε αυτή την εργασία, όμως υπάρχει στο GitHub repository του project [44].

5. Αξιολόγηση

Αυτό το κεφάλαιο εστιάζει στην αξιολόγηση της υλοποίησης του Saga Orchestrator. Η αξιολόγηση είναι απλή και εστιάζει σε δύο κύρια σενάρια. Στο πρώτο σενάριο, θα γίνει έλεγχος πως ο Orchestrator εκτελεί σωστά την «ιδανική» ροή και η παραγγελία καταλήγει σε κατάσταση COMPLETED. Στο δεύτερο σενάριο, θα γίνει έλεγχος ότι ο Orchestrator ενεργοποιεί τις σωστές αντισταθμιστικές ενέργειες ώστε το σύστημα να μην μείνει σε ασυνεπή κατάσταση, βάσει των θεωρητικών κανόνων που αναλύθηκαν στην βιβλιογραφική ανασκόπηση.

5.1 Περιβάλλον και Μεθοδολογία Δοκιμών

Οι δοκιμές εκτελέστηκαν σε τοπικό περιβάλλον, προσομοιώνοντας ένα κατανεμημένο σύστημα με χρήση Docker Compose [5]. Συγκεκριμένα, χρησιμοποιήθηκε το Docker Compose για την ταυτόχρονη εκτέλεση όλων των μικροϋπηρεσιών (Order, Payment, Product, Cart), καθώς και των RabbitMQ, PostgreSQL, MongoDB, Eureka και Keycloak. Το αντίστοιχο αρχείο συμπεριλαμβάνεται στο GitHub repository [44] του project.

Η μεθοδολογία που ακολουθήθηκε για την επαλήθευση των σεναρίων περιλάμβανε τα εξής βήματα:

1. Εισαγωγή αρχικών δεδομένων στις βάσεις (π.χ. δημιουργία προϊόντων με συγκεκριμένο απόθεμα, εγγραφή χρηστών).
2. Κλήση των REST APIs μέσω του εργαλείου Postman [30] για την έναρξη της δημιουργίας παραγγελίας.
3. Καταγραφή των logs του Orchestrator για την επιβεβαίωση των μεταβάσεων της μηχανής καταστάσεων.
4. Έλεγχος της τελικής κατάστασης στις βάσεις δεδομένων των υπηρεσιών (π.χ. αν μειώθηκε το απόθεμα, αν δημιουργήθηκε η εγγραφή πληρωμής, αν η παραγγελία πήρε την σωστή τελική κατάσταση).

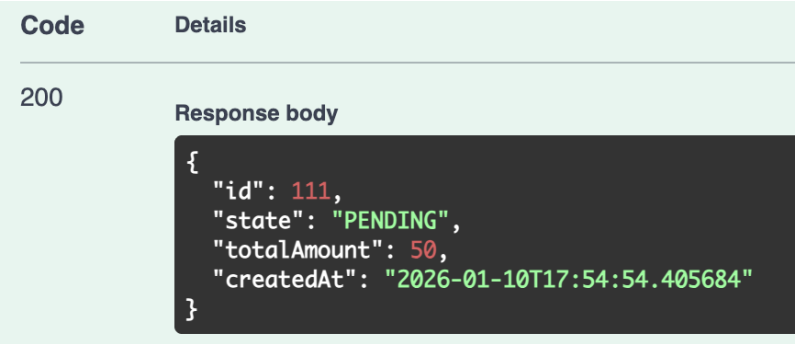
5.2 Σενάρια Αξιολόγησης και Αποτελέσματα

Ο στόχος αυτής της ενότητας είναι η επαλήθευση πως οι μεταβάσεις της μηχανής καταστάσεων γίνονται σωστά υπό κανονικές και μη κανονικές συνθήκες καθώς και ότι τα δεδομένα βρίσκονται σε μια έγκυρη κατάσταση στο τέλος των δοκιμών.

5.2.1 Σενάριο 1: Επιτυχής Δημιουργία Παραγγελίας

Ο στόχος του πρώτου σεναρίου αξιολόγησης είναι η επιβεβαίωση της ορθής λειτουργίας της «ιδανικής» ροής του Saga Orchestrator, όπως αυτή έχει περιγραφεί στο κεφάλαιο σχεδίασης. Συγκεκριμένα, θα εξεταστεί αν ο Orchestrator είναι σε θέση να συντονίσει επιτυχώς τις μικροϋπηρεσίες Order, Payment και Product ώστε μια παραγγελία να μεταβεί από την αρχική κατάσταση CREATED στην τελική κατάσταση COMPLETED ακολουθώντας όλες τις ενδιάμεσες καταστάσεις.

Η εκτέλεση του σεναρίου ξεκινά με την επιτυχή αυθεντικοποίηση του χρήστη μέσω του Keycloak και την αποστολή ενός checkout αιτήματος στο Cart Service. Η απάντηση του συστήματος κατά το στάδιο αυτό παρουσιάζεται στο Σχήμα 5.1, όπου επιβεβαιώνεται η επιτυχής έναρξη της ροής δημιουργίας παραγγελίας.



Σχήμα 5.1: Η JSON απάντηση του Cart Service κατά το checkout.

Στη συνέχεια, ο Saga Orchestrator αποστέλλει εντολή δέσμευσης του ποσού πληρωμής προς το Payment Service. Στο Απόσπασμα 5.1 παρουσιάζονται τα logs του Payment Service, όπου φαίνεται η επιτυχής εξουσιοδότηση και τελική χρέωση της πληρωμής, καθώς και η αποστολή του Payment Captured Event. Το γεγονός αυτό επιβεβαιώνει ότι το βήμα της πληρωμής ολοκληρώθηκε χωρίς σφάλματα.

```
2026-01-10T17:54:54.903Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command: AuthorizePaymentCommand
2026-01-10T17:54:55.053Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Authorizing payment for orderId: 111 with amount: 50.00
2026-01-10T17:54:55.054Z  INFO --- m.r.e.p.client.impl.StripeClient : Creating Stripe PaymentIntent for amount: 50.00 with payment method: pm_card_visa
2026-01-10T17:54:56.479Z  INFO --- m.r.e.p.client.impl.StripeClient : PaymentIntent created with ID: pi_3So6NHIUSvPu1gVq0dg4YNy8 and status: requires_capture
2026-01-10T17:54:56.492Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Payment authorized successfully for orderId: 111 with PaymentIntent: pi_3So6NHIUSvPu1gVq0dg4YNy8
2026-01-10T17:54:56.516Z  INFO --- m.r.e.p.h.AuthorizePaymentCommandHandler : Dispatching PaymentAuthorizedEvent for orderId: 111
2026-01-10T17:54:56.664Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command: CapturePaymentCommand
2026-01-10T17:54:56.666Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Capturing payment for orderId: 111
2026-01-10T17:54:56.670Z  INFO --- m.r.e.p.client.impl.StripeClient : Capturing PaymentIntent: pi_3So6NHIUSvPu1gVq0dg4YNy8
2026-01-10T17:54:57.811Z  INFO --- m.r.e.p.client.impl.StripeClient : PaymentIntent pi_3So6NHIUSvPu1gVq0dg4YNy8
```

```
    captured successfully with status: succeeded
2026-01-10T17:54:57.817Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Payment captured successfully for
    orderId: 111
2026-01-10T17:54:57.831Z  INFO --- m.r.e.p.h.CapturePaymentCommandHandler : Dispatching PaymentCapturedEvent
    for orderId: 111
```

Απόσπασμα 5.1: Σενάριο 1: Τα logs του Payment Service

Ακολούθως, ο Orchestrator εκδίδει εντολή δέσμευσης αποθέματος προς το Product Service. Στο Απόσπασμα 5.2 παρατηρείται η επιτυχής δέσμευση του αποθέματος και η αποστολή του Inventory Reserved Event, γεγονός που επιβεβαιώνει ότι το διαθέσιμο απόθεμα μειώθηκε σωστά σύμφωνα με τον σχεδιασμό.

```
2026-01-10T17:54:56.586Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command:
    ReserveInventoryCommand
2026-01-10T17:54:56.590Z  INFO --- m.r.e.p.s.impl.InventoryServiceImpl : Reserving product stock for orderId:
    111
2026-01-10T17:54:56.634Z  INFO --- m.r.e.p.s.impl.InventoryServiceImpl : Product stock reserved successfully
    for orderId: 111
2026-01-10T17:54:56.644Z  INFO --- m.r.e.p.h.ReserveInventoryCommandHandler : Dispatching
    InventoryReservedEvent for orderId: 111
```

Απόσπασμα 5.2: Σενάριο 1: Τα logs του Product Service

Τέλος, ο Orchestrator ολοκληρώνει τη ροή αποστέλλοντας την εντολή ολοκλήρωσης της παραγγελίας στο Order Service. Στο Απόσπασμα 5.3 φαίνεται η παραλαβή του Complete Order Command και η τελική μετάβαση της παραγγελίας στην κατάσταση COMPLETED.

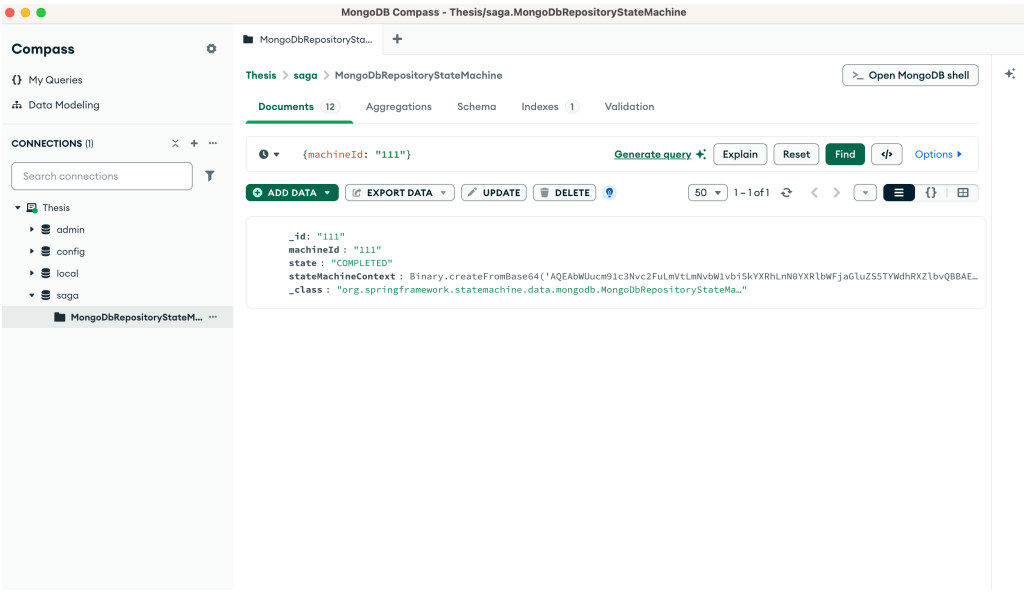
```
2026-01-10T17:54:54.489Z  INFO --- m.r.e.o.service.impl.OrderServiceImpl : Successfully created a new order
    with id: 111
2026-01-10T17:54:54.489Z  INFO --- m.r.e.o.service.impl.OrderServiceImpl : Dispatching OrderCreatedEvent for
    orderId: 111
2026-01-10T17:54:57.865Z  INFO --- m.r.e.o.listener.SagaCommandListener : Received command: CompleteOrderCommand
2026-01-10T17:54:57.866Z  INFO --- m.r.e.o.h.CompleteOrderCommandHandler : Handling CompleteOrderCommand for
    orderId: 111
```

Απόσπασμα 5.3: Σενάριο 1: Τα logs του Order Service

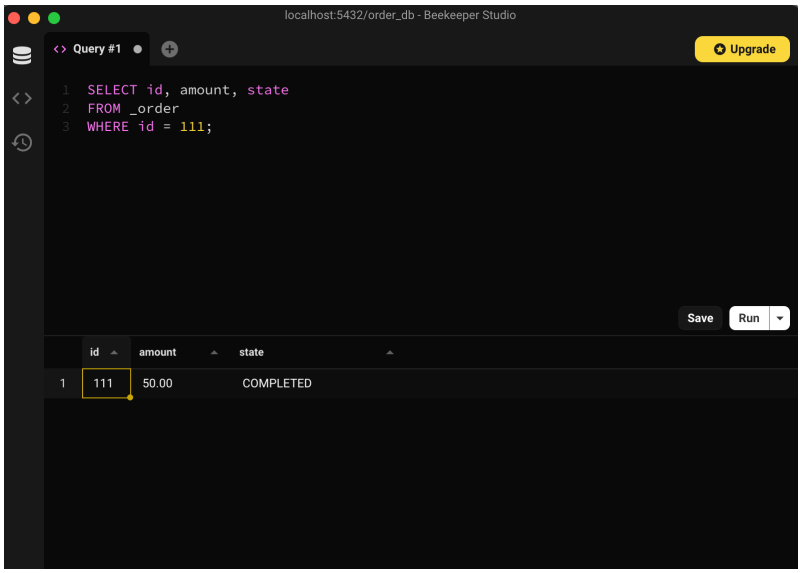
Η ορθή αποθήκευση της κατάστασης της μηχανής επιβεβαιώνεται μέσω της βάσης MongoDB, όπως παρουσιάζεται στην εικόνα 5.2, όπου καταγράφεται η τελική κατάσταση της επιτυχούς ροής. Παράλληλα, στην εικόνα 5.3 παρουσιάζεται η τελική κατάσταση της παραγγελίας στη βάση δεδομένων του Order Service.

Το συγκεκριμένο σενάριο επιβεβαιώνει ότι ο Saga Orchestrator λειτουργεί σύμφωνα με τον σχεδιασμό. Όλες οι επιμέρους ενέργειες εκτελέστηκαν με τη σωστή σειρά και το σύστημα κατέληξε σε μια συνεπή τελική κατάσταση.

```
2026-01-10T17:54:54.597Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
    OrderCreatedEvent
2026-01-10T17:54:54.598Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
    OrderCreatedEvent
2026-01-10T17:54:54.682Z  INFO --- m.r.e.service.OrchestratorService : No persisted state for orderId: 111.
```



Σχήμα 5.2: Προβολή αποθηκευμένης μηχανής κατάστασης επιτυχούς παραγγελίας από το MongoDB Compass.

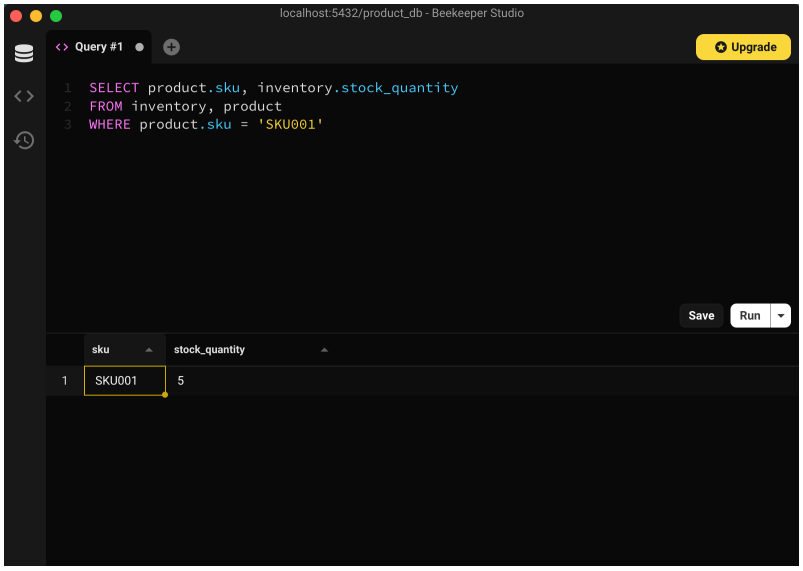


Σχήμα 5.3: Προβολή κατάστασης παραγγελίας στη βάση δεδομένων του Order Service.

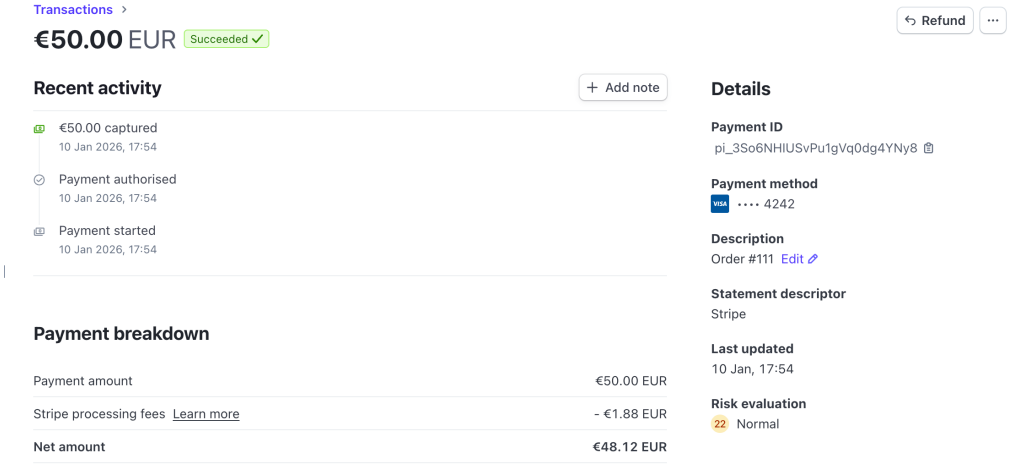
```
     Starting new saga.
2026-01-10T17:54:54.812Z  INFO --- m.r.e.action.AuthorizePaymentAction : Dispatching AuthorizePaymentCommand
     for orderId: 111
2026-01-10T17:54:56.527Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
     PaymentAuthorizedEvent
2026-01-10T17:54:56.527Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
     PaymentAuthorizedEvent
2026-01-10T17:54:56.546Z  INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 111
```

```
      in state: PENDING_PAYMENT_AUTHORIZATION
2026-01-10T17:54:56.550Z  INFO --- m.r.e.action.ReserveInventoryAction : Dispatching ReserveInventoryCommand
      for orderId: 111
2026-01-10T17:54:56.655Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
      InventoryReservedEvent
2026-01-10T17:54:56.655Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
      InventoryReservedEvent
2026-01-10T17:54:56.657Z  INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 111
      in state: PENDING_INVENTORY_RESERVATION
2026-01-10T17:54:56.660Z  INFO --- m.r.e.action.CapturePaymentAction : Dispatching CapturePaymentCommand for
      orderId: 111
2026-01-10T17:54:57.836Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
      PaymentCapturedEvent
2026-01-10T17:54:57.837Z INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
      PaymentCapturedEvent
2026-01-10T17:54:57.841Z INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 111
      in state: PENDING_PAYMENT_CAPTURE
2026-01-10T17:54:57.845Z INFO --- m.r.e.action.CompleteOrderAction : Dispatching CompleteOrderCommand for
      orderId: 111
```

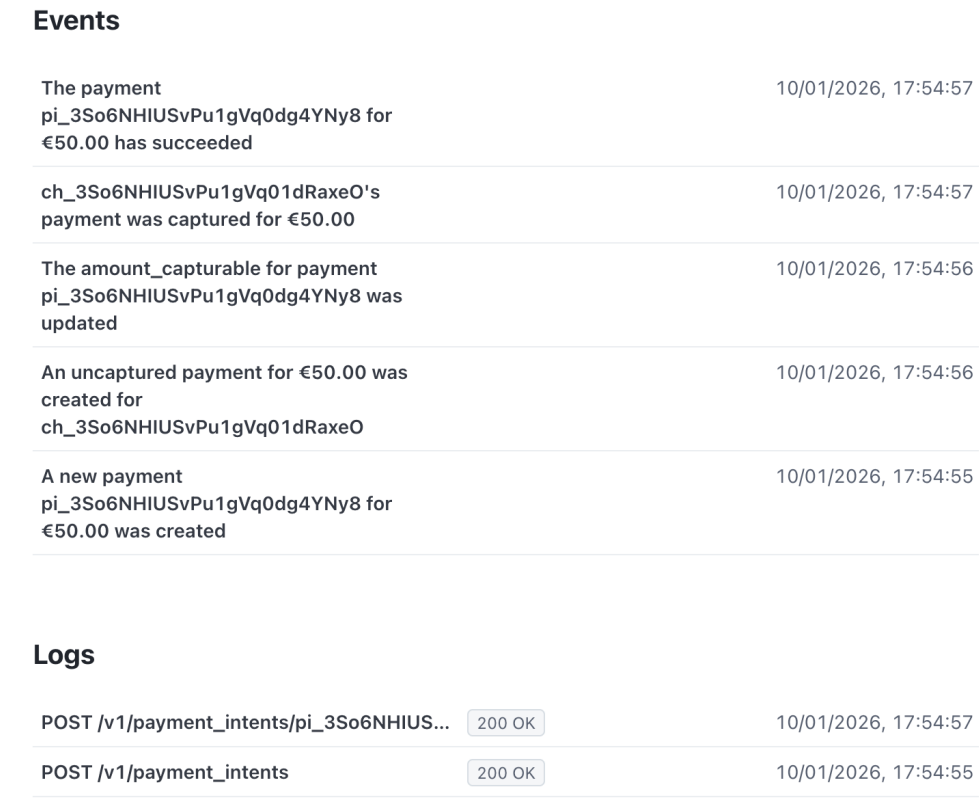
Απόσπασμα 5.4: Σενάριο 1: Τα logs του Saga Orchestrator



Σχήμα 5.4: Προβολή διαθέσιμου αποθέματος προϊόντος στη βάση δεδομένων του Product Service.



Σχήμα 5.5: Προβολή επιτυχούς χρέωσης ποσού παραγγελίας στο dashboard της Stripe.



Σχήμα 5.6: Προβολή των events στο dashboard της Stripe.

5.2.2 Σενάριο 2: Ανεπιτυχής Εκτέλεση

Στόχος αυτού του σεναρίου είναι η αξιολόγηση της συμπεριφοράς του Saga Orchestrator σε περίπτωση αποτυχίας μιας ενδιάμεσης ενέργειας. Συγκεκριμένα, εξετάζεται η ικανότητα του συστήματος να εκτελεί αντισταθμιστικές ενέργειες (compensating transactions) για να επαναφέρει τη συνέπεια των δεδομένων, όπως περιγράφηκε στην θεωρητική ανάλυση του Saga (βλ. ενότητα 3.3.2).

Για την εκτέλεση του σεναρίου, χρειάστηκε να προσομοιάσω ένα exception κατά την δέσμευση του αποθέματος. Για αυτόν τον λόγο έγιναν οι αλλαγές που φαίνονται στο απόσπασμα 5.5 στον Reserve Inventory Command Handler.

```
public void handle(ReserveInventoryCommand command) {
try {
    // inventoryService.reserveProductStock(command.orderId(), command.idempotencyId(), command.
        products());

    log.info("Dispatching InventoryReservedEvent for orderId: {}", command.orderId());
    throw new RuntimeException("Simulated inventory reservation failure");

    // rabbitTemplate.convertAndSend(
    // SAGA_EXCHANGE,
    // INVENTORY_RESERVED_EVENT_KEY,
    // new InventoryReservedEvent(
    // command.orderId()
    // )
    // );
} catch (Exception e) {
    log.error("Inventory reservation failed for orderId: {}", command.orderId(), e);
    log.info("Dispatching InventoryReservationFailedEvent for orderId: {}", command.orderId());

    rabbitTemplate.convertAndSend(
            SAGA_EXCHANGE,
            INVENTORY_RESERVATION_FAILED_EVENT_KEY,
            new InventoryReservationFailedEvent(
                    command.orderId(),
                    e.getMessage()
            )
    );
}
}
```

Απόσπασμα 5.5: Οι αλλαγές του Reserve Inventory Command Handler για το σενάριο 2

Η ροή ξεκινά με τη δημιουργία μιας νέας παραγγελίας και την επιτυχή δέσμευση του ποσού πληρωμής από το Payment Service. Στο Απόσπασμα 5.6 φαίνεται η επιτυχής εξουσιοδότηση της πληρωμής, γεγονός που επιβεβαιώνει ότι το saga προχώρησε κανονικά στο επόμενο βήμα.

```
2026-01-10T18:55:48.205Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command:
     AuthorizePaymentCommand
```

```
2026-01-10T18:55:48.305Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Authorizing payment for orderId: 113 with amount: 10.00
2026-01-10T18:55:48.306Z  INFO --- m.r.e.p.client.impl.StripeClient : Creating Stripe PaymentIntent for amount: 10.00 with payment method: pm_card_visa
2026-01-10T18:55:49.636Z  INFO --- m.r.e.p.client.impl.StripeClient : PaymentIntent created with ID: pi_3So7KCIUSvPu1gVq2Jw4He4b and status: requires_capture
2026-01-10T18:55:49.649Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Payment authorized successfully for orderId: 113 with PaymentIntent: pi_3So7KCIUSvPu1gVq2Jw4He4b
2026-01-10T18:55:49.661Z  INFO --- m.r.e.p.h.AuthorizePaymentCommandHandler : Dispatching PaymentAuthorizedEvent for orderId: 113
2026-01-10T18:55:49.716Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command: VoidPaymentCommand
2026-01-10T18:55:49.718Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Voiding payment for orderId: 113
2026-01-10T18:55:49.721Z  INFO --- m.r.e.p.client.impl.StripeClient : Voiding PaymentIntent: pi_3So7KCIUSvPu1gVq2Jw4He4b
2026-01-10T18:55:50.488Z  INFO --- m.r.e.p.client.impl.StripeClient : PaymentIntent pi_3So7KCIUSvPu1gVq2Jw4He4b voided successfully with status: canceled
2026-01-10T18:55:50.493Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Payment voided successfully for orderId: 113
2026-01-10T18:55:50.505Z  INFO --- m.r.e.p.h.VoidPaymentCommandHandler : Dispatching PaymentVoidedEvent for orderId: 113
```

Απόσπασμα 5.6: Σενάριο 2: Τα logs του Payment Service κατά το Authorize Command

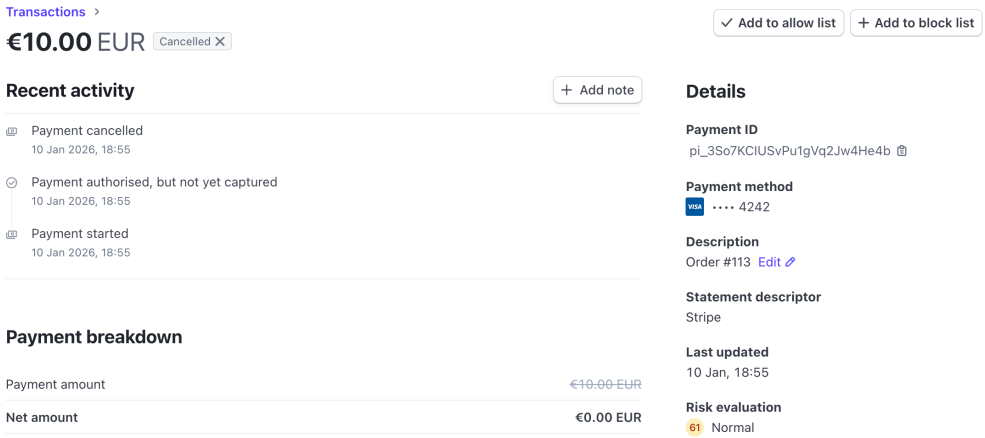
Κατά το στάδιο της δέσμευσης αποθέματος, το Product Service αποτυγχάνει και το σφάλμα αυτό οδηγεί στην αποστολή του Inventory Reservation Failed Event προς τον orchestrator, ο οποίος με τη σειρά του ξεκινά τη λογική αντιστάθμισης. Στο πλαίσιο αυτό, ο orchestrator αποστέλλει εντολή ακύρωσης της προηγούμενης πληρωμής. Στο Απόσπασμα 5.7 παρουσιάζεται η εκτέλεση της εντολής Void Payment Command και η επιτυχής ακύρωση της χρέωσης μέσω του Stripe, γεγονός που επιβεβαιώνεται και στο dashboard του Stripe στην εικόνα 5.7.

```
2026-01-10T18:55:49.716Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command: VoidPaymentCommand
2026-01-10T18:55:49.718Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Voiding payment for orderId: 113
2026-01-10T18:55:49.721Z  INFO --- m.r.e.p.client.impl.StripeClient : Voiding PaymentIntent: pi_3So7KCIUSvPu1gVq2Jw4He4b
2026-01-10T18:55:50.488Z  INFO --- m.r.e.p.client.impl.StripeClient : PaymentIntent pi_3So7KCIUSvPu1gVq2Jw4He4b voided successfully with status: canceled
2026-01-10T18:55:50.493Z  INFO --- m.r.e.p.service.impl.PaymentServiceImpl : Payment voided successfully for orderId: 113
2026-01-10T18:55:50.505Z  INFO --- m.r.e.p.h.VoidPaymentCommandHandler : Dispatching PaymentVoidedEvent for orderId: 113
```

Απόσπασμα 5.7: Σενάριο 2: Τα logs του Payment Service κατά το Void Command

Τέλος, το Order Service λαμβάνει την εντολή αποτυχίας και αλλάζει την κατάσταση της παραγγελίας σε FAILED, όπως φαίνεται στο Απόσπασμα 5.8. Η τελική κατάσταση της μηχανής καταστάσεων παρουσιάζεται στην εικόνα 5.8 και στην εικόνα 5.9 επιβεβαιώνεται η αποθήκευση της αποτυχημένης κατάστασης στη βάση δεδομένων του Order Service.

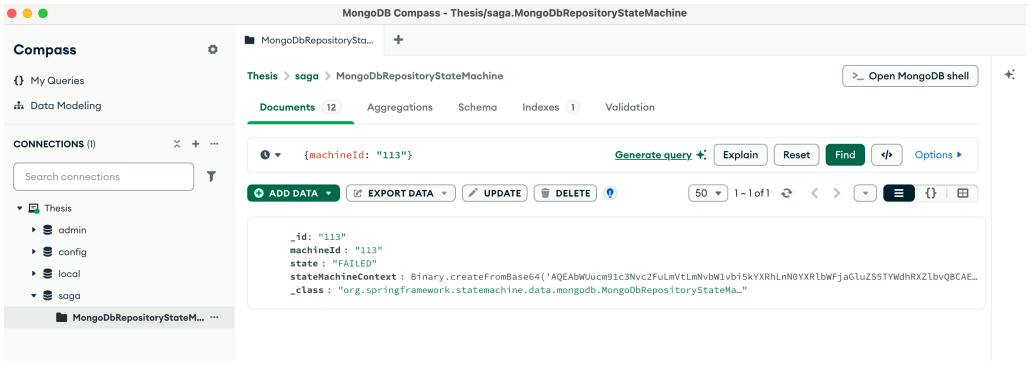
```
2026-01-10T18:55:47.904Z  INFO --- m.r.e.o.service.impl.OrderServiceImpl : Successfully created a new order with id: 113
2026-01-10T18:55:47.904Z  INFO --- m.r.e.o.service.impl.OrderServiceImpl : Dispatching OrderCreatedEvent for orderId: 113
2026-01-10T18:55:50.533Z  INFO 8 --- m.r.e.o.listener.SagaCommandListener : Received command: FailOrderCommand
2026-01-10T18:55:50.533Z  INFO 8 --- m.r.e.o.handler.FailOrderCommandHandler : Handling FailOrderCommand for
```



Σχήμα 5.7: Σενάριο 2: Προβολή ακυρωμένης χρέωσης ποσού παραγγελίας στο dashboard της Stripe.

```
    orderId: 113
```

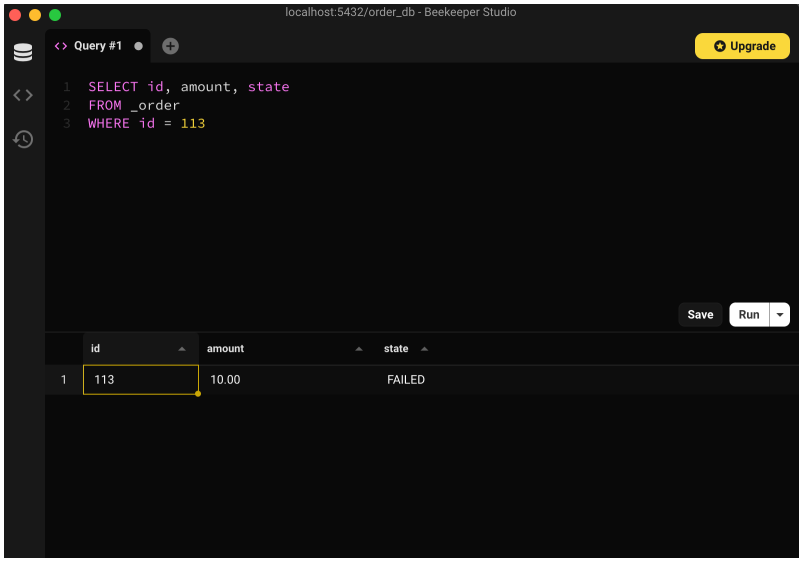
Απόσπασμα 5.8: Σενάριο 2: Τα logs του Order Service



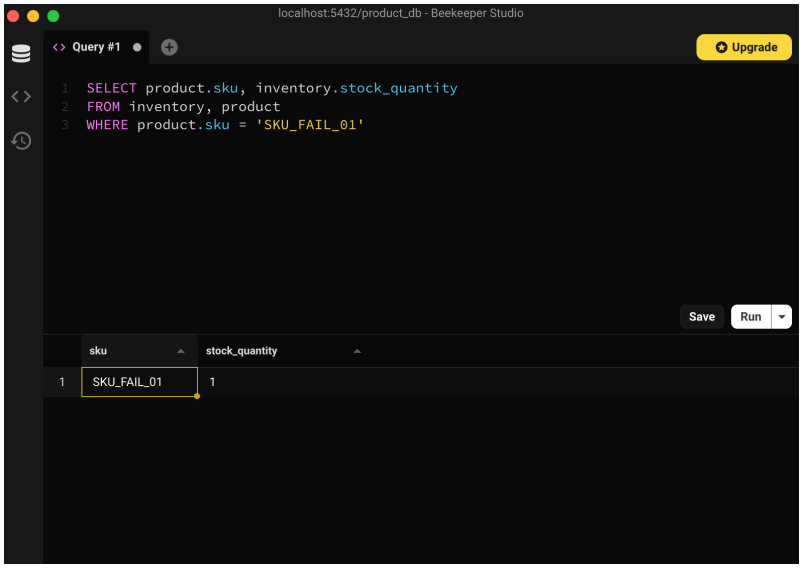
Σχήμα 5.8: Σενάριο 2: Προβολή αποθηκευμένης μηχανής κατάστασης αποτυχημένης παραγγελίας από το MongoDB Compass.

Στο Σχήμα 5.10 παρουσιάζεται το διαθέσιμο απόθεμα του προϊόντος στη βάση του Product Service, όπου επιβεβαιώνεται ότι το απόθεμα δεν μειώθηκε, καθώς η δέσμευση δεν ολοκληρώθηκε επιτυχώς.

```
2026-01-10T18:55:47.985Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
    OrderCreatedEvent
2026-01-10T18:55:47.985Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
    OrderCreatedEvent
2026-01-10T18:55:48.051Z  INFO --- m.r.e.service.OrchestratorService : No persisted state for orderId: 113.
    Starting new saga.
2026-01-10T18:55:48.163Z  INFO --- m.r.e.action.AuthorizePaymentAction : Dispatching AuthorizePaymentCommand
```



Σχήμα 5.9: Σενάριο 2: Προβολή κατάστασης παραγγελίας στη βάση δεδομένων του Order Service.



Σχήμα 5.10: Σενάριο 2: Προβολή διαθέσιμου αποθέματος προϊόντος στη βάση δεδομένων του Product Service.

```
    for orderId: 113
2026-01-10T18:55:49.670Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
    PaymentAuthorizedEvent
2026-01-10T18:55:49.670Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
    PaymentAuthorizedEvent
2026-01-10T18:55:49.684Z  INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 113
    in state: PENDING_PAYMENT_AUTHORIZATION
2026-01-10T18:55:49.687Z  INFO --- m.r.e.action.ReserveInventoryAction : Dispatching ReserveInventoryCommand
    for orderId: 113
```

```
2026-01-10T18:55:49.709Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
    InventoryReservationFailedEvent
2026-01-10T18:55:49.709Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
    InventoryReservationFailedEvent
2026-01-10T18:55:49.711Z  INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 113
    in state: PENDING_INVENTORY_RESERVATION
2026-01-10T18:55:49.713Z  INFO --- m.r.e.action.VoidAuthorizationAction : Dispatching VoidAuthorizationCommand
    for orderId: 113
2026-01-10T18:55:50.511Z  INFO --- m.r.e.listener.SagaEventListener : Received saga event in listener:
    PaymentVoidedEvent
2026-01-10T18:55:50.511Z  INFO --- m.r.e.dispatcher.SagaEventDispatcher : Received saga event of type:
    PaymentVoidedEvent
2026-01-10T18:55:50.515Z  INFO --- m.r.e.service.OrchestratorService : Restored state machine for orderId: 113
    in state: PENDING_AUTHORIZATION_VOID
2026-01-10T18:55:50.519Z  INFO --- m.r.e.action.FailOrderAction : Dispatching FailOrderCommand for orderId: 113
```

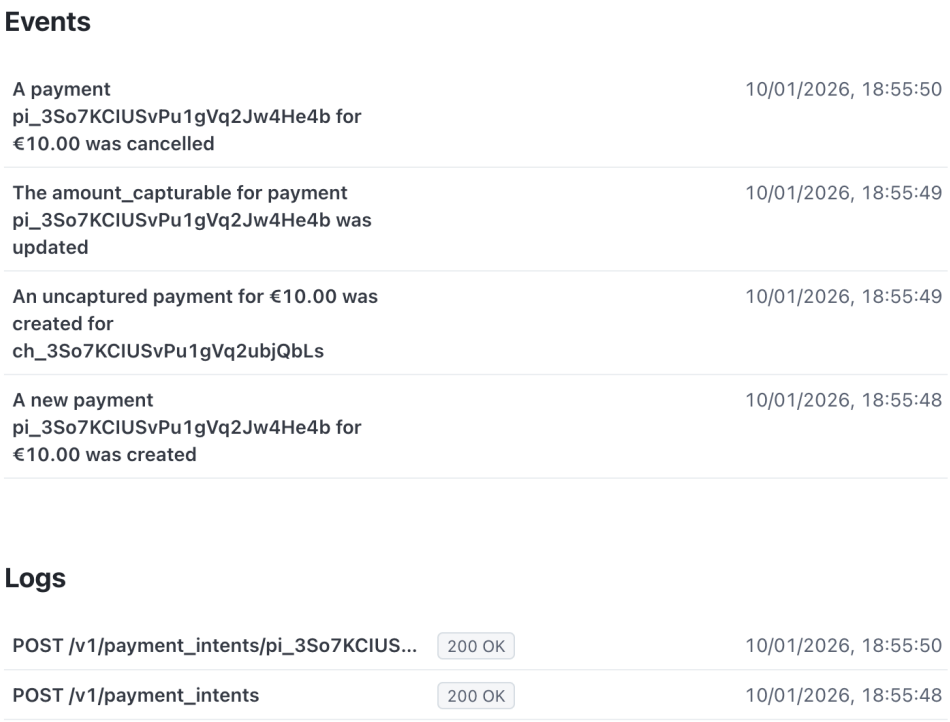
Απόσπασμα 5.9: Σενάριο 2: Τα logs του Saga Orchestrator

```
2026-01-10T18:55:49.698Z  INFO --- m.r.e.p.listener.SagaCommandListener : Received command:
    ReserveInventoryCommand
2026-01-10T18:55:49.699Z  INFO --- m.r.e.p.h.ReserveInventoryCommandHandler : Dispatching
    InventoryReservedEvent for orderId: 113
2026-01-10T18:55:49.699Z ERROR --- m.r.e.p.h.ReserveInventoryCommandHandler : Inventory reservation failed for
    orderId: 113

java.lang.RuntimeException: Simulated inventory reservation failure
    [...]

2026-01-10T18:55:49.700Z  INFO --- m.r.e.p.h.ReserveInventoryCommandHandler : Dispatching
    InventoryReservationFailedEvent for orderId: 113
```

Απόσπασμα 5.10: Σενάριο 2: Τα logs του Product Service

Μετά την ολοκλήρωση της ροής, ελέγχθηκε η κατάσταση στις βάσεις δεδομένων των μικροϋπηρεσιών για να επιβεβαιωθεί η ακεραιότητα του συστήματος.

Το σενάριο αυτό αποδεικνύει ότι ο Saga Orchestrator διαχειρίζεται σωστά τις αποτυχίες. Η αποτυχία της δέσμευσης αποθέματος οδήγησε στην επιτυχή εκτέλεση των αντισταθμιστικών ενεργειών, αποτρέποντας την ασυνέπεια των δεδομένων. Το αποτέλεσμα αυτό είναι σύμφωνο με τον θεωρητικό σχεδιασμό του Saga Pattern και επιβεβαιώνει ότι το σύστημα καταλήγει πάντα σε μια συνεπή τελική κατάσταση, ακόμη και υπό συνθήκες σφάλματος.



Σχήμα 5.11: Σενάριο 2: Προβολή των events στο dashboard της Stripe.

6. Συμπεράσματα και Μελλοντικές Επεκτάσεις

Στο πλαίσιο της παρούσας πτυχιακής εργασίας παρουσιάστηκε ο σχεδιασμός και η υλοποίηση ενός Saga Orchestrator με χρήση μηχανής καταστάσεων σε ένα σύστημα μικροϋπηρεσιών. Στόχος της εργασίας ήταν η κατανόηση και η πρακτική εφαρμογή του προτύπου Saga στην προσέγγιση orchestration, δίνοντας έμφαση στην διατήρηση της τελικής συνέπειας των δεδομένων. Μέσα από την υλοποίηση ενός απλού σεναρίου ηλεκτρονικού εμπορίου, έγινε σαφές πώς ένας κεντρικός Orchestrator μπορεί να συντονίζει τις επιμέρους υπηρεσίες, να διαχειρίζεται επιτυχείς και ανεπιτυχείς ροές και να ενεργοποιεί αντισταθμιστικές ενέργειες όταν αυτό απαιτείται.

Η χρήση της μηχανής καταστάσεων αποδείχθηκε ιδιαίτερα χρήσιμη, καθώς προσέφερε μια καθαρή και δομημένη αναπαράσταση της ροής του Saga. Κάθε βήμα της διαδικασίας είναι σαφώς ορισμένο και οι μεταβάσεις μεταξύ των καταστάσεων καθορίζονται από συγκεκριμένα γεγονότα, κάτι που διευκολύνει την κατανόηση και την επέκταση του συστήματος.

Τα αποτελέσματα αξιολόγησης έδειξαν ότι η υλοποίηση λειτουργεί σωστά τόσο σε σενάριο επιτυχούς εκτέλεσης όσο και σε σενάριο αποτυχίας, στο οποίο ενεργοποιούνται οι κατάλληλες αντισταθμιστικές ενέργειες. Με αυτόν τον τρόπο επιβεβαιώθηκε στην πράξη ότι το πρότυπο Saga αποτελεί μια ρεαλιστική και αποδοτική λύση για τη διαχείριση κατανεμημένων συναλλαγών.

Παρόλο που η εργασία καλύπτει τον βασικό σχεδιασμό και μια λειτουργική υλοποίηση, υπάρχουν αρκετά περιθώρια για μελλοντικές επεκτάσεις. Μια πιθανή κατεύθυνση είναι η υποστήριξη πιο σύνθετων σεναρίων Saga με περισσότερες υπηρεσίες και εναλλακτικές ροές εκτέλεσης. Επιπλέον, θα μπορούσε να προστεθεί καλύτερη παρακολούθηση της εκτέλεσης μέσω εργαλείων monitoring, ώστε να υπάρχει μεγαλύτερη ορατότητα στην πορεία κάθε Saga. Τέλος, θα είχε ενδιαφέρον η σύγκριση της παρούσας υλοποίησης με έτοιμα Saga frameworks, τόσο σε επίπεδο πολυπλοκότητας όσο και απόδοσης, προκειμένου να εξαχθούν πιο γενικά συμπεράσματα.

Συνολικά, η παρούσα εργασία έδειξε στην πράξη πώς μπορεί να υλοποιηθεί ένας Saga Orchestrator σε ένα σύστημα μικροϋπηρεσιών, αναδεικνύοντας τον ρόλο της μηχανής καταστάσεων στον έλεγχο της ροής. Μέσα από την υλοποίηση και την αξιολόγηση του συστήματος, έγινε σαφές ότι το πρότυπο Saga μπορεί να εφαρμοστεί με απλό και κατανοητό τρόπο, προσφέροντας μια λειτουργική λύση για τη διαχείριση κατανεμημένων συναλλαγών.

Βιβλιογραφία

[1] Sahin Aydin και Cem Berke Çebi. ``Comparison of choreography vs orchestration based saga patterns in microservices". Στο: *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*. IEEE. 2022, σσ. 1–6.

[2] Eric A Brewer. ``Towards robust distributed systems". Στο: *PODC*. Τόμ. 7. 10.1145. Portland, OR. 2000, σσ. 343–477.

[3] Mayank Chaturvedi κ.ά. ``From Monolith to Microservices: A Systematic Literature Survey". Στο: *2024 IEEE 3rd International Conference on Data, Decision and Systems (ICDDS)*. IEEE. 2024, σσ. 1–6.

[4] Juan Christian, Afdhal Kurniawan, Maria Susan Anggreainy κ.ά. ``Analyzing Microservices and Monolithic Systems: Key Factors in Architecture, Development, and Operations". Στο: *2023 6th International Conference of Computer and Informatics Engineering (IC2IE)*. IEEE. 2023, σσ. 64–69.

[5] *Docker Compose*. Docker. URL: `https://docs.docker.com/compose/` (επίσκεψη 22/12/2025).

[6] *Docker Documentation*. Docker. URL: `https://docs.docker.com` (επίσκεψη 21/12/2025).

[7] Chowdhury Abida Anjum Era κ.ά. ``A Comprehensive Study of Microservices Architecture in Comparison with SOA and Monolithic Models". Στο: *2025 2nd International Conference on Advanced Innovations in Smart Cities (ICAISC)*. IEEE. 2025, σσ. 1–6.

[8] Weibei Fan κ.ά. ``Method of maintaining data consistency in microservice architecture". Στο: *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing,(HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE Computer Society. 2018, σσ. 47–50.

[9] Diogo Faustino κ.ά. ``Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation". Στο: *Performance Evaluation* 164 (2024), σ. 102411.

[10] Hector Garcia-Molina και Kenneth Salem. ``Sagas". Στο: *ACM Sigmod Record* 16.3 (1987), σσ. 249–259.

[11] Seth Gilbert και Nancy Lynch. ``Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". Στο: *Acm Sigact News* 33.2 (2002), σσ. 51–59.

[12] Kishore Subramanya Hebbar. ``Optimizing Distributed Transactions in Banking APIs: Saga Pattern vs. Two-Phase commit (2PC)". Στο: *The American Journal of Engineering and Technology* 7.06 (2025), σσ. 157–169.

[13] *Hibernate ORM*. URL: `https://hibernate.org/orm/` (επίσκεψη 18/11/2025).

[14] Marek Horváth, Vladyslav Sakhnenko και Filip Gurbáľ. ``Comparison of scalability and performance in Microservices and Monolithic Architectures''. Στο: *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*. IEEE. 2024, σσ. 82–87.

[15] Jurayak Kader κ.ά. ``Comparative Analysis between Monolithic and Microservice Architecture Based on Web Streaming Applications''. Στο: *2025 International Conference on Electrical, Computer and Communication Engineering (ECCE)*. IEEE. 2025, σσ. 1–6.

[16] *Keycloak*. URL: `https://www.keycloak.org` (επίσκεψη 18/11/2025).

[17] Krishna Mohan Koyya και B Muthukumar. ``A survey of saga frameworks for distributed transactions in event-driven microservices''. Στο: *2022 Third International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*. IEEE. 2022, σσ. 1–6.

[18] Wen-Tin Lee κ.ά. ``A High Availability Microservices Architecture Implementation using Saga and Backup Mechanism''. Στο: *2023 10th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE. 2023, σσ. 470–471.

[19] Konstantin Malyuga κ.ά. ``Fault tolerant central saga orchestrator in RESTful architecture''. Στο: *2020 26th Conference of Open Innovations Association (FRUCT)*. IEEE. 2020, σσ. 278–283.

[20] *MongoDB*. URL: `https://www.mongodb.com` (επίσκεψη 03/12/2025).

[21] *MongoDB Compass*. MongoDB. URL: `https://www.mongodb.com/products/tools/compass` (επίσκεψη 03/12/2025).

[22] Philipp Mundhenk κ.ά. ``Reliable distributed systems''. Στο: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, σσ. 287–291.

[23] Antonio Navarro. ``Fundamentals of transaction management in enterprise application architectures''. Στο: *IEEE Access* 10 (2022), σσ. 124305–124332.

[24] Antonio Navarro. ``Local, Global and Saga Transactions for SOA Services and Microservices (* services)''. Στο: *IEEE Access* (2025).

[25] *Netflix Eureka*. Netflix. URL: `https://github.com/Netflix/eureka` (επίσκεψη 18/11/2025).

[26] Sam Newman. *Building Microservices*. Second Edition. Sebastopol, CA: O'Reilly Media, Inc, 2021.

[27] Vinicius L Nogueira κ.ά. ``Insights on microservice architecture through the eyes of industry practitioners''. Στο: *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2024, σσ. 765–777.

[28] *OAuth 2.0 Resource Server*. URL: `https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html` (επίσκεψη 18/11/2025).

[29] *PostgreSQL*. URL: `https://www.postgresql.org` (επίσκεψη 18/11/2025).

[30] *Postman*. URL: `https://www.postman.com/` (επίσκεψη 18/11/2025).

[31] *RabbitMQ*. URL: https://www.rabbitmq.com (επίσκεψη 19/11/2025).

[32] Rudra Pratap Singh κ.ά. ``Monolithic and Microservice Architecture: A Sustainable Approach''. Στο: *2025 3rd IEEE International Conference on Industrial Electronics: Developments & Applications (ICIDeA)*. IEEE. 2025, σσ. 1–6.

[33] *Spring Boot*. Spring. URL: https://spring.io/projects/spring-boot (επίσκεψη 18/11/2025).

[34] *Spring Cloud Gateway*. Spring. URL: https://spring.io/projects/spring-cloud-gateway (επίσκεψη 18/11/2025).

[35] *Spring Cloud LoadBalancer*. Spring. URL: https://docs.spring.io/spring-cloud-commons/reference/spring-cloud-commons/loadbalancer.html (επίσκεψη 18/11/2025).

[36] *Spring Cloud Netflix Eureka Client*. URL: https://central.sonatype.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-eureka-client (επίσκεψη 18/11/2025).

[37] *Spring Cloud Netflix Eureka Server*. URL: https://central.sonatype.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-eureka-server (επίσκεψη 18/11/2025).

[38] *Spring Data JPA*. URL: https://spring.io/projects/spring-data-jpa (επίσκεψη 18/11/2025).

[39] *Spring Data MongoDB*. URL: https://spring.io/projects/spring-data-mongodb (επίσκεψη 03/12/2025).

[40] *Spring for RabbitMQ (Spring AMQP)*. Spring. URL: https://spring.io/projects/spring-amqp (επίσκεψη 20/11/2025).

[41] *Spring Statemachine*. Spring. URL: https://spring.io/projects/spring-statemachine (επίσκεψη 18/11/2025).

[42] *Strategy pattern*. URL: https://en.wikipedia.org/wiki/Strategy_pattern (επίσκεψη 18/11/2025).

[43] *Stripe Java SDK*. URL: https://github.com/stripe/stripe-java (επίσκεψη 20/11/2025).

[44] *Thesis Project (source code)*. URL: https://github.com/roussosan/thesis-project.

[45] Victor Velepucha και Pamela Flores. ``A survey on microservices architecture: Principles, patterns and migration challenges''. Στο: *IEEE access* 11 (2023), σσ. 88339–88358.

[46] Yunhui Wang και Lei Yang. ``A distributed transaction performance optimization method''. Στο: *2023 5th International Conference on Electronics and Communication, Network and Computer Technology (ECNCT)*. IEEE. 2023, σσ. 71–75.