Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Τεχνικές για ανάπτυξη εφαρμογής με χρήση microservices** **Application development techniques using microservices**
Ονοματεπώνυμο Φοιτητή	**Σπυρίδων Αγγελόπουλος**
Πατρώνυμο	**Ζαφείριος**
Αριθμός Μητρώου	**ΜΠΣΠ/ 17001**
Επιβλέπων	**Ευθύμιος Αλέπης, Αναπληρωτής Καθηγητής**

Ημερομηνία Παράδοσης **Ιούνιος 2020**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Ευθύμιος Αλέπης Αναπληρωτής Καθηγητής	Μαρία Βίρβου Καθηγήτρια	Κων/νος Πατσάκης Επίκουρος Καθηγητής

Περίληψη

Στον προγραμματισμό μπορεί κανείς να παρατηρήσει με την πάροδο του χρόνου διάφορες τάσεις αφενός στις γλώσσες αλλά και σε διάφορες μορφές αρχιτεκτονικής που χρησιμοποιούνται. Οι γλώσσες που ανά χρονικές περιόδους κυριαρχούν υποδηλώνουν το "πώς", το μέσο δηλαδή με το οποίο γίνεται η ανάπτυξη του λογισμικού. Από την άλλη, η αρχιτεκτονική υποδηλώνει το "τι", ποιος είναι ο σκοπός της ανάπτυξης του λογισμικού. Μέσα από αυτή εμφανίζονται οι στόχοι αλλά και οι περιορισμοί που έχει η ανάπτυξη λογισμικού την αντίστοιχη περίοδο.

Τα τελευταία χρόνια μία από τις επικρατέστερες αρχιτεκτονικές ανάπτυξης πολύπλοκων λογισμικών και εφαρμογών είναι τα microservices. Για να κατανοήσει κάποιος ακριβώς αυτή την αρχιτεκτονική και να μπορέσει να σχεδιάσει το δικό του λογισμικό με αυτή τη μεθοδολογία πρέπει να εμβαθύνει στα εξής:

- **τι** ακριβώς είναι τα microservices
- **πώς** αναδύθηκαν και γιατί σαν κυριαρχούσα αρχιτεκτονική
- σε **ποιες** εφαρμογές ταιριάζουν

  Όταν κατανοήσει αυτά τα σημεία μπορεί να προχωρήσει στο πρακτικό κομμάτι
- πώς να **σχεδιάσει** μία εφαρμογή με microservices
- ποια είναι τα βασικά σημεία τους και πώς να **υλοποιήσει**
- πώς να **εντάξει** αυτό τον τρόπο προγραμματισμού σε ένα ευρύτερο πλαίσιο μέσα σε μία εταιρία/πολύπλοκο σύστημα

Microservices με λίγα λόγια θα μπορούσαμε να πούμε ότι είναι ο τρόπος που ένα σύνθετο μονολιθικό σύστημα μπορεί να αναπτυχθεί σαν ένα σύνολο περισσότερων αλλά απλούστερων εφαρμογών που επικοινωνούν μεταξύ τους.

Άρχισε να εμφανίζεται σαν μεθοδολογία από το 2000 αλλά τα τελευταία χρόνια έχει εδραιωθεί σαν τρόπος ανάπτυξης πολύπλοκων διαδικτυακών εφαρμογών καθώς όλες οι διαδικτυακές εφαρμογές ευρείας (και όχι μόνο) χρήσης την υιοθέτησαν.

Δίνει λύση σε συστήματα όπου η πολυπλοκότητα ξεπερνά τη συνηθισμένη. Όσο μία εφαρμογή μεγαλώνει σε μέγεθος αλλά και σε πολυπλοκότητα η ανάπτυξη νέων feature αλλά και το debugging των ήδη υπαρχόντων γίνεται όλο και πιο δύσκολη και σε κάποιες περιπτώσεις αδύνατη. Σε τέτοιες συνθήκες έρχονται συχνά τα microservices να δώσουν λύσεις.

Για να αρχίσει κάποιος να σχεδιάζει μία εφαρμογή με microservices πρέπει στο μυαλό του να αντικαταστήσει την μονολιθική εφαρμογή που ήξερε με ένα σύνολο απλούστερων εφαρμογών που επικοινωνούν μεταξύ τους. Έτσι τα διαφορετικά ξεχωριστά υποσυστήματα μετατρέπονται σε ξεχωριστές απλούστερες εφαρμογές. Αυτές με τι σειρά τους είναι ευκολότερο να αναπτυχθούν αφού δεν έχουν dependencies σε άλλα τμήματα της παλαιότερης μονολιθικής εφαρμογής.

Καθώς όμως το πλήθος των εφαρμογών μεγαλώνει, πρέπει ο προγραμματιστής να αντιμετωπίσει νέα προγραμματιστικά προβλήματα. Πώς αυτές οι εφαρμογές θα υλοποιηθούν; Πώς θα αναγνωρίζει η μία την άλλη; Πώς θα επικοινωνούν μεταξύ τους; Και το *κυριότερο*, πώς θα αποφύγει εισαγάγει εξαρτήσεις μεταξύ τους ώστε να μην καταλήξει καινούργια μονολιθική εφαρμογή.

Ένας προγραμματιστής σπάνια δουλεύει μόνος του. Και ακόμα πιο σπάνια μία εφαρμογή που χρειάζεται να δομηθεί με microservices αναπτύσσεται μόνο από ένα προγραμματιστή. Αυτό σημαίνει ότι αυτός ο νέος τρόπος αρχιτεκτονικής αλλά και προγραμματισμού πρέπει να καταφέρει να ενταχθεί παραγωγικά σε μία ευρύτερη "γραμμή παραγωγής" που ξεκινά από το coding και φτάνει μέχρι το deployment.

Summary

In programming one can observe over time various trends in languages but also in various forms of architectures being used. Languages that dominate from time to time indicate the "how", that is, the means by which software is developed. On the other hand, architecture indicate "what", what is the purpose of software development. Through it, the goals and the limitations of software development in the respective period appear.

In recent years, one of the most prevalent architectures for developing sophisticated software and applications is **microservices**. In order to understand exactly this architecture and to be able to design your own software with this methodology, one must delve into the following:

- **what** exactly are microservices
- **how** and **why** they emerged and as a dominant architecture
- in **which** applications they fit

Once he understands these points he can move on to the practical part

- how to **design** an application with microservices
- what are their key points and how to **implement**
- how to **integrate** this way of programming in a broader context within a company / complex system

Microservices in short is the way a complex monolithic system can be developed as a set of more but simpler applications that communicate with each other.

It started appearing as a methodology in 2000 but in recent years it has established itself as a way to develop complex online applications as all online applications of wide (and not only) use have adopted it.

It provides a solution in systems where complexity is greater than usual. As an application grows in size and complexity, the development of new features and debugging of existing ones becomes more and more difficult and in some cases impossible. In such conditions, microservices often can give solutions.

In order to start designing an application with microservices, one must replace in one's mind the monolithic application that he knew with a set of simpler applications that communicate with each other. Thus the different separate subsystems are transformed into separate simpler applications. These in turn are easier to develop as they have no dependencies on other parts of the older monolithic application.

But as the number of applications grows, the developer has to deal with new programming problems. How will these applications be implemented? How will they recognize each other? How will they communicate with each other? And how to avoid importing dependencies between them so as not to end up a new monolithic application.

A developer rarely works alone. And even more rarely an application that needs to be structured with microservices is only developed by one programmer. This means that this new way of architecture as well as planning must be able to be productively integrated into a wider "production line" that starts from coding and reaches deployment.

Στόχος

Όπως αναφέρθηκε, τα microservies επιφέρουν ένα paradigm shift στον τρόπο που μέχρι τώρα σκεφτόμασταν την υλοποίηση ενός πολύπλοκου υπολογιστικού συστήματος. Αυτό σημαίνει ότι ο προγραμματιστής που καλείται για πρώτη φορά να αναπτύξει μία τέτοια εφαρμογή θα κληθεί να αντιμετωπίσει πολλά προβλήματα, άλλα μικρά άλλα μεγάλα, πριν καν χρειαστεί να σκεφτεί το business logic της εφαρμογής του.

Σκοπός αυτής της εργασίας είναι να αναπτύξει μία εφαρμογή - πρότυπο που υλοποιεί ένα βασικό business logic με τη βοήθεια των microservices. Το business logic είναι τυπικό ενός συστήματος όπου τα microservices θα ήταν μία καλή αρχιτεκτονική αλλά επίτηδες μένει χαμηλό σε πολυπλοκότητα. Γιατί ταυτόχρονα αναπτύσσεται το κύριο κομμάτι της εργασίας (τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο) που είναι μία συνολική μεθοδολογία προγραμματισμού με microservices που αντιμετωπίζει τα εξής πεδία:

- **επιλογή των διαφορετικών microservices**. Πώς δηλαδή ο προγραμματιστής θα επιλέξει ποιες λειτουργίες πρέπει να αποκοπούν σε ποια microservices.
- **Containerization**. Πώς η χρήση containers είναι ιδανική για τα microservices ώστε διαφορετικά τμήματα να αναπτύσσονται ξεχωριστά αλλά να μπορούν να γίνουν deploy σαν σύνολο.
- **Service discovery**. Πώς διαφορετικές εφαρμογές ανακαλύπτουν η μία την άλλη αλλά και αντιμετωπίζουν το πρόβλημα της παροδικότητας των microservices. Δυνατότητα διασύνδεσης χωρίς hardcoded διευθύνσεις (Discovery client, Rest Templates, Feign client)
- **Externalized configuration**. Τα deployed microservices πρέπει να μπορούν να ελεγχθούν και απομακρυσμένα και να αλλάζει το configuration τους χωρίς να χρειάζονται redeployment ή ανθρώπινη παρέμβαση.
- **Επικοινωνία μεταξύ microservices**. Στην επικοινωνία μεταξύ microservices που λειτουργούν σε ephemeral containers υπάρχει ο κίνδυνος προβληματικά containers να επηρεάζουν τη λειτουργία άλλων. Για την αποφυγή τέτοιων προβλημάτων υπάρχουν μια σειρά από patterns όπως circuit breaker, fallbacks , bulkheads που εύκολα μπορούν να υλοποιηθούν μέσω βιβλιοθηκών όπως το netflix hystrix
- **Microservices design patterns** Ανάλυση διάφορων design pattern που έχουν εμφανιστεί για χρήση με microservices

Επιπρόσθετα υλοποιούνται και προτείνονται και τρόποι επίλυσης πρακτικών προβλημάτων:

- ενσωμάτωση των microservices/containers στο καθημερινό development
- deployment
- debugging σε συστήματα microservices

Στόχος είναι να ο αναγνώστης μέσα από την εργασία και τον αντίστοιχο κώδικα να αποκτήσει ένα **framework** πάνω στο οποίο να μπορέσει να χτίσει τη δική του εφαρμογή με microservies και να μπορέσει να είναι το ίδιο παραγωγικός όσο και πριν στα βασικά τμήματα της ανάπτυξης εφαρμογών:

1. στον αρχιτεκτονικό σχεδιασμό
2. στην υλοποίηση
3. στο deployment
4. στην αντιμετώπιση προβλημάτων (debugging)

Έχοντας διαβάσει την εργασία αυτή σε αντιπαραβολή με τον κώδικα ο αναγνώστης θα μπορεί να:

- κατανοεί τα προβλήματα που οδήγησαν στην ανάπτυξη των microservices
- εντοπίζει εφαρμογές που τα microservices μπορούν όντως αν βοηθήσουν
- χωρίζει αποτελεσματικά μία πολύπλοκη εφαρμογή σε μικρότερα services
- να αναπτύσσει ξεχωριστά τα microservices χρησιμοποιώντας σωστά design patterns
- να ενσωματώνει όλα τα microservies σε ένα "οικοσύστημα"

1 Εισαγωγή στα microservices

Αδιαμφισβήτητα τα microservices γίνονται όλο και πιο δημοφιλή. Ταυτόχρονα υπάρχουν κάποιοι που αμφισβητούν ότι είναι κάτι καινούργιο και υποστηρίζουν ότι είναι απλά ένα εξελιγμένο μοντέλο της Service Oriented Architecture. Ασχέτως αυτού, η αρχιτεκτονική με microservices παρουσιάζει σαφή πλεονεκτήματα, ειδικά στα σε agile μοντέλο ανάπτυξης.

1.1 Μονολιθικές εφαρμογές

Ας πάρουμε για παράδειγμα ότι θέλουμε να δημιουργήσουμε μία εφαρμογή αντίστοιχη του γνωστού Uber. Κατά πάσα πιθανότητα ο σχεδιασμός της θα ακολουθούσε την "εξάγωνη αρχιτεκτονική:"

Στην καρδιά της εφαρμογής είναι το business logic υλοποιημένη σε διάφορα τμήματα που καθορίζουν τα διάφορα services, τα domain objects και τα events. Γύρω από αυτό τον πυρήνα υπάρχουν διάφοροι adapters που λειτουργούν σαν διεπαφή με τον εξωτερικό κόσμο. Τέτοιοι adapters μπορούν να είναι συστήματα που προσφέρουν πρόσβαση σε βάσεις δεδομένων, αποστολείς μηνυμάτων και διαδικτυακοί adapters που είτε υλοποιούν κάποιο API είτε κάποια γραφική διεπαφή.

Παρόλο που η εφαρμογή έχει μία modular (τμηματική) δομή, γίνεται packaged και deployed σαν ένας μονόλιθος. Η δομή του "εκτελέσιμου" διαφέρει από γλώσσα σε γλώσσα και από framework se framework. Μία java εφαρμογή μπορεί να γίνει packaged σε WAR αρχεία και να γίνει deployed σε ένα Tomcat application server.

Εάν η παραπάνω αρχιτεκτονική φαίνεται οικεία είναι γιατί οι εφαρμογές αυτής της μορφής είναι ιδιαίτερα διαδεδομένες. Είναι απλές στην υλοποίηση καθώς τα διάφορα IDE και τα εργαλεία ανάπτυξης λογισμικού έχουν κατασκευαστεί με στόχο κυρίως την ανάπτυξη μίας συνολικής εφαρμογής. Επίσης είναι εύκολο να περάσουν από τεστ. Το μόνο που χρειάζεται για να κάνει κάποιος end-to-end testing είναι απλά να ξεκινήσει την εφαρμογή και να τεστάρει μέσω του UI με κάποιο εργαλείο όπως το selenium. Οι μονολιθικές εφαρμογές είναι από και στο να γίνουν deploy αφού χρειάζεται απλά η μεταφορά της packaged εφαρμογής στον application server. Τέλος όταν χρειάζεται να γίνει scaling της εφαρμογής, μπορούν να τρέχουν παράλληλα πολλές ίδιες εφαρμογές πίσω από ένα load balancer. Και το καλύτερο απ' όλα είναι ότι όσο η εφαρμογή είναι απλή, όντως όλα αυτά δουλεύουν πολύ καλά.

1.2 Τα προβλήματα της μονολιθικής εφαρμογής

Δυστυχώς, η απλουστευμένη προσέγγιση που αναλύσαμε έχει ένα τεράστιο περιορισμό. Οι επιτυχημένες εφαρμογές τείνουν να μεγαλώνουν με το χρόνο και στο τέλος καταλήγουν τεράστιες. Μετά από κάποια χρόνια η μικρή, απλή εφαρμογή έχει καταλήξει σε ένα μονόλιθο τεραστίων διαστάσεων.

Αν και όλοι θέλουμε να δημιουργήσουμε μία πολύ επιτυχημένη εφαρμογή που θα λειτουργεί και θα αναπτύσσεται για χρόνια, δυστυχώς όταν το μέγεθος γίνει πολύ μεγάλο, η περεταίρω ανάπτυξη γίνεται από δύσκολη έως σχεδόν αδύνατη. Το κύριο πρόβλημα πλέον είναι ότι η εφαρμογή έχει καταλήξει να είναι απίστευτα πολύπλοκη. Είναι πάρα πολύ μεγάλη για να την καταλάβει ένας μόνο προγραμματιστής. Σαν αποτέλεσμα η επίλυση προβλημάτων αλλά και η ανάπτυξη νέων λειτουργιών είναι δύσκολη, επίπονη και χρονοβόρα. Επίσης, ακριβώς για αυτό το λόγο, ότι δηλαδή το codebase έχει γίνει υπερβολικά πολύπλοκο, είναι πολύ πιθανό κάθε επιπλέον implementation να μην γίνεται σωστά. Αυτό έχει ως αποτέλεσμα τα λάθη να μην επιλύονται σωστά που με τη σειρά τους οδηγούν σε νέα λάθη.

Εξαιτίας του μεγάλο μεγέθους της εφαρμογής, συνήθως τείνει να μεγαλώνει και ο χρόνος που χρειάζεται να γίνει compiled ή και να ξεκινήσει η εφαρμογή. Κάτι που θα πρέπει να είναι, αν όχι στιγμιαίο, πολύ γρήγορο καταλήγει να παίρνει υπερβολικά πολύ χρόνο. Ίσως κάποιος αναρωτηθεί γιατί είναι πρόβλημα αυτό αφού γίνεται μόνο μία φορά. Αυτό είναι σωστό αλλά για το deployment. Κατά τη διάρκεια του development τόσο το compilation όσο και το startup γίνονται πάρα πολλές φορές. Έτσι, μεγαλώνοντας σε χρόνο, μειώνεται ο πραγματικός χρόνος που μπορεί να αφιερώσει ο developer και αντίστοιχα η παραγωγικότητά του.

Στις σύγχρονες εφαρμογές ένας στόχος που προσπαθούν να πετύχουν οι developers σε συνεργασία με τους devops συναδέλφους τους είναι το continuous deployment. Να μπορούν δηλαδή να μεταφέρουν αλλαγές που έχουν υλοποιηθεί από το development περιβάλλον, στο production μέσα στην ίδια μέρα ή ακόμα καλύτερα πολλές φορές την ημέρα. Σε μία μονολιθική εφαρμογή είναι ιδιαίτερα δύσκολο να πετύχει κανείς κάτι τέτοιο καθώς πρέπει να κάνει compile, package και εν τέλη deploy ολόκληρη την εφαρμογή. Καθώς οι επιμέρους αυτοί χρόνοι αυξάνονται, μικραίνουν τα πιθανά παράθυρα μέσα στην ημέρα για να γίνει το deployment. Επίσης, εξαιτίας της μεγάλης πολυπλοκότητας είναι δύσκολο να εμπιστευτεί κάποιος τις αλλαγές του με αποτέλεσμα να χρειάζεται πολύ testing που τις περισσότερες φορές γίνεται manually.

Και ενώ το scaling ήταν απλό όταν η εφαρμογή είναι απλή, όταν γίνεται πολύπλοκη τα επιμέρους τμήματά της αρχίζουν να "ανταγωνίζονται" το ένα το άλλο. Για παράδειγμα ένα module που θέλει να έχει πρόσβαση σε μια βάση χρειάζεται μεγάλες ταχύτητες I/O. Ένα άλλο module που υλοποιεί πολύπλοκο business logic μπορεί να θέλει ιδιαίτερα καλό επεξεργαστή. Καθώς όμως πρέπει να γίνουν deploy στο ίδιο υπολογιστικό σύστημα (πχ server) πρέπει αναγκαστικά να κάνουν όλα συμβιβασμούς.

Ένα ακόμα αρνητικό της μονολιθική εφαρμογής είναι η αξιοπιστία. Καθώς όλα τα επιμέρους τμήματά της συνυπάρχουν σε ένα υπολογιστικό σύστημα και τρέχουν σαν μία εφαρμογή είναι αναπόφευκτο λάθη του ενός να επηρεάζουν το άλλο. Ένα καταστροφικό bug σε ένα module θα ρίξει ολόκληρη την εφαρμογή. Ένα memory leak θα απορροφήσει όλη τη μνήμη. Ένα deadlock θα σταματήσει όλους τους υπολογισμούς.

Ένας ακόμα λόγος που οι προγραμματιστές αντιπαθούν αυτές τις τεράστιες μονολιθικές εφαρμογές είναι γιατί σε αυτές είναι υπερβολικά δύσκολο να ενταχθούν νέες γλώσσες, frameworks και τεχνολογίες. Όταν υπάρχει ήδη ένα τεράστιο codebase με ένα συγκεκριμένο framework, είναι δύσκολο να αντικατασταθεί από ένα καινούργιο ίσως καλύτερο. Αντίστοιχα όταν ολόκληρη η εφαρμογή είναι σε μια συγκεκριμένη γλώσσα είναι αδύνατο να γίνει το development ενός module σε μια άλλη που ίσως είναι καλύτερη για τη δεδομένη λειτουργία. Σαν αποτέλεσμα η εφαρμογή αυτή τεχνολογικά θα μείνει πίσω αφού δεν μπορεί να υιοθετήσει νέες τεχνολογίες.

Βλέπουμε δηλαδή ότι μία αρχιτεκτονική που προσφέρει πολλά πλεονεκτήματα όσο η πολυπλοκότητα είναι χαμηλή, καταλήγει όταν η πολυπλοκότητα αυξάνεται να αντιστρέφει τα αρχικά της θετικά και σχεδόν να εμποδίζει την περεταίρω ανάπτυξη.

1.3 Αντιμετωπίζοντας την πολυπλοκότητα με Microservices

Μια υπηρεσία, για παράδειγμα το Uber που αναφέραμε πριν, διαθέτει πολλά διαφορετικά τμήματα με ξεχωριστή λειτουργικότητα. Κάθε ένα τέτοιο τμήμα το αντιμετωπίζουμε σαν μία ξεχωριστή εφαρμογή. Αυτή μπορεί εσωτερικά να έχει την εξάγωνη αρχιτεκτονική που αναφέραμε πριν και να βγάζει διάφορα API προς τις υπόλοιπες εσωτερικές εφαρμογές της υπηρεσίας ή και προς άλλους clients. Αυτές οι διαφορετικές εφαρμογές συνήθως (και συστήνεται) τρέχουν σε διαφορετικά υπολογιστικά συστήματα.

Κάθε μικρο-εφαρμογή (microservices) λειτουργεί αυτόνομα. Έχει ένα API μέσω του οποίου επικοινωνούν οι άλλες εφαρμογές με αυτή και αντίστοιχα και αυτή με τα API άλλων εφαρμογών. Δηλαδή παύει η εφαρμογή να είναι το μικρότερο στοιχείο και αρχίζει να αποτελείται από άλλες μικρότερες.

Κάποιος που πρώτη φορά αντιμετωπίζει microservices μπορεί να σχολιάσει ότι με αυτό τον τρόπο έχουμε duplication πολλών λειτουργιών. Για παράδειγμα, κάθε microservices έχει τη δική του βάση, με το δικό το σχήμα, αντί για μία ενιαία βάση που χρησιμοποιείται για όλα τα δεδομένα. Αυτό παρόλο που μπορεί να παρουσιαστεί σαν overhead και να οδηγήσει και σε κάποιο duplication δεδομένων, είναι απαραίτητο για να μπορέσουν τα microservices να είναι loosely coupled (χαλαρά συνδεδεμένα) μεταξύ τους.

Τι σημαίνει όμως ακριβώς loosely coupled; Κάθε microservice πρέπει να μπορεί να λειτουργεί όσο το δυνατόν πιο αυτόνομα από τα υπόλοιπα. Με αυτό τον τρόπο είναι πολύ εύκολο να αλλάζουν αλλά και να μην βασίζονται το ένα στο άλλο. Σε διαφορετική περίπτωση, όταν δηλαδή τα microservices είναι πολύ στενά συνδεδεμένα, ουσιαστικά έχουμε μία μονολιθική εφαρμογή όπου τα modules/packages έχουν γίνει απλά ολόκληρες εφαρμογές.

1.4 Πλεονεκτήματα των microservices

Το σημαντικότερο πλεονέκτημα της αρχιτεκτονικής με microservices είναι ότι προσφέρει μία λύση στην αυξανόμενη πολυπλοκότητα ενός συστήματος. Χτίζοντας τα microservices αναγκάζεται ο developer να χωρίσει την εφαρμογή σε ξεχωριστά services τα οποία όμως έχουν σαφή όρια. Με αυτό τον τρόπο μπορούμε να έχουμε modules πολύ πιο σαφή και πολύ περισσότερο loosely coupled που μπορούν να αναπτυχθούν ευκολότερα και ταχύτερα.

Ένα άλλο σημαντικό πλεονέκτημα είναι ότι με αυτή την αρχιτεκτονική κάθε microservice μπορεί να αναπτυχθεί σε οποιαδήποτε γλώσσα και χρησιμοποιώντας οποιαδήποτε τεχνολογία. Το μόνο σταθερό είναι το API που πρέπει να υποστηρίζει. Έτσι οι προγραμματιστές μπορούν να επιλέξουν τυχόν διαφορετικές γλώσσες/τεχνολογίες αναλόγως με τα ιδιαίτερα χαρακτηριστικά που έχει κάθε microservice. Επίσης δίνεται η δυνατότητα σε νέα microservices να χρησιμοποιηθούν τεχνολογίες που δεν υπήρχαν όταν δημιουργήθηκαν προηγούμενα. Τέλος καθώς τα microservices είναι σχετικά μικρά, υπάρχει η δυνατότητα ακόμα και για καθολικής ανάπτυξης από την αρχή με κάποια πιο νέα τεχνολογία.

Σε αντίθεση με τη μονολιθική εφαρμογή όπου το deployment ήταν μία επίπονη και πολύπλοκη διαδικασία, πλέον κάθε microservice γίνεται deployed μόνο του. Το καθένα ακολουθεί το δικό του ρυθμό ανάπτυξης και μπορεί να περάσει live σε production με το που τελειώσει το testing. Σταματά δηλαδή το ένα τμήμα της εφαρμογής να δημιουργεί εμπόδια στο άλλο και όλα να περιμένουν κάποιο τρίτο που ήταν κλασικό σενάριο παλιότερα.

Όπως αναφέραμε, ένα ιδιαίτερα σημαντικό πρόβλημα μιας ώριμης μονολιθικής εφαρμογής είναι το scaling. Με τα microservices μπορούμε να κάνουμε scale όποιο microservice χρειάζεται μόνο και στο βαθμό που χρειάζεται επίσης. Είναι σύνηθες όταν ο orchestrator (πχ kubernetes) βλέπει κάποιο microservices ότι φτάνει στα όριά του να κάνει spin up περισσότερα instances. Επίσης μπορούν διαφορετικά microservices να τρέχουν σε διαφορετικό hardware αναλόγως με τις ανάγκες και τις ιδιαιτερότητες του καθένα.

1.5 Μειονεκτήματα των microservices

Το σύστημα που έχει αναπτυχθεί με microservices είναι ένα κατανεμημένο σύστημα. Για αυτό το λόγο ο developer πρέπει να γράψει κώδικα για την επικοινωνία μεταξύ των microservices (συνήθως μέσω μηνυμάτων) αλλά και να κάνει error handling για την περίπτωση που κάποια από τα υπόλοιπα με τα οποία θέλει να επικοινωνήσει δεν είναι διαθέσιμα. Αυτό προσθέτει ένα overhead στον όγκο του προγραμματισμού που πρέπει να κάνει.

Μαζί με το κατανεμημένο σύστημα έρχεται και η κατανεμημένη αρχιτεκτονική στις βάσεις. Αντί μιας καθολικής βάσης στην οποία μπορούμε να ελέγχουμε όλα τα transactions, κάθε microservice έχει τη δική του βάση. Επίσης συνήθως για την εκτέλεση μιας λειτουργίας πρέπει να γίνουν transactions σε πολλές βάσεις. Αυτό πρέπει να μπορέσει να αντιμετωπιστεί σαν ενιαίο transaction με έξτρα κώδικα και πολυπλοκότητα από την μεριά του προγραμματιστή.

Όπως αναφέρθηκε, τα microservices κάνουν το testing ευκολότερο αλλά ταυτόχρονα λίγο πιο πολύπλοκο ως προς το orchestration τους. Επειδή δεν σηκώνεται ολόκληρη η εφαρμογή ενιαία, αλλά μόνο το microservice πρέπει να σηκωθούν και τα αντίστοιχα υπόλοιπα microservices με τα οποία επικοινωνεί (ή να γίνουν stubs)

Σε περιπτώσεις που χρειάζεται να αναπτυχθούν features που θέλουν αλλαγές σε περισσότερα του ενός microservices πρέπει να γίνεται και προγραμματισμός του πώς οι αλλαγές αυτές θα γίνουν rollout ώστε να ικανοποιούνται τα dependencies. Αυτό αφενός δεν είναι συχνό φαινόμενο αν τα microservices είναι σωστά δομημένα όσον αφορά τη λειτουργικότητά τους. Αφετέρου μπορεί να αποφευχθεί με fallbacks στη λειτουργικότητα τους αλλά αυτό είναι επιπλέον κώδικας που πρέπει να αναπτυχθεί.

Το deployment της εφαρμογής ταυτόχρονα με το να γίνει πιο ευέλικτο και δυνατό έγινε και περισσότερο πολύπλοκο. Σε μία κλασική μονολιθική εφαρμογή έχουμε απλά έναν server. Σε περίπτωση που έχουμε και scaling έχουμε απλά πανομοιότυπους servers. Στην περίπτωση των microservices

όμως έχουμε πολλές διαφορετικές εφαρμογές σε διαφορετικά VMs/dockers σε διαφορετικά HWs. Όλο αυτό το σύστημα είναι σίγουρα πιο ευέλικτο αλλά πολύ πιο δύσκολο να οργανωθεί.

2 Υλοποίηση Εφαρμογής

Έχουμε ήδη περιγράψει τους λόγους που οδήγησαν στην εμφάνιση αυτής της νέας αρχιτεκτονικής υλοποίησης, τα θετικά της αλλά και τα αρνητικά της. Έχοντας αυτά στο νου καταλαβαίνουμε ότι ο developer καλείται να αλλάξει πολλά πράγματα στον τρόπου που δουλεύει με microservices στους παρακάτω τομείς:

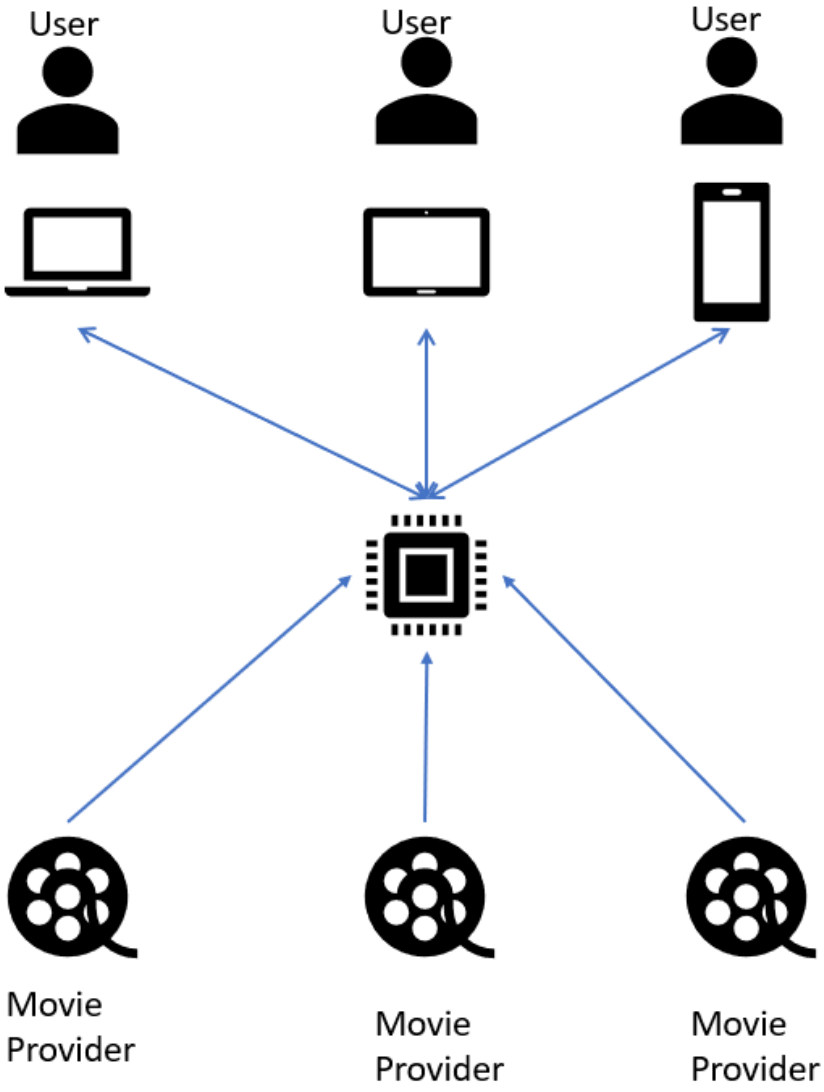
- σχεδιασμός (design)
- υλοποίηση (implementation)
- επίλυση προβλημάτων (debugging)
- προώθηση στην παραγωγή (deployment)

Όπως έχουμε ήδη αναφέρει σκοπός της εργασίας αυτής είναι να εμβαθύνει ακριβώς σε αυτούς τους τομείς, να δείξει τις αλλαγές που πρέπει να κάνει ο προγραμματιστής, τα προβλήματα που θα αντιμετωπίσει αλλά και να προτείνει λύσεις για καθένα από αυτά.

3 Περιγραφή

Μία από τις πρώτες εταιρίες μεγάλου μεγέθους που έκαναν πολύ δημοφιλή τα microservices ήταν το Netflix. Θεώρησε ότι αυτή η αρχιτεκτονική ταίριαζε στο distributed χαρακτήρα της και βοήθησε στο να την προωθήσει με αρκετές open source βιβλιοθήκες που βοηθούν στο microservice προγραμματισμό.

Σαν φόρο τιμής, αντίστοιχα και η δική μας εφαρμογή (**Rentflix**) υλοποιεί ένα απλουστευμένο σύστημα ενοικίασης ταινιών μέσω διαδικτύου. Το σενάριο που υλοποιεί είναι ότι υπάρχουν πολλοί "προμηθευτές" ταινιών (movie providers) στο σύστημα και μία βασική εφαρμογή που είναι υπεύθυνη για την ενοικίαση των ταινιών αυτών στους καταναλωτές.



Στην εφαρμογή υλοποιούμε το backend τμήμα, κάνοντας expose τα κατάλληλα APIs ώστε να μπορούν να χρησιμοποιηθούν από οποιοδήποτε client.

4 Τεχνολογίες που χρησιμοποιήθηκαν

4.1 Java

4.1.1 Λίγα λόγια

Η Java είναι μια γλώσσα προγραμματισμού που προέρχεται από τα εργαστήρια της Sun Microsystems και πατέρας της είναι ο James Gosling. Παρουσιάστηκε το 1995 σαν το κύριο κομμάτι της Java Platform της Sun Microsystems. Οι αρχικοί java compilers, virtual machines και βιβλιοθήκες αναπτύχθηκαν από τη Sun, η οποία όμως το 2007 τους κυκλοφόρησε υπό την άδεια GNU GPL. Σαν γλώσσα κληρονομεί πολλά στοιχεία από το συντακτικό της C και της C++ αλλά σε πολλά σημεία μένει σε υψηλότερο επίπεδο αρχιτεκτονικά από αυτές. Οι εφαρμογές που είναι γραμμένες σε java τυπικά γίνονται compile σε bytecode που στη συνέχεια μπορεί να τρέξει (θεωρητικά) σε κάθε java virtual machine (JVM) ασχέτως αρχιτεκτονικής. Είναι δηλαδή μια γλώσσα γενικού σκοπού, concurrent (δηλαδή τα προγραμμάτά της μπορούν να σχεδιαστούν να και να υλοποιηθούν σαν τμήματα που μπορούν να εκτελεστούν παράλληλα), αντικειμενοστραφής και class-based (δηλαδή η κληρονομικότητα βασίζεται σε κλάσεις αντικειμένων και όχι στα ίδια τα αντικείμενα). Έχει σχεδιαστεί ειδικά για να έχει όσο το δυνατόν λιγότερες εξαρτήσεις όσον αφορά την υλοποίηση. Σκοπός της είναι «να γράφεται μία φορά και να τρέχει παντού», δηλαδή ο ίδιος κώδικας να τρέχει σε διαφορετικές πλατφόρμες .

Επιγραμματικά αναφέρουμε κάποια βασικά πλεονεκτήματά της:

- platform independent (write once, run everywhere)
- simple (ο προγραμματιστής δεν χρειάζεται να ασχοληθεί με pointers)
- object oriented (inheritance, encapsulation, polymorphism, dynamic binding)
- robust (διαχείριση μνήμης, garbage collection, exception handling, type checking)
- secure (όλα τα προγράμματα τρέχουν μέσα σε μιας μορφής "sandbox")
- multithreaded

4.1.2 Java & microservices

Όπως αναφέραμε παραπάνω, ένα από τα βασικά πλεονεκτήματα των microservices είναι ότι απελευθερώνουν τον χρήστη από την ανάγκη να επιλέξει μία και μόνο γλώσσα ανάπτυξης λογισμικού.

Παρόλα αυτά σχεδόν το σύνολο του "οικοσυστήματος" microservices που αναπτύξαμε έγινε με τη χρήση java. Οι λόγοι ήταν συγκεκριμένοι και πολύ χρήσιμο το να αναφερθούν.

1. Εμπειρία του προγραμματιστή: Ένας προγραμματιστής αναπτύσσει γρηγορότερα σε μια γλώσσα που γνωρίζει καλύτερα. Επιπλέον, όταν χρειάζεται να γίνει ανάπτυξη με καινούργιες τεχνολογίες, αρχιτεκτονικές ή frameworks καλό είναι να επιλέγεται κάποιο από αυτά σαν γνωστό και να μην είναι όλα καινούργια
2. Συντακτικό με annotations. Ένα πολύ χρήσιμο χαρακτηριστικό της java είναι τα annotations που κάνουν πολύ γρήγορη την ανάπτυξη microservies ειδικά όταν χρησιμοποιούνται με κάποιο framework όπως το spring boot. Ο κώδικας είναι πιο καθαρός, διαβάζεται και επεκτείνεται πολύ ευκολότερα

3. Η JVM σαν πλατφόρμα δίνει στον προγραμματιστή την ευκαιρία να γράψει τμήματα και σε διαφορετικές γλώσσες. Για παράδειγμα μπορεί να χρησιμοποιήσει groovy (μέσω gradle) για το χτίσιμο ή spock για τα τεστ.

Σίγουρα όμως ο πρωταρχικός λόγος που η java επιλέχτηκε ήταν το spring boot framework, το οποίο είναι τόσο σημαντικό που αξίζει να αναπτυχθεί περεταίρω.

4.2 Spring Boot Framework

Το Spring framework χρησιμοποιεί νέες τεχνικές όπως Aspect-Oriented programming (AOP), Plain Old Java Object (POJO), και dependency injection για την ανάπτυξη enterprice εφαρμογών, αφαιρώντας έτσι τις δυσκολίες και τις πολυπλοκότητες που έχουν άλλες αντίστοιχες τεχνολογίες (όπως τα enterprice java beans EJB). Το Spring είναι ένα ελαφρύ σχετικά framework ανοιχτού κώδικα που δίνει στους προγραμματιστές τη δυνατότητα να αναπτύξουν απλές, αξιόπιστες και scalable enterprise εφαρμογές. Κυρίως επικεντρώνεται στο να προσφέρει διάφορους τρόπους διαχείρισης των διαφορετικών business object. Με αυτές τις τεχνικές έκανε ευκολότερη την ανάπτυξη διαδικτυακών εφαρμογών από τα παραδοσιακά frameworks όπως JDBC, JSP, Java Servlets. Το Spring framework μπορεί να θεωρηθεί μία ομπρέλα κάτω από την οποία υπάρχουν διάφορα άλλα μικρότερα frameworks που μπορούν να λειτουργήσουν συνδυαστικά. Αυτά τα υπο-frameworks ονομάζονται layers και ενδεικτικά είναι το Spring Web MVC, Spring Data, Spring AOP, Spring ORM, Spring Web Flow κ.ά.

Κάποια από τα βασικότερα χαρακτηριστικά του είναι τα εξής:

- **Inversion of Control Container:** Είναι το βασικό container στο οποίο με τη χρήση inversion of control / dependency injection για να μπορέσει να παρέχει ένα reference ενός object σε μια κλάση σε runtime χρόνο.
- **Data access framework:** Δίνει τη δυνατότητα χρήσης APIs όπως το JDBC / Hibernate για την αποθήκευση δεδομένων λύνοντας προβλήματα όπως database interaction, exception handling, transaction handling κ.ά.
- **Spring MVC framework:** Επιτρέπει την ανάπτυξη διαδικτυακών εφαρμογών με την αρχιτεκτονική MVC
- **Transaction Management:** Βοηθά στην διαχείριση transaction τόσο global (μέσω ενός application server) αλλά και τοπικών (μέσω JDBC Hibernate κτλ)
- **JDBC Abstraction Layer**: βοηθά στην αντιμετώπιση λαθών με εύκολο τρόπο στην αλληλεπίδραση με JDBC
- **Spring TestContext**: βοηθά στο unit testing και integration testing των spring εφαρμογών.

Δυστυχώς όμως χρειαζόταν (αρχικά τουλάχιστον) πολύ και πολύπλοκο configuration μέσω xml αρχείων και trial and error.

Η λύση σε αυτό ήρθε από την δημιουργό εταιρία Pivotal με το Spring Boot. Το Spring Boot είναι ένα framework βασισμένο σε microservices που δίνει τη δυνατότητα να αναπτυχθούν production-ready εφαρμογές σε πολύ λίγο χρόνο. Βασίζεται στην έννοια του convention over configuration έχοντας προ-παραμετροποιημένες πολλές λειτουργίες και επιλογές σύμφωνα με τη γνώμη των δημιουργών του. Επίσης έρχεται με configuration (πάλι opinionated) για πολλά 3rd party libraries που συχνά καλείται να κάνει integrate ο προγραμματιστής.

4.2.1 Χαρακτηριστικά

Κάποια από τα βασικότερα χαρακτηριστικά του Spring Boot framework είναι τα εξής:

- Αποφεύγει το configuration μέσω XML και αντίθετα, το κάνει μέσω java κώδικα.
- Προσφέρει εύκολη δημιουργία REST Endpoint
- Περιλαμβάνει έναν embedded tomcat server
- Προσφέρει εύκολο deployment (είτε jar είτε war)
- Microservices Αρχιτεκτονική

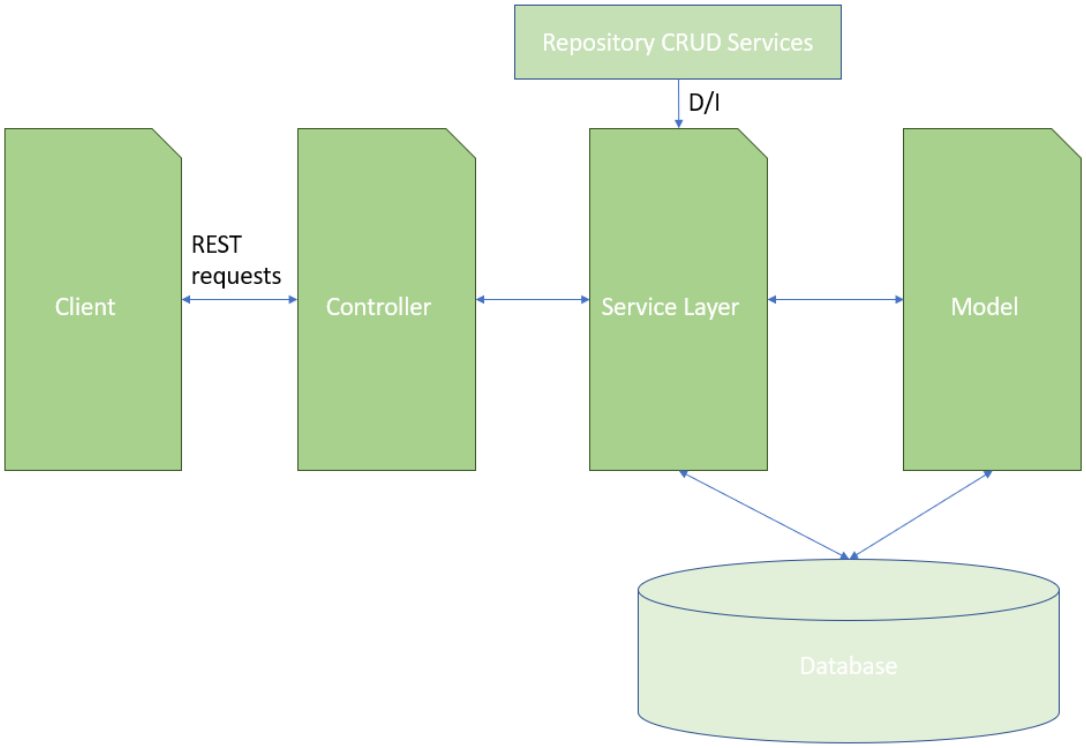
4.2.2 Αρχιτεκτονική

Στο Spring Boot υπάρχουν διαφορετικά επίπεδα που συνεισφέρουν με διαφορετικές λειτουργίες:

- **Presentation Layer:** περιλαμβάνει τα views (frontent)
- **Data Access Layer:** προσφέρει CRUD (create, retrieve, update, delete) λειτουργίες σε μια βάση
- **Service Layer:** περιλαμβάνει τα service classes και χρησιμοποιεί services από το data access layer
- **Integration Layer:** περιλαμβάνει διάφορα web services (μέσω διαδικτύου)

Σε αυτά έρχονται να προστεθούν utility classes, validator classes και view classes.

Καθώς το Spring Boot χρησιμοποιεί όλα τα features και τα modules του Spring (spring data, spring mvc κτλ) η αρχιτεκτονική του είναι σχεδόν ίδια με αυτή του spring MVC εκτός του ότι δεν χρειάζονται DAO και DAOimpl κλάσεις. Η δημιουργία ενός data access layer χρειάζεται απλά μία repository κλάση που παρέχει ένα crud σύνολο λειτουργιών.



1. Ο client κάνει HTTP REST calls

2. Ο controller που αντιστοιχεί στο κατάλληλο mapping λαμβάνει το call, το επεξεργάζεται και καλεί διάφορα services αν χρειάζεται επιπλέον service logic.
3. Το κύριο κομμάτι του business logic βρίσκεται στο service layer που μπορεί να το εφαρμόζει σε δεδομένα από τη βάση όπως αυτά παρέχονται από το JPA με entity κλάσεις
4. Η σελίδα επιστρέφεται στον client εάν δεν έχει συμβεί κάποιο λάθος στη διαδικασία.

4.3 Version Control

Σε μεγάλα projects όπου οι αλλαγές στον κώδικα είναι συνεχείς και οι προσθήκες στην code base γίνονται από πολλούς προγραμματιστές είναι μεγάλη η δυσκολία της καταγραφής των διάφορων αλλαγών αλλά και του «ποιος άλλαξε τι». Καθώς οι ομάδες σχεδιάζουν και αναπτύσσουν το λογισμικό, είναι αναπόφευκτο πολλαπλές εκδοχές (versions) του ίδιου αρχείου να βρίσκεται στους υπολογιστές διαφορετικών developer. Ακόμα διάφορα features η ακόμα και bugs μπορεί να υπάρχουν μόνο σε συγκεκριμένες versions.

Για αυτό το λόγο αναπτύχθηκαν διάφορα συστήματα επονομαζόμενα **revision control systems, version control systems** ή απλά **source control**. Σε αυτά οι διάφορες αλλαγές στον κώδικα ταυτοποιούνται από κάποιο μοναδικό κωδικό, ενώ καταχωρείται η ώρα και ο χρήστης που έκανε την αλλαγή. Οι διάφορες αλλαγές μπορούν να αποθηκευτούν, να συγκριθούν μεταξύ τους και κάποιες φορές να συνεννοηθούν.

Ο χρήστης ενός VCS θα πρέπει να έχει κατανοήσει τους παρακάτω όρους:

- **Repository**: Το μέρος όπου τα αρχεία και τα διάφορα απαραίτητα metadata για τη λειτουργία του VCS αποθηκεύονται. Το repository μπορεί να είναι είτε τοπικό είτε κεντρικό και σε πολλά VCS, συνυπάρχουν και τα δύο είδη.
- **Revision** (version): Κάθε αλλαγή που βρίσκεται αποθηκευμένη στο repository. Branch: Ένα σύνολο αρχείων που έχουν διακλαδωθεί σε κάποιο σημείο στο παρελθόν ώστε δύο διαφορετικά versions του ίδιο αρχείου να μπορούν να αναπτυχθούν και να αλλαχθούν σε διαφορετικούς ρυθμούς και ανεξάρτητα μεταξύ τους.
- **Checkout**: Η δημιουργία ενός τοπικού αντίγραφου του repository ώστε να μπορεί ο developer να εργαστεί πάνω του. Κατά το checkout παίρνουμε είτε την τελευταία revision του repo ή κάποια συγκεκριμένη παλαιότερη που επιθυμούμε.
- **Commit**: Η εγγραφή των διαφόρων αλλαγών στο repository.
- **Merge**: Η συνένωση δύο διαφορετικών branch, επιλύοντας τυχόν διαφορές (conflicts) στον κώδικα ώστε να καταλήξουμε στο επιθυμητό αποτέλεσμα.

Το workflow που ακολουθείται και το οποίο είναι άσχετο με τον τρόπο υλοποίησης και του είδους του VCS είναι το εξής: Ο developer κάνει checkout, δηλαδή δημιουργεί ένα αντίγραφο ενός συγκεκριμένου revision του repository που θέλει να αλλάξει (συνήθως το τελευταίο). Επιφέρει τις αλλαγές που επιθυμεί και αφού τις συγκεντρώσει τις κάνει commit είτε στο τελευταίο revision, είτε δημιουργώντας ένα νέο branch.

4.3.1 Git

4.3.1.1 Λίγα λόγια

Το Git είναι ένα DVCS (distributed version control system) με πρωταρχικό στόχο την ταχύτητα. Ο αρχικός σχεδιασμός και η ανάπτυξη έγινε από τον Linus Torvalds για να χρησιμοποιηθεί για revision control στο Linux Kernel, αλλά από τότε χρησιμοποιείται ευρέως σε πληθώρα open source project, αλλά και σε εταιρικό επίπεδο. Διανέμεται υπό την άδεια ανοιχτού λογισμικού GNU General Public License V2.

4.3.1.2 Χαρακτηριστικά

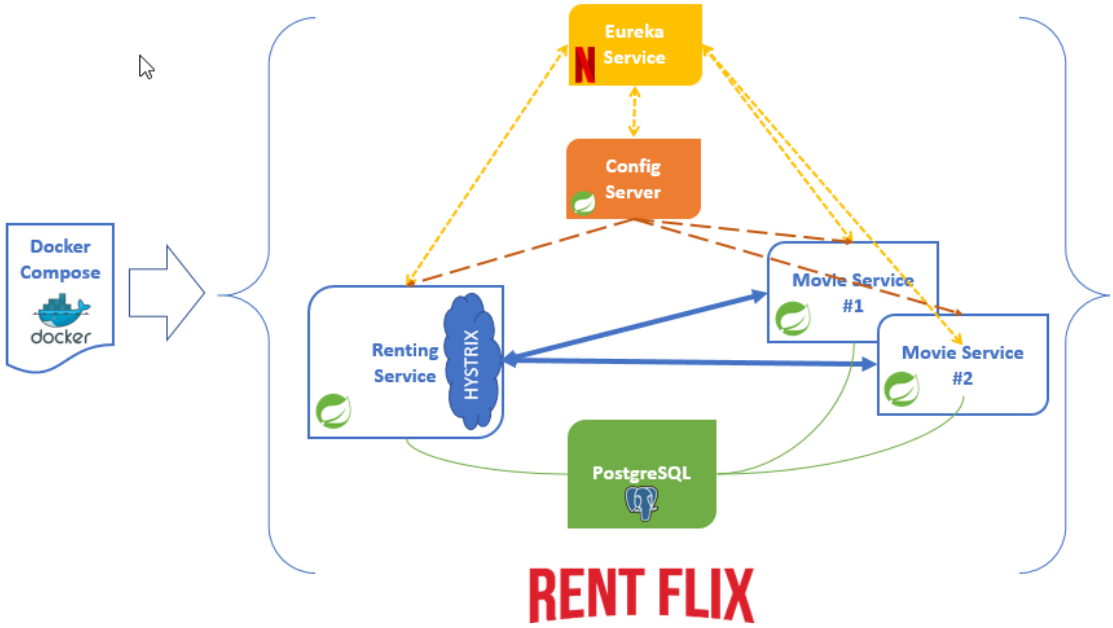
Το Git, υποστηρίζει πολύ γρήγορη δημιουργία branch αλλά και merge περιλαμβάνοντας τα κατάλληλα εργαλεία ώστε να μπορεί ο developer να εργαστεί αλλά και να περιηγηθεί στον κώδικα μη-γραμμικά. Βασικό αξίωμα κατά τον σχεδιασμό του git ήταν ότι μία αλλαγή γίνεται merge περισσότερες φορές από όσες γράφεται και συνεπώς η δημιουργία branch και είναι πολύ γρήγορη αλλά και το ίδιο το branch πολύ ελαφρύ καθώς αποτελεί απλά μία αναφορά σε ένα commit. Όπως προείπαμε, το git είναι DVCS, distributed, δηλαδή διανεμημένο. Αυτό σημαίνει ότι κάθε developer κατέχει ένα πλήρες τοπικό αντίγραφο του repository με ολόκληρη την ιστορία όλων των αλλαγών σε όλα τα branches. Οι αλλαγές που επιτελεί στον κώδικα αποθηκεύονται πρώτα στο τοπικό repository (local repository) και μετά εάν το επιθυμεί προωθούνται στο απομακρυσμένο. Επειδή δημιουργήθηκε για να χρησιμοποιηθεί από το linux kernel, είναι ιδιαίτερα αποτελεσματικό στα μεγάλα projects. Η ύπαρξη του local repository το κάνει να είναι σε άλλη τάξη μεγέθους από άλλα vcs. Δεν γίνεται δηλαδή αργότερο όσο το code base και το project μεγαλώνουν. Το data model που χρησιμοποιεί το git εξασφαλίζει την κρυπτογραφική ακεραιότητα (cryptographic integrity) κάθε bit του project. Για κάθε αρχείο και commit που γίνεται, υπολογίζεται το checksum του, και με αυτό ακριβώς το checksum ανακτώνται από το repository. Γι αυτό το λόγο είναι αδύνατο ο χρήστης να πάρει κάτι διαφορετικό από αυτό ακριβώς που έβαλε. Και γι αυτό είναι αδύνατο να γίνει οποιαδήποτε αλλαγή χωρίς να αλλάξουν τα IDs των πάντων από το σημείο αυτό και μετά. Οπότε εάν ο χρήστης κατέχει το ID ενός συγκεκριμένου commit, εξασφαλίζει όχι μόνο ότι το project είναι ακριβώς το ίδιο με όταν έγινε το commit, αλλά και ότι τίποτα στο παρελθόν δεν είχε αλλάξει.

4.3.1.3 Workflow

Ένα ακόμα χαρακτηριστικό του Git που οφείλεται στην ιστορία του ως ενός καθαρά linux development tool είναι η πολυμορφικότητά του. Δεν αποτελεί ένα μονολιθικό εργαλείο, αλλά ένα σετ εντολών που δίνουν σε διαφορετικούς χρήστες να δημιουργήσουν το δικό τους τρόπο εργασίας. Έτσι δεν υπάρχει ένα μοναδικό workflow, αλλά διάφορα τα οποία όμως κινούνται με παραλλαγές γύρω από τα εξής βήματα:

- Ενημέρωση του local repository
  - git pull (τραβάει όλες τις αλλαγές από το remote repository)
- Δημιουργία ενός branch για την ανάπτυξη ενός συγκεκριμένου feature
  - git checkout –b branch-name (δημιουργεί ένα νέο τοπικό branch)
  - Ανάπτυξη του feature
  - git add (προσθέτει τα νέα τυχόν αρχεία που δημιουργήθηκαν)
  - git status & git diff (ενημέρωση για τις αλλαγές που έχουν γίνει)
  - git commit –m “message” (αποθήκευση των αλλαγών σαν commit)
- Μετάβαση στο πρωταρχικό branch
  - git checkout master
- Merging του feature-branch με το master-branch
  - git merge branch-name
- Ενημέρωση του remote repository
  - git push

5 Αρχιτεκτονική



Το παραπάνω σχήμα δείχνει την αρχιτεκτονική της εφαρμογής μας. Έχουμε δημιουργήσει ένα docker compose αρχείο το οποίο όταν το τρέχουμε μεσω του docker κανει spin up όλη τη διασυνδεδεμένη εφαρμογή μας. Τα microservices που την απαρτίζουν επιγραμματικά είναι τα εξής:

1. **Eureka Service**: Υλοποίηση του cloud eureka server που βοηθά στο Service Registration και το Serice Discovery.
2. **Config Server:** Υλοποίηση του Spring Cloud Config. Προσφέρει server και client side support για externalized configuration.'
3. **Renting Service**. Spring boot microservice, υλοποιεί το business logic του τμήματος ενοικίασης των ταινιών
4. **Movie Service**. Spring boot microservice, υλοποιεί έναν φανταστικό movie provider και υπάρχει σε περισσότερα του ενός container ώστε να δείξει την παρουσίας πολλών διαφορετικών
5. **PostgreSQL**. Το container με τη βάση στην οποία γράφουν τα renting service & movie service containers.

Η επιλογή των microservices αυτών έγινε ώστε να έχουμε μία μικρογραφία ενός distributed microservices ecosystem όπου κάθε εφαρμογή μπορεί να υπάρξει σε περισσότερα από ένα container χωρίς να δημιουργεί πρόβλημα αλλά και, το κυριότερο, να μην είναι ορατό από κάποιον τρίτο.

Δηλαδή όταν κάποιος αλληλοεπιδρά με το σύστημα δεν έχει γνώση ούτε αντιλαμβάνεται αν είναι μία μονολιθική εφαρμογή ή αν αποτελείται από περισσότερα microservices και πόσα είναι αυτά.

Η επιτυχία της σχεδίασης μιας εφαρμογής με microservices είναι όταν κάποιος τρίτος παρατηρητής δεν μπορεί να αντιληφθεί την αρχιτεκτονική αυτή, αλλά το μόνο που αντιλαμβάνεται είναι το availbalility και η σταθερότητά της. Όπως , για παράδειγμα, όταν κάποιος κάνει stream μία ταινία από το netflix δεν γνωρίζει σε ποιον server ήταν η ταινία, σε ποιον server έγιναν render τα UI. Ή αντίστοιχα όταν κάποιος μπαίνει στο facebook δεν γνωρίζει σε ποιον container τρέχει το κομμάτι του messaging και σε ποιο το κομμάτι των notiftication. Το μόνο που αντιλαμβάνεται είναι η συνολική, πάντα available, εφαρμογή.

Ακολούθως θα αναλύσουμε ένα προς ένα τα επιμέρους microservices και θα δούμε λεπτομέρειες της υλοποίησής τους.

6 Παρουσίαση Microservices

6.1 Eureka Server

6.1.1 Service Discovery

Όπως έχουμε περιγράψει στην εφαρμογή μας, όπως και σε κάθε υλοποίηση με microservices υπάρχει ένα οικοσύστημα εφαρμογών οι οποίες επικοινωνούν μεταξύ τους. Πώς όμως γνωρίζουν "που" είναι η κάθε μία για να επικοινωνήσουν ή, ακόμα πιο σημαντικό, πώς γνωρίζουν "ποιές" αλλά και "πόσες" είναι; Πώς δηλαδή θα λύσουμε το πρόβλημα του **Service Discovery**;

Σε μία κλασική μονολιθική υλοποίηση κάθε τμήμα της εφαρμογής είναι μοναδικό και εσωτερικό. Τυχόν εξωτερικές εφαρμογές είναι σε συγκεκριμένες διευθύνσεις και συνήθως μοναδικές.

Σε ένα σύστημα microservices, ειδικά όταν αυτές υλοποιούνται μέσα σε containers, κάθε ένα microservices μπορεί να υπάρχει σε περισσότερα του ενός instances και σε διευθύνσεις που δεν είναι σταθερές. Άρα η εύκολη λύση που μπορεί κάποιος να σκεφτόταν για την επικοινωνία μεταξύ τους δηλαδή το να γράψει hardcoded διευθύνσεις είναι αδύνατη. Πόσο μάλλον όταν έχουμε πολλά instances και θέλουμε να κάνουμε load balancing μεταξύ τους.

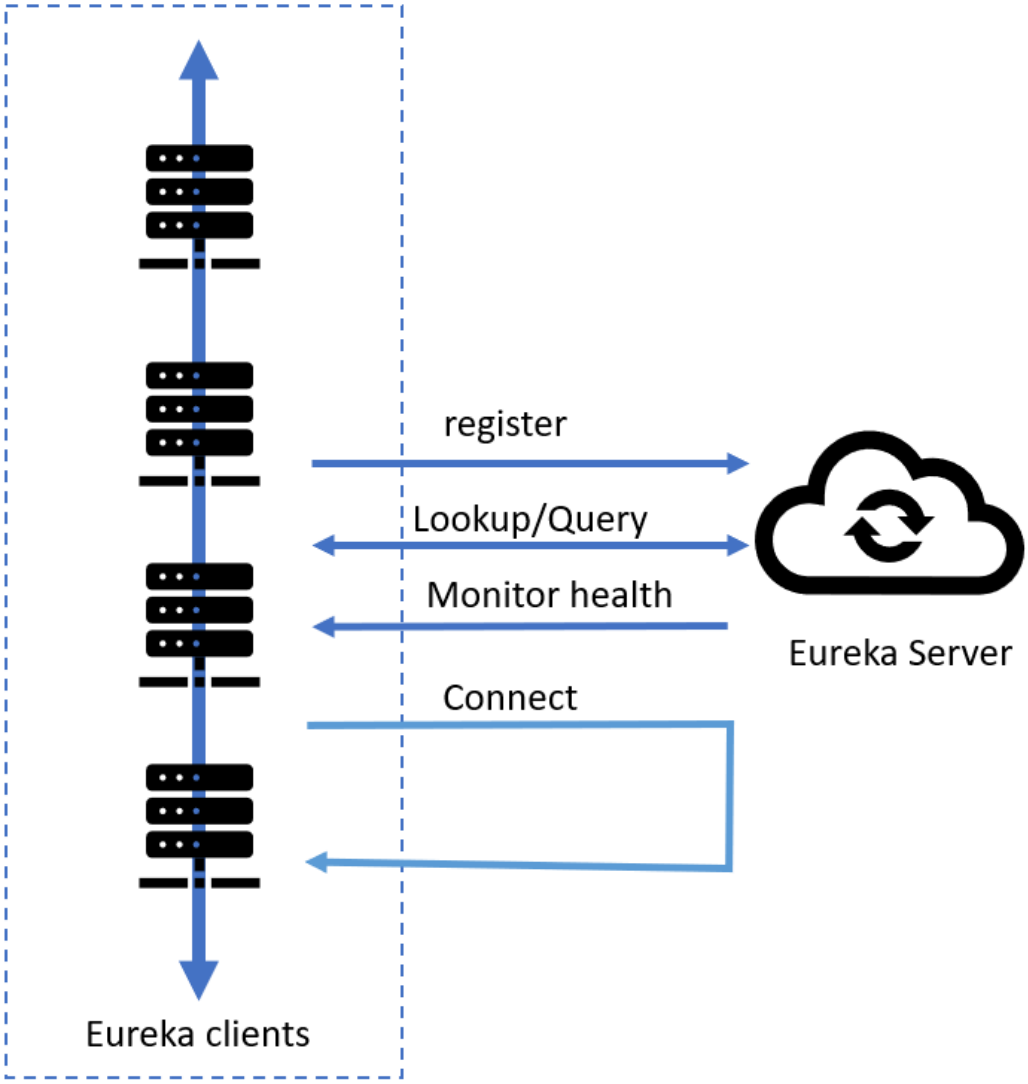
Χρειαζόμαστε μία λύση που θα έχει τα παρακάτω χαρακτηριστικά:

1. High availability: Σε περιπτώσεις πολλών instances πρέπει επικοινωνίες προς ένα προβληματικό instance να μεταδίδονται σε άλλα υγιή
2. load balancing: σε περιπτώσεις υψηλού φόρτου αυτό πρέπει να μοιράζεται μεταξύ των instances
3. resiliency: πρέπει να υποστηρίζει local caching
4. fault tolerance: όταν κάποιο instance δεν είναι available πρέπει να αφαιρείται από τη λίστα των available

6.1.2 Netflix Eureka

Μία τέτοια υλοποίηση είναι το Netflix Eureka. Μία open source εφαρμογή ανεπτυγμένη από το Netflix για να λύσουν in house προβλήματα service discovery. Συγκεκριμένα λύνει τα εξής θέματα:

1. service registration: επιτρέπει σε services να "εγγραφούν" σε ένα κατάλογο με διαθέσιμα services σε ένα κεντρικό server
2. client lookup of servive address: ο client μπορεί να αναζητήσει τη διεύθυνση για κάποιο συγκεκριμένο service από αυτό τον κατάλογο
3. information sharing: διαμοιράζεται την πληροφορία για τα διάφορα services μεταξύ των nodes
4. health monitoring: επιβλέπει τα διάφορα services και αν κάποιο από αυτά δεν είναι "υγειές" το αφαιρεί από τον κατάλογο.



Για να δουλέψει το Netflix Eureka χρειάζεται ένας server και ένας αριθμός από clients. Στην εφαρμογή μας ο server είναι το microservice "**eurekaserver**". Για να ενεργοποιήσουμε το eureka server πρέπει αρχικά να τον κάνουμε import μέσω του mvn:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
</dependencies>
```

Και να δημιουργήσουμε μία Spring Boot Εφαρμογή με το @EnableEurekaServer annotation:

```java
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

Πρέπει επίσης να κάνουμε και το απαραίτητο configuration. Αυτό γίνεται μέσω του application.yml αρχείου:

```yaml
server:
  port: 8761

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
  serviceUrl:
    defaultZone: http://localhost:8761
```

Αναλυτικά έχουμε κάνει τις εξής ρυθμίσεις:

- server.port: η πόρτα στην οποία θα λειτουργεί το microservice μας
- eureka.client.registerWithEureka: false προφανώς αφού είναι ο server δεν θέλουμε να κάνει register στον εαυτό του.
- serviceUrl.defaultZone: Έχει να κάνει με το zoning που χρησιμοποιεί το eureka αλλά είναι μια λειτουργία που δεν θα τη χρησιμοποιήσουμε.

Με αυτές τις επιλογές έχουμε ένα eureka server που τρέχει. Όμως πρέπει κάπως οι clients να γνωρίζουν ότι υπάρχει και να επικοινωνούν μαζί του. Για να γίνει αυτό πρέπει σε κάθε client να ενεργοποιήσουμε τον eureka client με το annotation @EnableEurekaClient:

```java
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
public class Application {
    public static void main(String[] args) { SpringApplication.run(Application.class, args); }
}
```

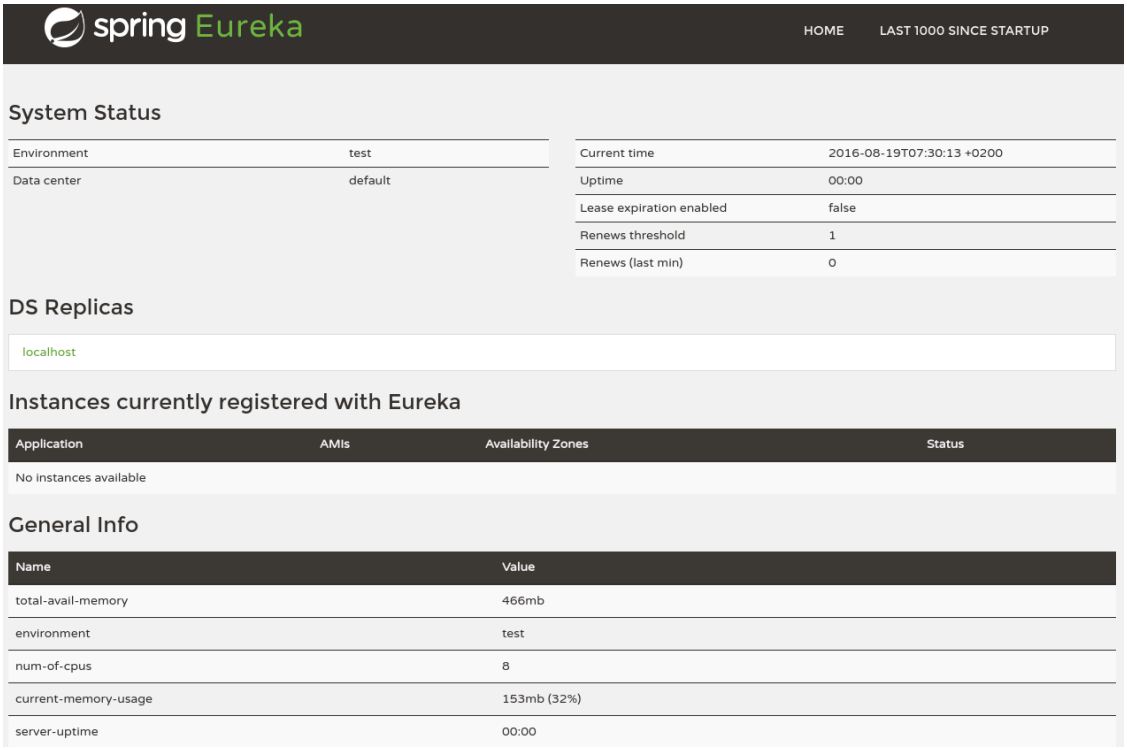
Και να κάνουμε τις αντίστοιχες ρυθμίσεις στο application.yml:

```yaml
spring:
    application:
        name: ONOMA_MICROSERVICE
eureka:
    instance:
```

```
    preferIpAddress: true
client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
        defaultZone: http://localhost:8761/eureka/
```

Με το eureka.client.registerWithEureka: true ενεργοποιούμε το service registration με το όνομα spring.application.name. Επίσης επιλέγουμε local caching του καταλόγου των services με το eureka.client.fetchRegistry:true και τέλος του λέμε σε πιο συγκεκριμένο zone θέλουμε να κάνουμε register με το eureka.client.serviceUrl.defaultZone

Το eureka server όταν τρέχει σηκώνει και ένα web ui στο οποίο μπορεί ο χρήστης να δει τα διάφορα services που έχουν γίνει register:



Με την υλοποίηση και τις ρυθμίσεις που έχουμε κάνει ως τώρα, έχουμε ένα eureka server στον οποίο οι διάφοροι clients γίνονται register. Αφού όμως γίνονται register, πώς κάνουν lookup τα υπόλοιπα services? Αυτό θα το δούμε αναλυτικότερα στην παρουσίαση της κάθε εφαρμογής.

6.2 Cloud Config

6.2.1 Διαχωρισμός κώδικα και παραμετροποίησης

Κάθε εφαρμογή ανεξαρτήτως μεγέθους έχει διάφορες παραμέτρους που μπορούν να αλλάξουν. Όσο η εφαρμογή μεγαλώνει και γίνεται περισσότερο πολύπλοκη τόσο αυτές οι παράμετροι γίνονται περισσότερες και η διαχείρισή τους ένα θέμα που ο προγραμματιστής δεν μπορεί να αποφύγει.

Λύση 1η: hard-coded τιμές

Αρχικά σε μία απλή εφαρμογή, ο προγραμματιστής κάνει όλες τις επιλογές για τις τιμές αυτών των παραμέτρων. Συχνά δεν αντιλαμβάνεται καν ότι το κάνει καθώς δεν περνά από το μυαλό του ότι θα μπορούσε κάποιος χρήστης σε κάποιο συγκεκριμένο σενάριο να θελήσει διαφορετικές τιμές. Για αυτό το λόγο καταλήγει στην πιο απλή λύση: τιμές hard-coded μέσα στον κώδικα. Θα μπορούσε κανείς να πει ότι εφόσον δεν δίνει δυνατότητα παραμετροποίησης δεν μπορούμε να μιλάμε για configuration. Αλλά το γεγονός ότι υπάρχουν αυτές οι τιμές μέσα στον κώδικα αποτελούν hard-coded configuration.

Λύση 2η: configuration files

Καθώς η πολυπλοκότητα της εφαρμογής αυξάνεται, αυτές οι διάσπαρτες τιμές αρχίζουν να γίνονται όλο και πιο δύσκολο να οργανωθούν και να διαχειριστούν οπότε αρχίζουν να συγκεντρώνονται σε ξεχωριστές classes που περιέχουν μόνο configuration values. Αυτό είναι βολικό ώστε να μπορεί να αλλάξει ο προγραμματιστής πριν το compilation κάποια τιμή, αλλά δε δίνουν τη δυνατότητα για **runtime configuration**. Το επόμενο βήμα είναι να χρησιμοποιήσει κάποιο εξωτερικό **configuration file**. Αυτό μπορεί να είναι κάποιο **XML** αρχείο ή κάποιο **YAML** αρχείο. Η ιδέα είναι ότι αυτό το αρχείο θα περιέχει όλες τις τιμές που θα μπορεί κάποιος να παραμετροποιήσει για την εφαρμογή αυτή και μάλιστα θα φορτώνεται στην έναρξη της εφαρμογής ώστε να μην χρειάζεται ξεχωριστό compilation κάθε φορά.

Ξαφνικά όμως ο προγραμματιστής αντιλαμβάνεται ότι αρχίζει να γράφει κώδικα καθαρά για να διαχειριστεί το configuration της εφαρμογής του. Αυτό σημαίνει ότι η εφαρμογή του γίνεται πολυπλοκότερη χωρίς αντίστοιχα να προστίθενται έξτρα features. Και όπως έχουμε αναφέρει, αυτό είναι ένα κλασικό πρόβλημα μιας μονολιθικής εφαρμογής.

Λύση 3η: πλήρης διαχωρισμός implementation από configuration.

Αν χρησιμοποιήσουμε το **Spring framework** μας δίνεται αυτή ακριβώς η δυνατότητα του configuration file χωρίς να χρειαστούμε να κάνουμε τίποτα. Το framework κατά το bootstrap της εφαρμογής αναζητά σε συγκεκριμένο μέρος ένα configuration file και είναι επίσης υπεύθυνο για να μεταφέρει αυτές τις τιμές σε placeholders που έχουμε θέσει εμείς στον κώδικα.

6.2.2 Configuration και Microservices

Με αυτό τον τρόπο έχουμε αποφύγει τον επιπλέον κώδικα/υλοποίηση για να διαχειριστούμε το configuration της εφαρμογής μας. Όμως, όπως ήταν αναμενόμενο, σε μία εφαρμογή που χρησιμοποιεί microservices υπάρχουν ιδιαίτερα προβλήματα όσον αφορά το configuration:

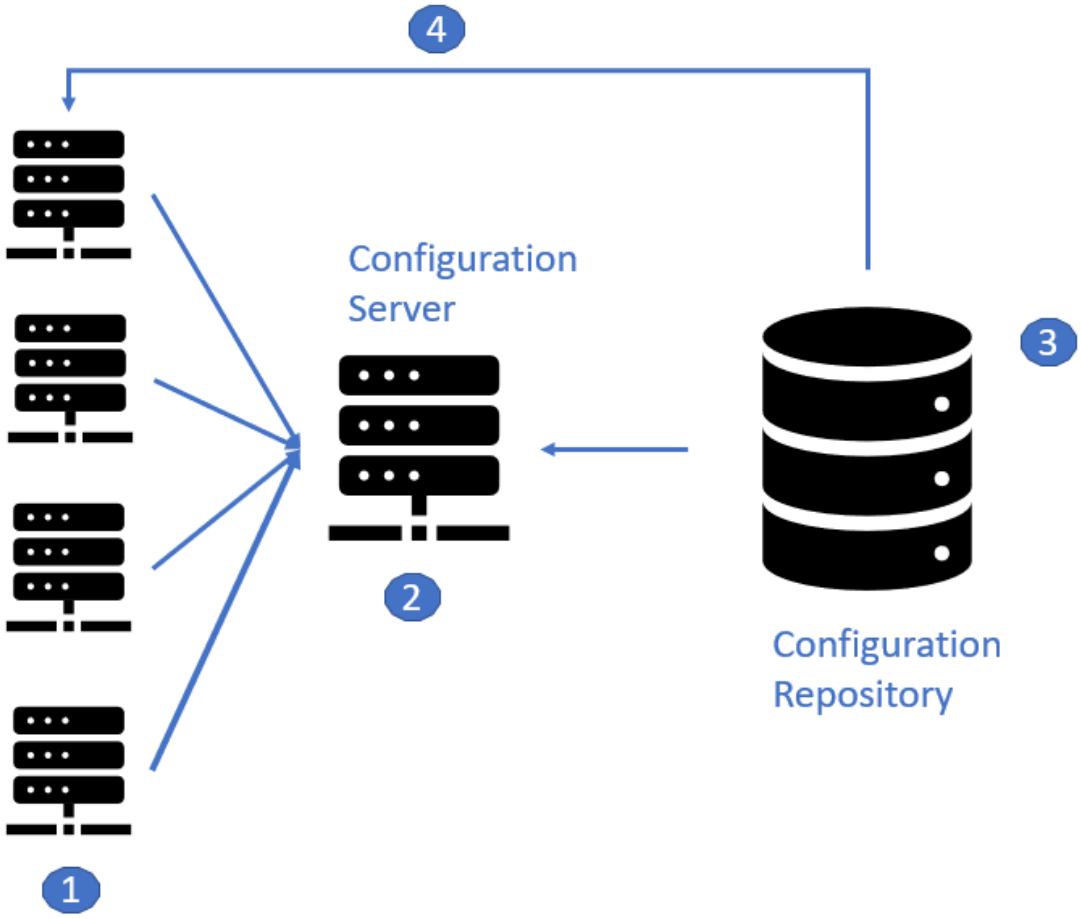
1. Πάρα πολλές παράμετροι (configuration parameters) για πολλές εφαρμογές
2. πολλές παράμετροι είναι κοινές αλλά άλλες διαφοροποιούνται
3. πολλά διαφορετικά instances της ίδιας εφαρμογής
4. Όταν πρέπει να αλλάξει κάποια παράμετρος πρέπει να "ενημερωθούν" (update) όλα τα instances του microservice αυτού

Άρα είναι κατανοητό ότι χρειάζεται μία επιπλέον λύση που θα έχει τα παρακάτω χαρακτηριστικά:

- **segregation**: το configuration θα διαχωρίζεται απόλυτα από το implementation και δε θα γίνεται καν deployed μαζί με την εφαρμογή
- **abstraction**: η πρόσβαση στο configuration δε θα γίνεται άμεσα αλλά μέσω ενός interface (πχ REST)
- **centralization**: όλο το configuration όλων των εφαρμογών θα γίνεται από ένα κεντρικό μέρος
- **availability**: από τη στιγμή που όλα τα services θα βασίζονται σε ένα για το onfiguration τους, αυτό πρέπει να είναι πάντα διαθέσιμο.

6.2.2.1 Spring Cloud Configuration Server

Μία τέτοια υλοποίηση είναι ο Spring Cloud Configuration Server. Μια server - client λύση για το πρόβλημα που μόλις περιγράψαμε. Στην εφαρμογή μας είναι το microservices **cloud config.** Η λειτουργία του μπορεί να περιγραφεί ως εξής:



1. Η εφαρμογή κατά το bootstrap κάνει access to congfiguration server
2. Ο configuration server φέρνει το configuration από ένα εξωτερικό configuration repository
3. Το configuration αποθηκεύεται σε εξωτερικό repository για availability & change tracking.
4. Όταν κάποια τιμή αλλάξει στο repository, ενημερώνονται τα κατάλληλα applications ώστε να ενημερωθούν και να φορτώσουν πάλι το configuration.

Υλοποίηση στον Server

Η υλοποίηση του Spring Cloud Config έγινε ως εξής:

Αρχικά πρέπει να κάνουμε import τα απαραίτητα maven dependencies:

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
  </dependency>
```

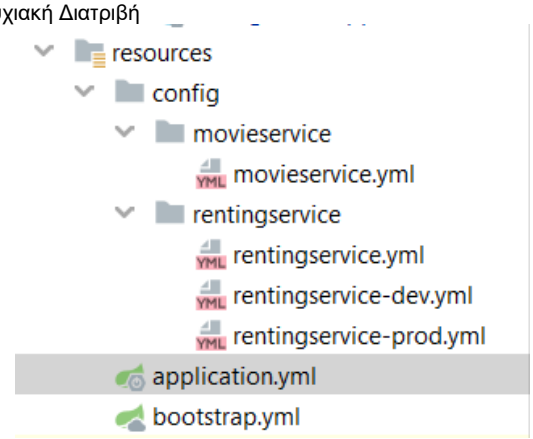
Για να "ενεργοποιήσουμε' τη λειτουργία του server πρέπει στη main class μας να προσθέσουμε το Annotation @EnableCloudConfig

```java
@SpringBootApplication
@EnableEurekaClient
@EnableConfigServer
public class ConfigServerApplication {

    public static void main(String[] args) {

        SpringApplication.run(ConfigServerApplication.class, args);
    }

}
```

Με αυτό τον το microservices αυτό λειτουργεί σαν Configuration Server, αλλά δεν του έχουμε κάνει τις απαραίτητες ρυθμίσεις για να λειτουργεί όπως εμείς θέλουμε. Ο Spring Configuration Server μπορεί να "σερβίρει" configuration αποθηκευμένο με δύο τρόπους:

1. Σε local φάκελο
2. Σε εξωτερικό git repository.

Για να χρησιμοποιήσουμε τον 1ο τρόπο πρέπει στο φάκελο resources να αποθηκεύσουμε τα κατάλληλα .yml αρχεία που θέλουμε για κάθε service. Σε περίπτωση που θέλουμε διαφορετικο configuration για διαφορετικά profile, χρησιμοποιούμε την ονοματολογία service-profile.yml. Για παράδειγμα στην εφαρμογή μας έχουμε δημιουργήσει τα αντίστοιχα αρχεία:

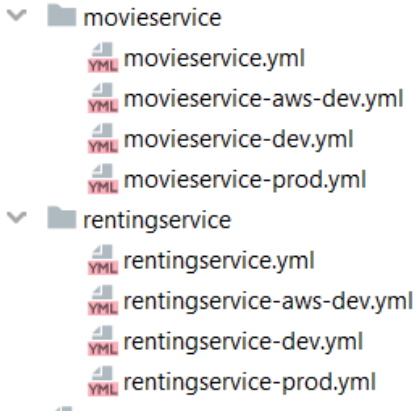


Για να χρησιμοποιήσουμε εξωτερικό Git repository πρέπει να κάνουμε στο application.yml αρχείο του Cloud Config server μας τις εξής ρυθμίσεις:

```yaml
spring:
  cloud:
    config:
      discovery:
        enabled: true
      server:
        encrypt.enabled: false
        git:
          uri: https://github.com/spirosag/configuration-repo/
          searchPaths: rentingservice,movieservice
          username: native-cloud-apps
          password: 0ffended
```

1. spring.cloud.config.discovery.enabled: Ενεργοποιούμε τη χρήση DiscoveryClient για τον αρχικό εντοπισμό του server
2. spring.cloud.config.server.git.uri Το uri του εξωτερικού git repository
3. spring.cloud.config.server.git.searchPaths Τα paths στα οποία θα ψάξει για configuration σε αυτό το repository
4. spring.cloud.config.server.encrypt.enabled αν θέλουμε encryption
5. spring.cloud.config.server.git.username , spring.cloud.config.server.git.password το username και password αν έχουμε encryption

Στην υλοποίησή μας έχουμε υλοποιήσει και τους δύο τρόπους για λόγους παραδείγματος. Ενεργοποιημένο έχουμε μόνο τον τρόπο με το git καθώς είναι ο ενδεδειγμένος για production. Το git repository είναι ένα github repository με τα εξής αρχεία:



Περιλαμβάνει δηλαδή τα αντίστοιχα αρχεία που θα βάζαμε μέσα στα resources με τον 1ο τρόπο.

Υλοποίηση στον Client

Αντίστοιχα, για να μπορέσουμε να συνδέσουμε κάποιο service σαν client στον configuration server πρέπει να κάνουμε τα παρακάτω:

Αρχικά να φέρουμε το maven dependency:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

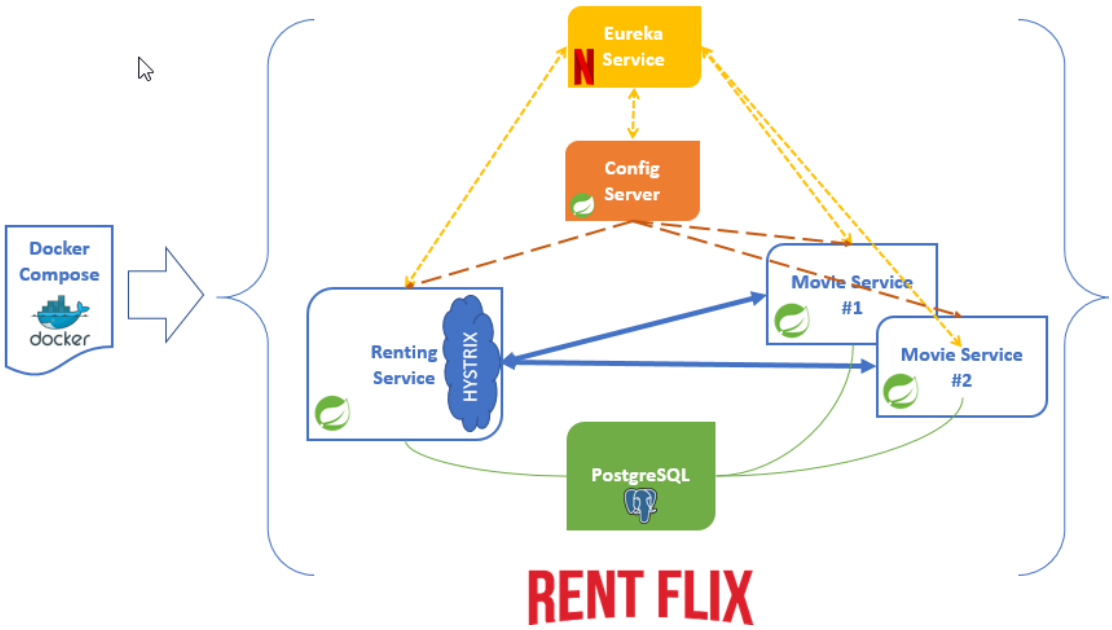
Αναφέραμε ότι το configuration το φέρνει από τον server κατά το bootstrap. Για να το ενεργοποιήσουμε στο bootstrap.yml κάνουμε την εξής ρύθμιση:

```
cloud:
  config:
    enabled: true
```

Σε περίπτωση που δεν χρησιμοποιούσαμε eureka server για να "ανακαλύψουμε" το server θα έπρεπε να κάνουμε έξτρα ρυθμίσεις για το πού βρίσκεται ο server μας. Αλλά βλέπουμε ότι οι επιλογές που έχουμε κάνει έχουν αρχίσει να μας κάνουν ήδη τη ζωή ευκολότερη.

6.3 Movie Service

Όπως αναλύσαμε και στην αρχιτεκτονική, η εφαρμογή μας λειτουργεί ως εξής:



Έχουμε πολλά instances από **Movie Services**. Αυτά αναπαριστούν τους διάφορους παρόχους ενοικίασης ταινιών. Αυτό θα μπορούσε να είναι μια εταιρία παραγωγής ή κάποιο τηλεοπτικό κανάλι. Αυτά παρέχουν τις ταινίες προς ενοικίαση οι οποίες αποθηκεύονται στη βάση και διαχειρίζονται από το **Renting Service**

Αναλυτικά οι classes του Movie Service και η λειτουργικότητά τους έχει ως εξής:

6.3.1 **Application** class

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
public class Application {
    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);
    }
}
```

- @SpringBootApplication : ενεργοποιεί το spring boot σαν spring boot application
- @EnableEurekaClient: λέμε στο service να χρησιμοποιήσει το Eureka Server για να επικοινωνήσει με τα υπόλοιπα services
- @EnableCircuitBreaker Ενεργοποιεί ένα συγκεκριμένο design pattern για την επικοινωνία με άλλα services που ονομάζεται circuit breaker. Λόγω της σημασίας του θα αναλυθεί διεξοδικά παρακάτω

6.3.2 Movie Service Controller

```java
@RestController
@RequestMapping(value = "v1/movies")
public class MovieServiceController {
    @Autowired
    private MovieService movieService;

    @RequestMapping(value = "/{movieId}", method = RequestMethod.GET)
    public Movie getMovie(@PathVariable("movieId") String movieId) { return movieService.getMovie(movieId); }

    @RequestMapping(value = "/{movieId}", method = RequestMethod.PUT)
    public void updateMovie(@PathVariable("movieId") String movieId, @RequestBody Movie movie) {
        movieService.updateMovie(movie);
    }

    @RequestMapping(value = "/{movieId}", method = RequestMethod.POST)
    public void saveMovie(@RequestBody Movie org) { movieService.saveMovie(org); }

    @RequestMapping(value = "/{movieId}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteMovie(@PathVariable("movieId") String movieId, @RequestBody Movie org) {
        movieService.deleteMovie(org);
    }
}
```

Εδώ δημιουργούμε ένα **controller** δηλαδή ένα Componet το οποίο ακούει σε HTTP commands σε ένα συγκεκριμένο path. Η λειτουργικότητα που θέλουμε να του δώσουμε είναι ένα CRUD API για ταινίες. δηλαδή κάποιος (σε αυτή την περίπτωση το Renting Service) μπορεί να κάνει τις εξής λειτουργίες:

Path	HTTP COMMAND	Λειτουργία
/v1/movies/{movieId}	GET	**επιστρέφει την ταινία που αντιστοιχεί στον κωδικό αυτόν**
/v1/movies/{movieId}	PUT	**κάνει update τις πληροφορίες υπάρχουσας ταινίας που αντιστοιχεί στον κωδικό αυτόν**
/v1/movies/{movieId}	POST	**σώζει μία ταινία σε αυτό τον κωδικό**
/v1/movies/{movieId}	**DELETE**	**διαγράφει την ταινία που αντιστοιχεί στον κωδικό αυτόν**

Τις λειτουργίες αυτές τις κάνει μέσω του MovieService . Ο λόγος που δεν τις κάνει απευθείας ο contoller είναι για να έχουμε **separation of concerns**: Ο controller είναι αποκλειστικά η "πόρτα" προς τον έξω κόσμο και το MovieService το "μυαλό". Το MovieService γίνεται ΄@Autowired, δηλαδή το dependncy injection του Spring το δημιουργεί και το κάνει inject μέσα στην class μας. Αντίστοιχα επειδή το @MovieServiceController είναι @RestController αυτό σημαίνει ότι είναι Spring Component και άρα και αυτό θα το δημιουργήσει το Spring μόνο του. Δεν χρειάζεται να καλέσουμε κάποιο constructor του.

6.3.3 Movie service

```
@Service
public class MovieService {
    @Autowired
    private MovieRepository movieRepository;

    public Movie getMovie(String movieId) { return movieRepository.findById(movieId); }

    public void saveMovie(Movie movie){
        movie.setId( UUID.randomUUID().toString());

        movieRepository.save(movie);

    }

    public void updateMovie(Movie movie) { movieRepository.save(movie); }

    public void deleteMovie(Movie movie) { movieRepository.delete( movie.getId()); }
}
```

Όπως είπαμε, τις CRUD ενέργειες για τις ταινίες, δηλαδή:

1. **Create**
2. **Read**
3. **Update**
4. **Delete**

τις κάνουμε μέσω του Movie Service.

Σε αυτό παρατηρούμε τα εξής:

- Είναι και αυτό ένα Spring @Component που σημαίνει ότι θα το δημιουργήσει το Spring με το Dependency Injection μηχανισμό του.
- Περιέχει ένα @Autowired MovieRepository.
- όλες οι μέθοδοι getMovie(), updateMovie(), saveMovie(), deleteMovie() απλά μεταβιβάζουν calls σε αντίστοιχες μεθόδους του MovieRepository

6.3.4 Movie Repository

```
@Repository
public interface MovieRepository extends CrudRepository<Movie,String>  {
    Movie findById(String movieId);
}
```

Το MovieRepository περιλαμβάνει όλη την υλοποίηση για την επαφή με την βάση μας. Όμως με μια πρώτη ματιά φαίνεται σαν να μην περιέχει καθόλου υλοποίηση. Εδώ φαίνεται η μαγεία και η δύναμη του Spring καθώς μέσα σε 3 γραμμές έχουμε πάρει ένα τεράστιο functinality:

- @Repository: υποδηλώνει ότι το interface αυτό είναι repository, και ενεργοποιεί το μηχανισμό του Spring JPA
- extends CrudRepository<Movie,String>: κάνουμε extend το interface CrudRepository για μία βάση όπου θα αποθηκεύουμε Movie με id ένα String

- Movie findById(String movieId);δίνουμε μία έξτρα μέθοδο στο inteface μας πέρα από αυτά που δηλώνονται στο CrudRepository

Κάποιος θα αναρωτηθεί ότι έχουμε κάνει extend ένα interface και δεν έχουμε πουθενά το πραγματικό implementation. Αυτό το κομμάτι έρχεται να μας το δώσει "δωρεάν" το Spring. Καθώς ο dependency injection μηχανισμός του θα βρεί ένα υπάρχον component που θα είναι implementation του CrudRepository και θα το κάνει inject στη θέση του Movie Repository.

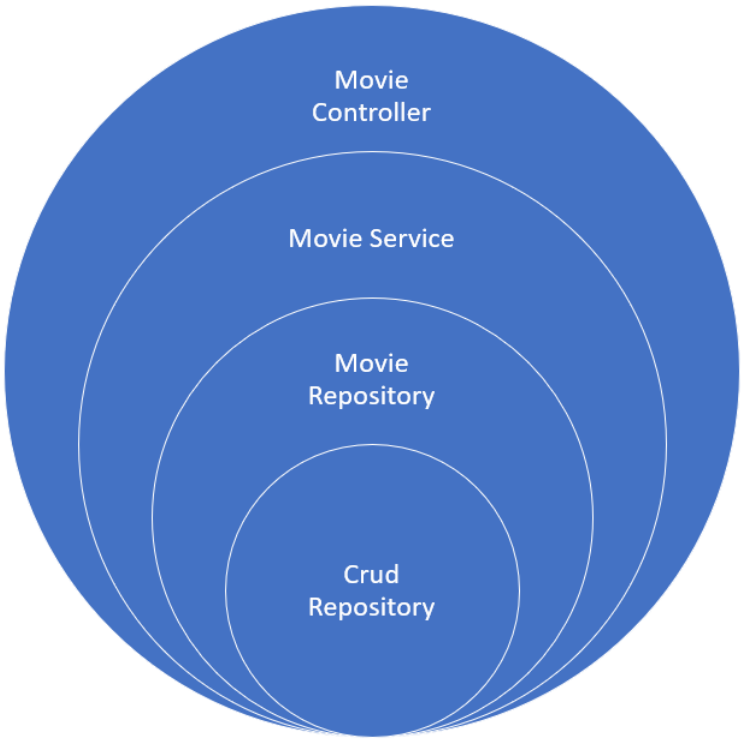
Στο σημείο αυτό πολύ σωστά κάποιος θα αναρωτηθεί: "Για τις μεθόδους που έχει ήδη declared το CrudRepository μπορεί να έχει ήδη implementation το spring. Εμείς εδώ όμως προσθέσαμε μία νέα μέθοδο κάνοντας extend το CrudRepository. Πού θα βρεθεί το implementation αυτό;"

Τη λύση σε αυτό το ερώτημα έρχεται να τη δώσει πάλι το Spring, καθώς το component που θα κάνει autowire στη θέση του MovieRepository όχι μόνο έχει implementation για τις ήδη declared μεθόδους του CrudRepository αλλά μπορεί "βλέποντας" το όνομα μιας νέας μεθόδου να δημιουργεί το αντίστοιχο query που πρέπει να κάνει στη βάση! Στη δική μας περίπτωση θα αναζητήσει ένα movie με id ένα συγκεκριμένο string.

6.3.5 Separation of concerns

Αναφέραμε αρχικά ότι ο MovieController δεν κάνει το business logic, αλλά ότι αυτό γίνεται στο MovieService. Στη συνέχεια είδαμε ότι τα method calls απλά μεταβιβάζονται στο MovieRepository. Τί ακριβώς συμβαίνει και πού έχουμε κάνει λάθος;

Η απάντηση είναι πουθενά. Αυτό είναι ένα κλασικό design pattern όταν γράφουμε μία spring εφαρμογή. Θέλουμε να διατηρούμε ένα καθαρό separation of concerns: Σε κάθε layer έχουμε ένα συγκεκριμένο functionality που υλοποιούμε και εμπεριέχει άλλα components χωρίς να ενδιαφέρεται για το δικό τους implementation:



1. Στο layer του Movie Controller υλοποιούμε functionality που έχει να κάνει με το web (http calls, return codes etc). Αυτό περνά με τη σειρά του τα calls στο Movie Service χωρίς να ξέρει τι ακριβώς κάνει

2. Στο layer τουMovie Service υλοποιούμε το business logic σχετικά με το API μας. Για παράδειγμα αν έπρεπε να κάνουμε έξτρα calls σε άλλα APIs για να πάρουμε πληροφορία θα το κάναμε εδώ. Σε αυτή την περίπτωση το business είναι ξεκάθαρο. Με τη σειρά του περνάει τα calls στο Movie Repository
3. Στο Movie Repository υλοποιούμε την επαφή με τη βάση. Είμαστε τυχεροί γιατί με το Spring το μόνο που χρειάζεται να κάνουμε είναι να κάνουμε extend το CrudRepository και έχουμε έτοιμο το implementation.

Με αυτό τον τρόπο πετυχαίνουμε αφενός κάθε επίπεδο να γνωρίζει μόνο ότι είναι απαραίτητο για τη λειτουργικότητά του και αφετέρου να μπορούμε να αλλάξουμε κάποια υλοποίηση χωρίς να χρειάζεται να αλλάξουμε όλη την υλοποίηση.

Για παράδειγμα αν αποφασίσουμε να αλλάξουμε βάση, τότε θα αλλάξουμε απλά την υλοποίηση στο MovieRepository και Movie Service και Movie Controller θα μείνουν ανέγγιχτα. Αντίστοιχα, αν αλλάξουμε το business logic, θα αλλάξουμε μόνο το Movie Service, ή αν αλλάξουμε κάτι στην διεπαφή με τον έξω κόσμο θα αλλάξουμε μόνο κάτι στο Movie Controller

6.3.6 Movie

Αναφέραμε ότι στο Movie Repository αποθηκεύουμε Movie. Η κλάση Movie υποδηλώνει την ταινία και έχει τα παρακάτω attributes:

```java
@Entity
@Table(name = "movies")
public class Movie {
    @Id
    @Column(name = "movie_id", nullable = false)
    String id;

    @Column(name = "title", nullable = false)
    String title;

    @Column(name = "storyline", nullable = false)
    String storyline;

    @Column(name = "director", nullable = false)
    String director;

    @Column(name = "plot", nullable = false)
    String plot;
```

Εδώ πρέπει να επεξηγήσουνε τα εξής:

- @Entity: υποδηλώνουμε ότι η κλάση αυτή θα είναι ένα entity κατά το javax persistence. Θα είναι δηλαδή κάτι που θα το αποθηκεύσουμε σε μια βάση
- @Table(name = "movies"): ορίζουμε ότι θα αποθηκευτεί σε ένα table με το όνομα "movies". Στην περίπτωσή μας θα είναι ένα table στην postgres βάση μας.
- @Id : υποδηλώνει ότι το String id θα είναι το key στο table μας

- @Column(name = "movie_id", nullable = false): Ορίζουμε το όνομα του collumn καθώς και το αν θα μπορεί να είναι άδεια, δηλαδή με null τιμή.

  Αντίστοιχα έχουμε και την υλοποίηση των getters & setters αλλά δεν χρήζουν επεξήγησης:

```
public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getTitle() { return title; }

public void setTitle(String title) { this.title = title; }

public String getStoryline() { return storyline; }

public void setStoryline(String storyline) { this.storyline = storyline; }

public String getDirector() { return director; }

public void setDirector(String director) { this.director = director; }

public String getPlot() { return plot; }

public void setPlot(String plot) { this.plot = plot; }
```

6.3.7 Resource files

Ένα σημαντικό χαρακτηριστικό του Spring είναι το "**Convention over configuration**". Με αυτό εννοούμε ότι αντί να γράφουμε configuration για το οτιδήποτε, υπάρχουν "προ-αποφασισμένα" conventions για το που θα γίνεται τι και πώς. Ένα από αυτά τα conventions είναι τα αρχεία που περιέχουν μεταβλητές για configuration. Αυτά θέτονται στο φάκελο resources και για το συγκεκριμένο service είναι τα εξής:

6.3.7.1 bootstrap.yml

```
spring:
  application:
    name: movieservice
  profiles:
    active:
      default
  cloud:
    config:
      enabled: true
```

Αυτό το αρχείο φορτώνεται κατά το bootstrap του service, δηλαδή κατά την πρώτη - πρώτη αρχικοποίηση και έναρξη. Εδώ έχουμε κάνει τις εξής ρυθμίσεις:

- spring.application.name: το όνομα του service
- spring.profiles.active: με ποιο spring profile θα ξεκινάει εάν δεν του έχουμε ορίσει κάποιο εμείς
- spring.cloud.config.enabled: ενεργοποιούμε τη σύνδεση με το spring cloud config server μας

6.3.7.2 application.yml

```yaml
eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
        defaultZone: http://localhost:8761/eureka/

 #Setting the logging levels for the service
logging:
  level:
    com.netflix: WARN
    org.springframework.web: WARN
    com.microcorp: DEBUG
```

Περιλαμβάνει τις βασικές ρυθμίσεις:

- eureka.client.registerWithEureka: ενεργοποιούμε το registration του service στο eureka server μας
- eureka.client.fetchRegistry : ενεργοποιούμε το local caching του eureka registry
- eureka.client.serviceUrl.defaultZone: επιλέγουμε σε ποιο συγκεκριμένο zone θα κάνει register στο eureka (εάν είχαμε περισσότερα)
- eureka.instance.preferIpAddress: προτιμούμε IPs από dns names
- logging.level: επιλέγουμε το επίπεδο logging που θέλουμε για τα διάφορα components

Σημείωση: Σε αυτές τις ρυθμίσεις θα έρθουν να προστεθούν και αυτές που θα πάρει από τον cloud config server.

6.3.7.3 schema.sql

```
CREATE TABLE movies (
  movie_id        VARCHAR(100) PRIMARY KEY NOT NULL,
  title                 TEXT NOT NULL,
  storyline           TEXT NOT NULL,
  director          TEXT NOT NULL,
  plot          TEXT    NOT NULL);


INSERT INTO movies (movie_id,  title, storyline, director, plot)
VALUES ('e254f8c-c442-4ebe-a82a-e2fc1d1ff78a', 'Batman Begins', 'After training with his mentor, Batman

INSERT INTO movies (movie_id, title, storyline, director, plot)
VALUES ('442adb6e-fa58-47f3-9ca2-ed1fecdfe86c', 'The Godfather', 'The aging patriarch of an organized cr
```

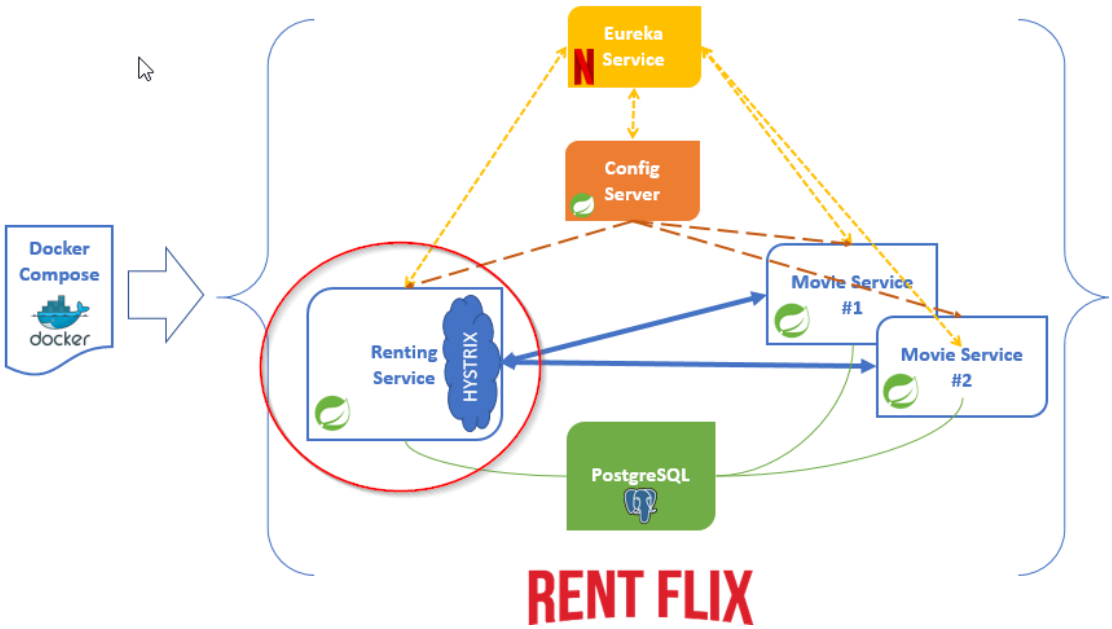
Όπως είπαμε έχουμε convention over configuration με το Spring. Έτσι, για να αρχικοποιήσουμε τη βάση μας δε χρειάζεται να γράψουμε επιπλέον κώδικα. Το μόνο που χρειάζεται είναι να βάλουμε τις sql εντολές που θέλουμε να τρέξουμε σε ένα schema.sql αρχείο μέσα στο φάκελο resources και το Spring θα αναλάβει να τις τρέξει για εμάς κατά το initialization.

Εμείς εδώ επιλέγουμε να :

1. κάνουμε DROP το table της βάσης μας με τις ταινίες
2. να αποθηκεύσουμε κάποια dummy δεδομένα.

6.4 Renting Service

Το **renting-service** είναι το microservice που υλοποιεί την λειτουργία της ενοικιάσεως των ταινιών. Ο χρήστης (φυσικός άνθρωπος) μπορεί να επιλέγει μία ταινία που παρέχεται από κάποιο Movie Service και να το κατοχυρώνει ως δανεισμένη σε αυτόν. Εφόσον η υλοποίησή μας είναι για backend, το Renting Service δίνει το απαραίτητο API. Όπως βλέπουμε από το σχήμα της αρχιτεκτονικής έχει τα παρακάτω χαρακτηριστικά:



- συνδέεται με το config-server για να πάρει configuration

- συνδέεται με το eureka-server για να μπορέσει να ενημερωθεί για τα υπόλοιπα services
- επικοινωνεί με τα διάφορα instances του movie-service
- γράφει τα δεδομένα του στην PostgreSQL βάση.

Οι κλάσεις του renting service είναι οι εξής:

6.4.1 Application Class

```
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
public class Application {
    @LoadBalanced
    @Bean
    public RestTemplate restTemplate() { return new RestTemplate(); }

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

Αντίστοιχα με το application.class του Movie Service, έχουμε την εξής annotations

- @SpringBootApplication: ενεργοποιούμε το Spring Boot υποδηλώνοντας ότι αυτή είναι μια Spring Boot εφαρμογή
- @EnableEurekaClient: ενεργοποιούμε τη λειτουργικότητα του eureka client. Αυτό σημαίνει ότι για να συνδεθεί με τα υπόλοιπα services θα γίνει μέσω του eureka και όχι κατευθείαν με hard-coded IP
- @EnableCircuitBreaker: Ενεργοποιεί ένα συγκεκριμένο design pattern για επικοινωνία με άλλα services το οποίο και θα αναλύσουμε παρακάτω.

Επίσης δημιουργούμε ένα RestTemplate Spring bean που θα το χρησιμοποιήσουμε για επικοινωνία με άλλα services. Το ότι είναι Spring bean σημαίνει ότι είναι ένα singleton που γίνεται managed από το spring framework και autowired όπου χρειάζεται.

6.4.2 Movie Rest Template Client

```
@Component
public class MovieRestTemplateClient {
    @Autowired
    RestTemplate restTemplate;

    public Movie getMovie(String movieId){
        ResponseEntity<Movie> restExchange =
                restTemplate.exchange(
                        url: "http://movieservice/v1/movies/{movieId}",
                        HttpMethod.GET,
                        requestEntity: null, Movie.class, movieId);

        return restExchange.getBody();
    }
}
```

Ο Movie RestTemplate Client είναι ένα @Component (δηλαδή Spring bean) που χρησιμοποιεί RestTemplate για να "φέρει" ταινίες. Ας δούμε τι ακριβώς σημαίνει αυτό:

Αρχικά έχουμε @Autowired ένα RestTemplate. Δηλαδή ζητάμε από το spring να μας "εισάγει" (inject) ένα object RestTemplate. Πού θα το βρει όμως; Θα είναι το RestTemplate bean που δημιουργήσαμε στο application.class. Το RestTemplate είναι μία utility class που εκτελεί HTTP commands, φέρνει το αποτέλεσμα και το μετατρέπει σε συγκεκριμένα objects.

Συγκεκριμένα στην μέθοδο Movie getMovie(String movieId) δίνουμε το id μιας ταινίας και μας επιστρέφεται πίσω η ταινία. Αυτή η ταινία βρίσκεται σε κάποιο movie service όμως! Οπότε για να την φέρουμε κάνουμε το εξής:

Εκτελούμε την μέθοδο restTemplate.exchange() και κάνουμε HttpMethod.GET στη διεύθυνση:

http://movieservice/v1/movies/{movieId}

Το RestTemplate γνωρίζει να κάνει το call αλλά και να μετατρέψει την απάντηση σε Movie object. Πώς όμως ξέρει να βρει τη διεύθυνση όταν δεν περιέχει IP αλλά μόνο το όνομα movieservice; Εδώ έρχεται να βοηθήσει το eureka server. Από τη στιγμή που έχουμε ενεργοποιήσει το @EurekaClient θα μετατρέψει το http://movieservice/ στην κατάλληλη διεύθυνση ώστε να μπορέσει να επικοινωνήσει το RestTemplate. Πρέπει να σημειώσουμε ότι αυτό δε συμβαίνει μόνο σε συνεργασία με το RestTemokate, αλλά ότι για την εφαρμογή μας οπουδήποτε χρησιμοποιήσουμε αντίστοιχες διευθύνσεις θα γίνουν resolve από το registry του Eureka Server.

6.4.3 Service Config

```
@Component
public class ServiceConfig{

  @Value("${example.property}")
  private String exampleProperty="";

  public String getExampleProperty(){
    return exampleProperty;
  }
}
```

Η συγκεκριμένη κλάση μας βοηθά να επιδείξουμε μία επιπλέον λειτουργία του Spring και ένα καλό coding practice.

Όπως έχουμε αναφέρει μπορούμε να έχουμε configuration values μέσα σε διάφορα αρχεία όπως application.yml, application.properties κτλ που το Spring υποστηρίζει. Πώς όμως θα εισάγουμε αυτές τις τιμές στον κώδικά μας; Αυτό το πετυχαίνουμε με το annotation @Value.

Στη συγκεκριμένη περίπτωση το Spring θα ψάξει για την τιμή example.property και θα την κάνει inject στη μεταβλητή exampleProperty. Αυτή η λειτουργία υπάρχει σε όλα τα spring beans όμως αν αρχίσουμε να κάνουμε inject @Value μεταβλητές παντού τότε αρχίζει το anti-pattern ότι ο κώδικάς μας "μολύνεται" με configuration. Επίσης, αν χρησιμοποιούσαμε την ίδια τιμή example.property σε πάρα πολλά σημεία και στο μέλλον θέλαμε να βάλουμε και κάποιο business logic πίσω από αυτή θα ήταν δύσκολο.

Για αυτό καλό είναι να χρησιμοποιούμε κάποια configuration class στην οποία συγκεντρώνουμε τις τιμές αυτές και μετά τις κάνουμε expose στην υπόλοιπη εφαρμογή μέσω μεθόδων όπως η getExampleProperty().

6.4.4 Movie

```
public class Movie {
    String id;
    String title;
    String storyline;
    String director;
    String plot;


    public String getId() { return id; }

    public void setId(String id) { this.id = id; }

    public String getTitle() { return title; }

    public void setTitle(String title) { this.title = title; }

    public String getStoryline() { return storyline; }

    public void setStoryline(String storyline) { this.storyline = storyline; }

    public String getDirector() { return director; }

    public void setDirector(String director) { this.director = director; }

    public String getPlot() { return plot; }

    public void setPlot(String plot) { this.plot = plot; }


}
```

Αυτή είναι η κλάση που αναπαριστά την ταινία που προσφέρει προς ενοικίαση το Renting Service. Είναι όμοια με την αντίστοιχη κλάση που έχουμε ήδη αναλύσει στο Movie service. Είναι σημαντικό όταν δουλεύουμε με microservices, αυτά να μιλάνε με το ίδιο model. Σε διαφορετική περίπτωση, αν για παράδειγμα το Movie του Renting Service και το Movie του Movie Service ήταν διαφορετικά, όταν το Rest Template έφερνε τις ταινίες από το Movie Services δε θα μπορούσε να τις μετατρέψει σε ταινία του Rentint Service.

6.4.5 Renting

```
@Entity
@Table(name = "rentings")
public class Renting {
  @Id
  @Column(name = "renting_id", nullable = false)
  private String rentingId;

  @Column(name = "movie_id", nullable = false)
  private String movieId;

  @Transient
  private String movieTitle ="";

  @Transient
  private String storyline ="";

  @Transient
  private String director ="";

  @Transient
  private String plot ="";

  @Column(name = "renter_name", nullable = false)
  private String renterName;

  @Column(name = "renting_type", nullable = false)
  private String rentingType;

  @Column(name = "renting_max", nullable = false)
  private Integer rentingMax;
```

Αντίστοιχα με την Movie, η Renting class αναπαριστά την ενοικίαση μίας ταινίας. Σε αυτή χρησιμοποιούμε τα εξής annotations:

- @Entity: υποδηλώνουμε ότι είναι ένα entity class, δηλαδή μία κλάση που θα αποθηκευτεί στη βάση ώστε να την κάνει manage το hibernate.
- @Table(name = "rentings"): δηλώνουμε το όνομα του πίνακα στον οποίο θέλουμε να αποθηκευτεί στην SQL βάση μας.
- @Id: δηλώνει ότι το field αυτό θα λειτουργεί σαν key στο table
- @Column(name = "renting_id", nullable = false): δηλώνουμε το όνομα της στήλης στο οποίο θα αποθηκευτεί το αντίστοιχο field και αν θέλουμε να επιτρέπεται null τιμή

- @Transient: υποδηλώνει ένα πεδίο το οποίο δεν είναι persistent (δεν αποθηκεύεται στη βάση). Στην εφαρμογή μας αποθηκεύουμε το movieId και επομένως αν χρειαστούμε πάλι τις πληροφορίες για το movieTitle μπορούμε να τις βρούμε πάλι (μέσω του movie service). Έτσι αφενός γλιτώνουμε duplication της πληροφορίας αλλά και να έχουμε stale δεδομένα: Εάν αλλάξει το όνομα της ταινίας στο movie service δε θα το έχουμε αποθηκευμένο λάθος στο rengint service.

6.4.6 Renting Repository

```
@Repository
public interface RentingRepository extends CrudRepository<Renting,String>  {
    public List<Renting> findByMovieId(String movieId);
    public Renting findByMovieIdAndRentingId(String movieId, String rentingId);
}
```

Το Renting Repository είναι το @Repository που μας δίνει πρόσβαση στη βάση για να αποθηκεύουμε τις ενοικιάσεις. Ακολουθεί το patter που είδαμε και στο movie repository:

- κάνει extend ένα CrudRepository<Renting,String> δηλαδή ένα CRUD (create read update delete) repository με key String και value Renting.
- προσθέτει δύο επιπλέον μεθόδους, τις
  - findByMovieId(String movieId) που επιστρέφει όλες τις ενοικιάσεις της ίδιας ταινίας
  - findbyMovieIdandRentingId(String movieId, String rentingId) που βρίσκει την ενοικίαση με συγκεκριμένο movie και renting id.

Εδώ αξίζει να σημειώσουμε ότι το Spring Data JPA (το module του spring που μας δίνει αυτό το functionality) είναι αρκετά έξυπνο στο να ξεχωρίζει και τον αριθμό των αποτελεσμάτων ενός query που κάνει μια μέθοδος. Για παράδειγμα, αναγνωρίζει ότι η findByMovieId() θα επιστρέψει πολλά αποτελέσματα γιατί έχουμε θέσει το return value σαν List . Σε αντίθετη περίπτωση, αν περιμέναμε ένα object και έβρισκε πολλά, θα είχαμε exception.

6.4.7 Renting Service Controller

Ο Renting Service Controller είναι ο Controller, δηλαδή το component που διαχειρίζεται API endpoints, μέσω του οποίου παρέχουμε το functionality των ενοικιάσεων προς το χρήστη. Ας δούμε σε τμήματα τις λειτουργίες του:

```
@RestController
@RequestMapping(value="v1/movies/{movieId}/rentings")
public class RentingServiceController {
    private static final Logger logger = LoggerFactory.getLogger(RentingServiceController.class);
    @Autowired
    private RentingService rentingService;

    @Autowired
    private ServiceConfig serviceConfig;
```

- @RestController: Στη java υπάρχει η δυνατότητα ένα annotation να συμπεριλαμβάνει σε αυτό περισσότερα annotations. Εδώ, το @RestController annotation είναι ένας συνδυασμός των @Controller και @ResponseBody . Είναι δηλαδή ένας controller και το return object των μεθόδων του χρησιμοποιείται σαν body στο response που θα επιστραφεί στο αντίστοιχο request (θα επεξηγηθεί παρακάτω)
- Ορίζουμε το path στο οποίο θα "ακούει" ο Controller μας. Αυτό το κάνουμε στο annotation @RequestMapping. Μάλιστα, το path είναι v1/movies/{movieId}/rentings στο οποίο το spring θα

αναγνωρίζει οποιαδήποτε τιμή στη θέση του movieId και θα μπορεί να μας το δώσει και στον controller σαν argument (θα το δούμε στη συνέχεια)

- κάνουμε @Autowired τα RentingService και Serviceconfig. Αυτό σημαίνει ότι το spring κατά το runtime θα κάνει inject πραγματικά objects σε αυτά τα fields.
- Δημιουργούμε ένα Logger για να μπορέσουμε να εκτυπώσουμε τα logs μας και όχι απλά να κάνουμε println()

```
@RequestMapping(value="/",method = RequestMethod.GET)
public List<Renting> getRentings(@PathVariable("movieId") String movieId) {
    logger.debug("RentingServiceController Correlation id: {}",
            UserContextHolder.getContext().getCorrelationId());
    return rentingService.getRentingsByMovie(movieId);
}
```

Δημιουργήσαμε ένα endpoint στο path "/" για μέθοδο HTTP GET. Πρέπει να σημειώσουμε ότι το path σε method level λειτουργεί προσθετικά στο αρχικό path που δηλώσαμε σε class level. Δηλαδή:

τελικό endpoint path = class level path + method level path

Σε αυτή τη μέθοδο επιστρέφουμε μία λίστα από ενοικιάσεις και παίρνουμε σαν argument ένα movieId. Αυτό το argument θα το βρει μόνο του το spring από το path στο οποίο έγινε το GET . Την λειτουργία αυτή την παίρνουμε μέσω του @PathVariable

Το movieId αυτό το περνάμε στο Renting Service και βρίσκουμε όλες τις ενοικιάσεις για αυτή την ταινία και τις επιστρέφουμε σε λίστα.

Η μέθοδος getRentings() έχει σαν return value ένα List<Rentings> . Επειδή έχουμε χρησιμοποιήσει το annotation @ResponseBody, αυτή η λίστα θα δοθεί σαν body στο HTTP response με τη μορφή JSON.

```
@RequestMapping(value="/{rentingId}",method = RequestMethod.GET)
public Renting getRentings(@PathVariable("movieId") String movieId,
                           @PathVariable("rentingId") String rentingId) {

    return rentingService.getRenting(movieId, rentingId);
}
```

Εδώ αλλάζουμε το path, προσθέτοντας και το rentingId και χρησιμοποιούμε δύο @PathVariable ένα για το movieId και ένα για το rentingId. Αντίστοιχα επιστρέφουμε την ενοικίαση που αντιστοιχεί στα ids αυτά αφού την έχουμε βρει μέσω του Renting Service.

```
@RequestMapping(value="{rentingId}",method = RequestMethod.PUT)
public void updateRentings(@PathVariable("rentingId") String rentingId,
                           @RequestBody Renting renting) {
    rentingService.updateRenting(renting);
}
```

Το ιδιαίτερο σε αυτό το endpoint είναι ότι ακούσει σε μέθοδο HTTP PUT. Γενικά όταν κάνουμε PUT σε κάποιο συγκεκριμένο resource αντικαθιστούμε το resource αυτό με εκείνο που δίνουμε σαν argument. Χρησιμεύει δηλαδή για να κάνουμε update κάποιο resrouce. Αντίστοιχα και εδώ κάνουμε update το Renting που αντιστοιχεί στο rentingId που έχουμε πάρει από το @PathVariale με το Renting

που μας δίνεται μέσα στο @RequestBody . Request body είναι το body του HTTP request που μας γίνεται και το Spring κάνει μόνο του το deserialization από json σε java object.

```
@RequestMapping(value="/",method = RequestMethod.POST)
public void saveRentings(@RequestBody Renting renting) { rentingService.saveRenting(renting); }
```

Αντίστοιχα, η HTTP POST παίρνει ένα resource από το request και το δημιουργεί/σώζει χωρίς αυτό να υπάρχει από πριν. Έτσι και εδώ παίρνει ένα Renting (ενοικίαση) από το @RequestBody και το σώζει μέσω του Renting Service.

```
@RequestMapping(value="{rentingId}",method = RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteRenting(@PathVariable("rentingId") String rentingId,
                          @RequestBody Renting renting) {
    rentingService.deleteRenting(renting);
}
```

Αντίστοιχα για να διαγράψουμε ένα resource χρησιμοποιούμε HTTP GET. Εδώ διαγράφουμε την αντίστοιχη ενοικίαση. Με το annotation @ResponseStatus(HttpStatus.NO_CONTENT) επιστρέφουμε ένα HTTP response με κωδικό 204 , δηλαδή ότι διαγράφτηκε και δε χρειάζεται κάποια επιπλέον πράξη.

6.4.8 Discovery Service

```
@Service
public class DiscoveryService {
    @Autowired
    private DiscoveryClient discoveryClient;

    public List getEurekaServices(){
        List<String> services = new ArrayList<>();

        discoveryClient.getServices().forEach(serviceName -> {
            discoveryClient.getInstances(serviceName).forEach(instance->{
                services.add( String.format("%s:%s",serviceName,instance.getUri()));
            });
        });

        return services;
    }
```

Το Discovery service είναι ένα @Service (δηλαδή ένα spring component στο οποίο συγκεντρώνουμε business logic) το οποίο το χρησιμοποιούμε περισσότερο για debugging. Σκοπός του είναι να προσφέρει όλα τα services που είναι available στο eureka server.

Για να το κάνουμε αυτό κάνουμε @Autowired το DiscoveryClient και με discoveryClient.getInstances() παίρνουμε όλα τα services. Βλέπουμε την απλότητα του Spring καθώς δε χρειάζεται να ξέρουμε τίποτα για το πώς αυτό επικοινωνεί με το Eureka. Τέλος κάνουμε ένα απλό string manipulation στη μορφή που θέλουμε και τα επιστρέφουμε

6.4.9 Tools Controller

```
@RestController
@RequestMapping(value="v1/tools")
public class ToolsController {
    @Autowired
    private DiscoveryService discoveryService;

    @RequestMapping(value="/eureka/services",method = RequestMethod.GET)
    public List<String> getEurekaServices() {

        return discoveryService.getEurekaServices();
    }
}
```

Ο συγκεκριμένος Controller δεν αποτελεί τμήμα του core functionality της εφαρμογής αλλά είναι χρήσιμος για το debugging. Κάποιες φορές είναι χρήσιμο για το developer να βλέπει σε ένα live microservice με ποιά άλλα microservices επικοινωνεί. Για αυτό το λόγο, έχουμε κάνει expose τα microserves που βλέπει μέσω του Eureka σε ένα /tools endpoint. Απλά κάνουμε autowire το DiscoveryService και επιστρέφουμε τα Eureka Services.

6.4.10 Renting Service

Είναι το service που περιλαμβάνει το βασικό business logic του Renting service. Περιλαμβάνει τα calls που πρέπει να γίνουν στα άλλα services (movie services) και χρησιμοποιεί resilience design patterns για να επικοινωνεί μαζί τους. Θα αναλύσουμε διεξοδικά την υλοποίηση και παραμετροποίηση των pattern αυτών, αλλά η θεωρητική τους περιγραφή γίνεται σε ξεχωριστό κεφάλαιο.

Λόγω του μεγέθους και της πολυπλοκότητας της κλάσης αυτής θα την αναλύσουμε τμηματικά.

```
@Service
public class RentingService {
    private static final Logger logger = LoggerFactory.getLogger(RentingService.class);
    @Autowired
    private RentingRepository rentingRepository;

    @Autowired
    ServiceConfig config;

    @Autowired
    MovieRestTemplateClient movieRestClient;
```

- @Service: Ειδοποιούμε το Spring ότι αυτό είναι ένα spring component και πρέπει να το κάνει manage σαν spring bean
- δημιουργούμε ένα slf4j logger για να εκτυπώνουμε τα logs μας.
- κάνουμε @Autowired τα beans RentingRepository, ServiceConfig και MovieRestTemplateClient. Το dependency injection του spring θα μας τα φέρει στο runtime χωρίς να χρειαστεί εμείς να κάνουμε κάτι άλλο.

```
public Renting getRenting(String movieId, String rentingId) {
    Renting renting = rentingRepository.findByMovieIdAndRentingId(movieId, rentingId);

    Movie movie = getMovie(movieId);

    return renting
            .withMovieTitle( movie.getTitle())
            .withStoryline( movie.getStoryline())
            .withPlot( movie.getDirector() )
            .withDirector( movie.getPlot() )
            .withComment(config.getExampleProperty());
}

@HystrixCommand
private Movie getMovie(String movieId) {
    return movieRestClient.getMovie(movieId);
}
```

Η μέθοδος getRenting(String movieId, String rentingId) παίρνει σαν arguments ένα movieId και ένα rentingId και επιστρέφει την αντίστοιχη ενοικίαση. Το σωστό object το βρίσκει από τη βάση μέσω του RentingRepository κάνοντας ένα query findByMovieIdAndRentingId()

Όμως όπως είδαμε στην Renting.class οι πληροφορίες σχετικές με την ταινία δεν αποθηκεύονται στη βάση για να αποφύγουμε stale data. Επομένως το object που μας επιστρέφεται από το repository είναι ελλιπές.

Τις επιπλέον πληροφορίες θα τις φέρουμε από το movieRestClient μέσω της getMovie() μεθόδου. Αυτή χρησιμοποιεί το Movie Rest Template Client για να φέρει τις πληροφορίες. Ο Movie Rest Template Client χρησιμοποιεί ένα rest template, όπως είδαμε, για να επικοινωνήσει με το movie service αλλά για το RentingService.class αυτό είναι transparent.

Το πιο σημαντικό είναι το annotation **@HystrixCommand** στηνgetMovie(). Με αυτό το annotation κάνουμε wrap την getMovie() με ένα Hystrix Circuit Breaker. Δηλαδή το Hystrix θα κάνει monitor τη μέθοδο αυτή και αν αργήσει να επιστρέψει θα τη διακόψει για να μην έχουμε degradation. Αυτός είναι ο πιο βασικός τρόπος που μπορεί να χρησιμοποιηθεί το Hystrix και χρησιμοποιεί όλες τις defualt τιμές (για παράδειγμα το timeout στα 1000 milliseconds). Να σημειωθεί ότι για να λειτουργήσει το @HystrixCommand πρέπει να συνδυάζεται με το @EnableCircuitBreaker που στη δική μας περίπτωση το έχουμε βάλει στο application.class

```
@HystrixCommand(fallbackMethod = "buildFallbackMovieList",
        threadPoolKey = "rentingByMovieThreadPool",
        threadPoolProperties =
                {@HystrixProperty(name = "coreSize",value="30"),
                 @HystrixProperty(name="maxQueueSize", value="10")},
        commandProperties={
                 @HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="10"),
                 @HystrixProperty(name="circuitBreaker.errorThresholdPercentage", value="75"),
                 @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="7000"),
                 @HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="15000"),
                 @HystrixProperty(name="metrics.rollingStats.numBuckets", value="5")}
)
public List<Renting> getRentingsByMovie(String movieId){
    logger.debug("RentingService.getRentingsByMovie  Correlation id: {}",
            UserContextHolder.getContext().getCorrelationId());
    randomlyRunLong();

    return rentingRepository.findByMovieId(movieId);
}

private List<Renting> buildFallbackMovieList(String movieId){
    List<Renting> fallbackList = new ArrayList<>();
    Renting renting = new Renting()
            .withId("0000000-00-00000")
            .withMovieId( movieId )
            .withRenterName("Sorry no renting information currently available")
            .withComment("Sorry no renting information currently available")
            .withDirector("Sorry no renting information currently available");
    fallbackList.add(renting);
    return fallbackList;
}
```

Με την getRentingsByMovie() βρίσκουμε όλα τις ενοικιάσεις για μία συγκεκριμένη ταινία. Πάλι όπως βλέπουμε έχουμε κάνει wrap την μέθοδο με @HystrixCommand , ενεργοποιώντας δηλαδή το circuit breaker, αλλά αυτή τη φορά χρησιμοποιούμε δικό μας συγκεκριμένο configuration αντί του default:

- fallbackMethod: Σε περίπτωση που ενεργοποιηθεί το circuit breaker θα εκτελεστεί αυτή η μέθοδος σαν **fallback**. Έχουμε δηλαδή συνδυασμό του **circuit breaker** με **fallback** pattern
- threadPoolKey: τα calls αυτού του hystrix command θα γίνουν σε ένα συγκεκριμένο ξεχωριστό thread pool με αυτό το όνομα. Έχουμε δηλαδή συνδυασμό του **circuit breaker** με **bulkheads**
- threadPoolProperties: δίνουμε επιπλέον configuration για fine-tuning του hystrix command:
    - coreSize: το μέγεθος του thread pool
    - maxQueueSize: το μέγιστο μέγεθος του queue μπροστά από το thread pool
    - circuitBreaker.requestVolumeThreshold: ο μίνιμουμ αριθμός call που πρέπει να γίνουν process σε ένα συγκεκριμένο time window πριν το hystrix αρχίσει να εξετάζει αν πρέπει να ενεργοποιήσει το circuit breaker
    - circuitBreaker.error-ThresholdPercentage: το ποσοστό των calls που πρέπει να γίνουν fail μέσα σε ένα συγκεκριμένο time window ώστε να ενεργοποιηθεί το circuit breaker.
    - circuitBreaker.sleepWindowInMilliseconds: ο χρόνος που το hystrix θα περιμένει πριν ξαναδοκιμάσει αυτό το call αφότου το circuit breaker ενεργοποιηθεί.
    - metrics.rollingStats.timeInMilliseconds: το time window στο οποίο αναφερόμασταν πριν
    - metrics.rollingStats.numBuckets: ο αριθμός των buckets στο οποίο χωρίζεται το time window

Με άλλα λόγια με αυτό το configuration το Hystrix θα εκτελέσει όλα τα calls αυτής της μεθόδου σε ένα threadpool με όνομα threadPoolKey και με coreSize αριθμό threads. Αν κάποιο call κάνει περισσότερο από το timeoutInMilliseconds θα διακόψει το call και θα εκτελέσει την fallbackMethod αντί αυτού. Αν τώρα μέσα σε ένα rolling time window διάρκειας metrics.rollingStats.timeInMilliseconds πάνω από το circuitBreaker.error-ThresholdPercentage γίνουν fail, τότε θα ενεργοποιηθεί το **circuit breaker**. Αυτό σημαίνει ότι για το Hystrix το resource αυτό είναι unavailable και δε θα δοκιμάσει άλλα calls σε αυτό πριν περάσουν sleepWindowInMilliseconds δευτερόλεπτα.

Εφόσον σε εμάς η getRentingsByMovie() χτυπάει το RentingRepository που γνωρίζουμε ότι είναι local resource και όχι remote δε θα κάνει ποτέ timeout. Για αυτό με την randomlyRunLong() κάνουμε επίτηδες κάποια τυχαία calls να κάνουν timeout.

Σε αυτή την περίπτωση το hystrix εκτελεί την buildFallbackMovieList() που επιστρέφει κάποιες dummy data.

```
private void randomlyRunLong(){
  Random rand = new Random();

  int randomNum = rand.nextInt( bound: (3 - 1) + 1) + 1;

  if (randomNum==3) sleep();
}

private void sleep(){
    try {
        Thread.sleep( millis: 11000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Η υλοποίηση της rundomlyRunLong() είναι ιδιαίτερα απλή και προκαλεί μία καθυστέρηση (Thread.sleep(1100) σε μία στις τρεις εκτελέσεις της. Αυτή η καθυστέρηση είναι αρκετή για να ενεργοποιήσει το fallback του hystrix

```
public void saveRenting(Renting renting){
    renting.withId( UUID.randomUUID().toString());

    rentingRepository.save(renting);
}

public void updateRenting(Renting renting){
    rentingRepository.save(renting);
}

public void deleteRenting(Renting renting){
    rentingRepository.delete( renting.getRentingId());
}
```

Οι saveRenting(), updateRenting() και deleteRenting() κάνουν απλά save, update kai delete τις αντίστοιχες ενοικιάσεις μέσω του RentingRepository.

6.4.11 Resource files

Και εδώ χρησιμοποιούμε τρία διαφορετικά resource files:

6.4.11.1 bootstrap.yml

```
spring:
  application:
    name: rentingservice
  profiles:
    active:
      default
  cloud:
    config:
      enabled: true
```

Εδώ κάνουμε ρυθμίσεις που φορτώνονται κατά το bootstraping της εφαρμογής:

- spring.application.name : το όνομα της εφαρμογής μας
- spring.profiles.active: το spring profile που ενεργοποιείται αν δεν ορίσουμε κάποιο άλλο
- spring.cloud.config.enabled: ενεργοποιούμε τη σύνδεση με το cloud config server ώστε να πάρει και το επιπλέον shared configuration

6.4.11.2 application.yml

```yaml
eureka:
  instance:
    preferIpAddress: true
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

#Setting the logging levels for the service
logging:
  level:
    com.netflix: WARN
    org.springframework.web: WARN
    com.microcorp: DEBUG
```

- eureka.client.registerWithEureka: registration του service στο eureka server
- eureka.client.fetchRegistry : local caching του eureka registry
- eureka.client.serviceUrl.defaultZone: επιλέγουμε σε ποιο συγκεκριμένο zone θα κάνει register στο eureka
- eureka.instance.preferIpAddress: προτιμούμε IPs από DNS names
- logging.level: επιλέγουμε το επίπεδο logging που θέλουμε για τα διάφορα components

6.4.11.3 schema.sql

```sql
DROP TABLE IF EXISTS rentings;

CREATE TABLE rentings (
  renting_id        VARCHAR(100) PRIMARY KEY NOT NULL,
  movie_id    TEXT NOT NULL,
  renting_type      TEXT NOT NULL,
  renter_name      TEXT NOT NULL,
  renting_max       INT   NOT NULL,
  renting_allocated INT,
  comment           VARCHAR(100));


INSERT INTO rentings (renting_id,  movie_id, renting_type, renter_name, renting_max, renting_allocated)
VALUES ('f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a', 'e254f8c-c442-4ebe-a82a-e2fc1d1ff78a', 'pre-paid','John Smith', 100,5);
INSERT INTO rentings (renting_id,  movie_id, renting_type, renter_name, renting_max, renting_allocated)
VALUES ('t9876f8c-c338-4abc-zf6a-ttt1', 'e254f8c-c442-4ebe-a82a-e2fc1d1ff78a', 'paid-after','Jack Black', 7, 3);
INSERT INTO rentings (renting_id,  movie_id, renting_type, renter_name, renting_max, renting_allocated)
VALUES ('38777179-7094-4200-9d61-edb101c6ea84', '442adb6e-fa58-47f3-9ca2-ed1fecdfe86c', 'paid-after','Helen Brown', 15,8);
INSERT INTO rentings (renting_id,  movie_id, renting_type, renter_name, renting_max, renting_allocated)
VALUES ('08dbe05-606e-4dad-9d33-90ef10e334f9', '442adb6e-fa58-47f3-9ca2-ed1fecdfe86c', 'pre-paid','Mary White', 16,16);
```

Και εδώ για να αρχικοποιήσουμε τη βάση μας χρησιμοποιούμε το schema.sql αρχείο το οποίο αυτόματα το διαβάζει το Spring και εκτελεί τις SQL εντολές που περιέχει:

1. κάνει DROP τον πίνακα εάν υπάρχει
2. δημιουργεί τον πίνακα που αντιστοιχεί στο object Renting

3. εισάγει (INSERT) κάποια dummy data

7 Resilience Design Patterns

Όταν σχεδιάζουμε μία εφαρμογή με microservices πρέπει στην αρχιτεκτονική και στην υλοποίησή μας να κάνουμε αυτό που ονομάζουμε *paradigm shift*, δηλαδή να αναθεωρήσουμε πράγματα που μέχρι τώρα θεωρούσαμε δεδομένα. Σε μια κλασική μονολιθική εφαρμογή τα διάφορα modules με τα οποία επικοινωνεί το component μας είναι τμήμα της ίδιας εφαρμογής, γνωρίζουμε ακριβώς που βρίσκεται και πώς να επικοινωνήσουμε μαζί του, είναι ένα και μοναδικό και μπορούμε να θεωρήσουμε αυτονόητο ότι θα μας απαντήσει, διαφορετικά η εφαρμογή δε θα λειτουργεί.

Αντίθετα, σε μια εφαρμογή με microservices, όταν ένα service επικοινωνεί με κάποιο άλλο πρέπει να λαμβάνουμε υπόψη μας τα εξής:

1. **δεν γνωρίζουμε πού βρίσκεται αυτό το service***: μπορεί να είναι κομμάτι της εφαρμογής μας, μπορεί εξωτερικό service, μπορεί να είναι service της εφαρμογής μας που λειτουργεί σαν proxy για κάποιο εξωτερικό
2. **δεν γνωρίζουμε πόσα όμοια services υπάρχουν:** μπορεί να υπάρχουν πολλά διαφορετικά instances άρα δεν μπορούμε να είμαστε βέβαιοι για το load που διαχειρίζεται το καθένα
3. **δεν γνωρίζουμε σε πόσο χρόνο θα μας απαντήσει**: εφόσον δεν γνωρίζουμε που βρίσκεται και τι load έχει δεν μπορούμε να εγγυηθούμε ότι θα απαντήσει σε συγκεκριμένο χρόνο
4. **δεν γνωρίζουμε *εάν* θα μας απαντήσει:** μπορεί το συγκεκριμένο service να μην είναι καν available

Καταλαβαίνουμε ότι καθώς η αρχιτεκτονική της εφαρμογής με microservices είναι ευμετάβλητη, έτσι και η αρχιτεκτονική της υλοποίησής μας πρέπει να είναι "αμυντική" και να προφυλασσόμαστε από τυχόν αποτυχίες. Για να το επιτύχουμε αυτό υπάρχουν τρία πολύ αποτελεσματικά design patterns:

1. circuit breaker
2. fallback
3. bulkeheads

7.1.1 Circuit Breaker

Το Circuit Breaker παίρνει το όνομά του από τα ηλεκτρικά κυκλώματα. Όταν θέλουμε να προφυλάξουμε ένα κύκλωμα από υπερφόρτωση του βάζουμε μία ασφάλεια η οποία καίγεται "κόβοντας" το κύκλωμα. Με αυτό τον τρόπο προστατεύουμε τις συσκευές του κυκλώματος αλλά το κύκλωμά μας πλέον δεν λειτουργεί.

Αντίστοιχα, στα microservices όταν θέλουμε να κάνουμε ένα call σε ένα άλλο service για το οποίο δεν είμαστε σίγουροι σε πόση ώρα θα μας απαντήσει, παρατηρούμε το call και το "κόβουμε" όταν αργήσει παραπάνω από κάποιο όριο. Επίσης όταν αρκετά calls σε ένα συγκεκριμένο απομακρυσμένο service έχουν αποτύχει τότε θεωρούμε το service αυτό unavailable και δεν ξοδεύουμε χρόνο στο να δοκιμάζουμε να το ξαναχτυπήσουμε.

7.1.2 Fallback

Κάθε remote call σε κάποιο άλλο service γίνεται για κάποιο λόγο και επομένως κάθε φορά που γίνεται fail ένα τέτοιο call χάνουμε ένα τμήμα της λειτουργικότητας. Αντί η εφαρμογή μας να πετάξει απλά κάποιο exception και να ακολουθήσει ένα error case με το fallback design pattern όταν κάποιο remote call αποτύχει το αντικαθιστούμε με κάποια άλλη ενέργεια.

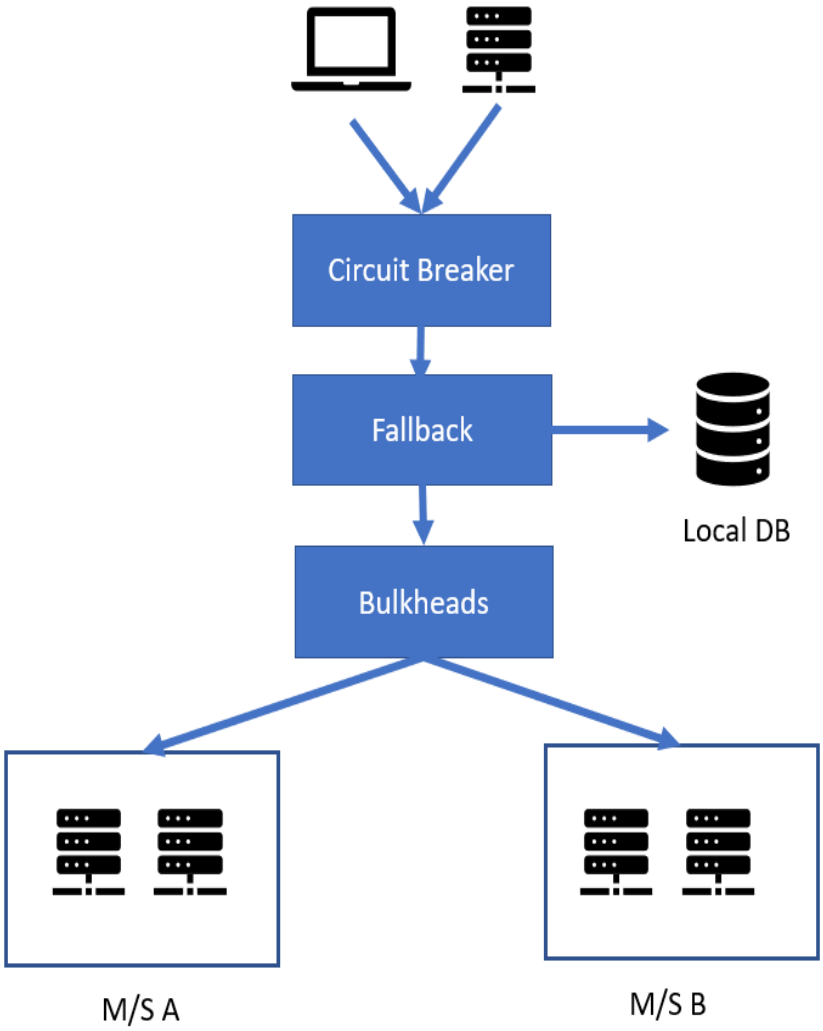
Για παράδειγμα, ας φανταστούμε στο backend ενός social site ότι υπάρχει ένα call σε ένα remote service που κάνει data analysis και επιστρέφει τα πιο relevant posts για κάποιο συγκεκριμένο χρήστη. Στην περίπτωση που το service αυτό δεν απαντήσει, αντί να χάσουμε τελείως τη

λειτουργικότητα μπορούμε με fallback να αντικαταστήσουμε τα relevant posts με τα πιο πρόσφατα. Έτσι προσφέρουμε κάποια λειτουργικότητα στο χρήστη.

7.1.3 Bulkheads

Το design pattern αυτό προέρχεται από τα πλοία όπου ο σκελετός του πλοίου χωρίζεται σε αυτόνομα και υδατοστεγή (bulkheads) τμήματα ώστε αν κάποιο από αυτά έχει διαρροή να μην πλημμυρίσει ολόκληρο και βουλιάξει.

Αντίστοιχα, στα microservices επειδή συνήθως έχουμε πολλά calls σε πολλά διαφορετικά services καλό είναι να τα διαχωρίζουμε σε ξεχωριστά thread pools ώστε ένα ενδεχόμενο πρόβλημα σε κάποιο άλλο service να μην επηρεάζει την επικοινωνία με τα υπόλοιπα. Τα thread pools λειτουργούν δηλαδή σαν bulkheads.



7.1.4 Spring Cloud Netflix Hystrix

Θεωρητικά κάθε ένα από αυτά τα patterns είναι απλά στην κατανόηση και θα μπορούσε ο κάθε developer να τα κάνει implement μόνος του. Όμως αυτό αφενός θα ήταν χρονοβόρο αφετέρου θα εισήγαγε επιπλέον κώδικα απλά για να κάνουμε monitor τη λειτουργικότητα του ήδη υπάρχοντος κώδικα.

Αυτό στα microservices και με το Spring δεν το θέλουμε και μπορούμε να το αποφύγουμε με το Spring Cloud Netflix Hystrix.

Το Spring Cloud Netflix Hystrix είναι μία βιβλιοθήκη που επιτρέπει στο developer να κάνει χρήση των τριών αυτών pattern σε όποιο call επιθυμεί εύκολα και γρήγορα. Συγκεκριμένα δίνει τις εξής δυνατότητες:

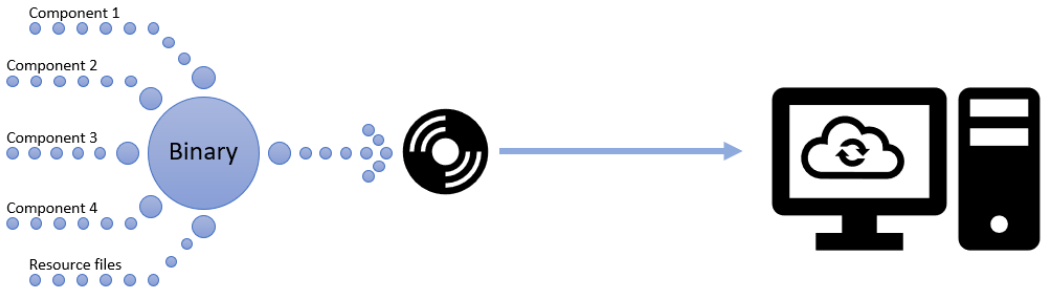
- Να κάνει wrap κάποιο remote call με circuit breaker
- Να ενεργοποιεί τα circuit breaker με διαφορετικά timeouts.
- Να αναθέτει σε κάθε circuit breaker ένα fallback strategy για να το εκτελεί όταν "κοπεί" το circuit breaker
- Να χρησιμοποιεί διαφορετικά thread pools για τα διαφορετικά circuit breakers ώστε να χρησιμοποιεί και το bulkhead pattern.

Το Hystrix χρησιμοποιεί annotations για να ενεργοποιηθεί και περισσότερες λεπτομέρειες τόσο για την λειτουργικότητά του όσο και για την παραμετροποίησή του βρίσκονται στην ανάλυση του Renting Service

8 Deployment

Σε αυτό το σημείο έχουμε αναπτύξει τα αρνητικά των μονολιθικών εφαρμογών, πώς τα microservices εμφανίστηκαν παρουσιάζοντας μία καινούργια αρχιτεκτονική και πώς ο προγραμματιστής μπορεί να σχεδιάσει και να αναπτύξει μία εφαρμογή με microservices αποτελεσματικά. Δεν έχουμε μιλήσει καθόλου όμως για το επόμενο βήμα μετά την ανάπτυξη του κώδικα, το **deployment:** τον τρόπο με τον οποίο τα εκτελέσιμα συγκεντρώνονται μαζί με τα απαραίτητα configuration αρχεία και τα τρέχουμε σε κάποιο server.

Στις μονολιθικές εφαρμογές αυτό το βήμα ήταν απλό:



Καθώς όλα τα επιμέρους components ήταν απλά τμήματα της ίδιας εφαρμογής, το μόνο που χρειαζόταν ήταν το compilation και το packaging σε μια μορφή (jar, war etc) αρχείου που ο server θα μπορούσε να εκτελέσει. Στη συνέχεια, το deployment ήταν απλά η μεταφορά και η εκτέλεση του αρχείου στο server. Δυνατότητα για scaling είχαμε μόνο κάθετα με τη χρήση κάποιου καλύτερου server.

Σε μια εφαρμογή με microservices αυτό δεν είναι δυνατόν. Κάθε εφαρμογή είναι ξεχωριστή, χρειάζεται να τρέχει σε ένα ή περισσότερα instances και στον ίδιο ή διαφορετικούς servers. Ξαφνικά το deployment από κάτι απλό έγινε ένα καινούργιο πρόβλημα που χρειάζεται να αντιμετωπίσουμε.

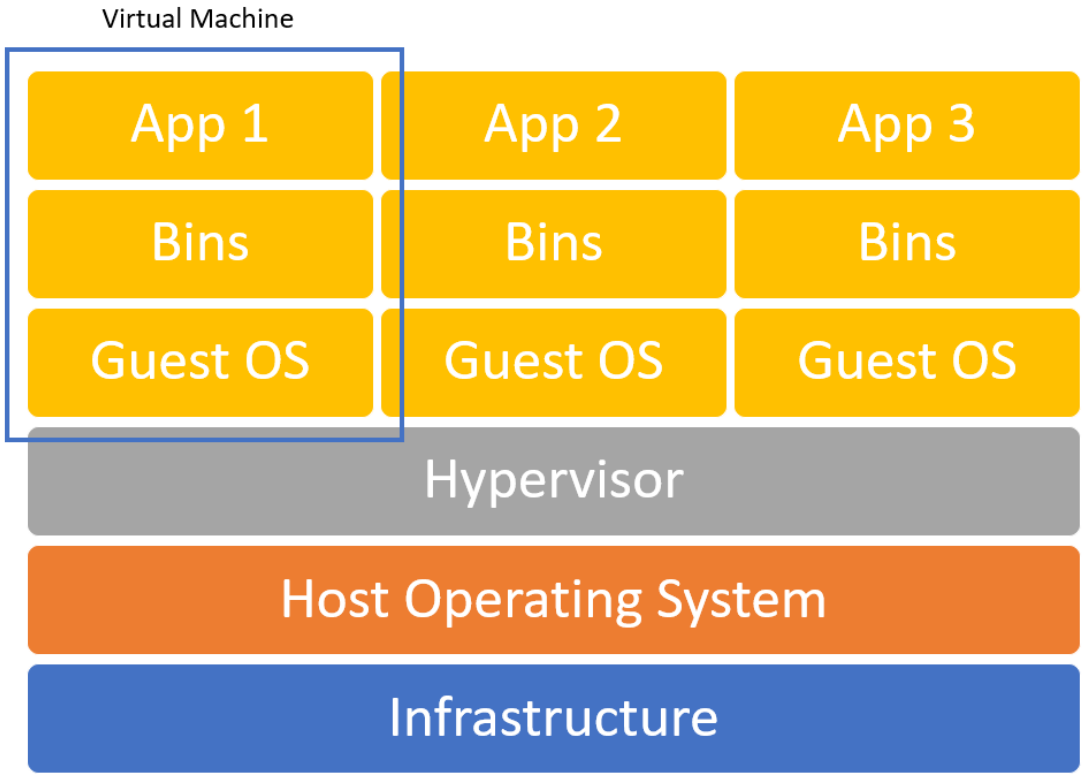
8.1 Docker containers

Μία λύση για το συγκεκριμένο πρόβλημα είναι η χρήση docker containers. Η χρήση τους είναι τόσο διαδεδομένη που πλέον θεωρείται defacto τρόπος υλοποίησης. Τι είναι όμως το docker και τι τα docker containers

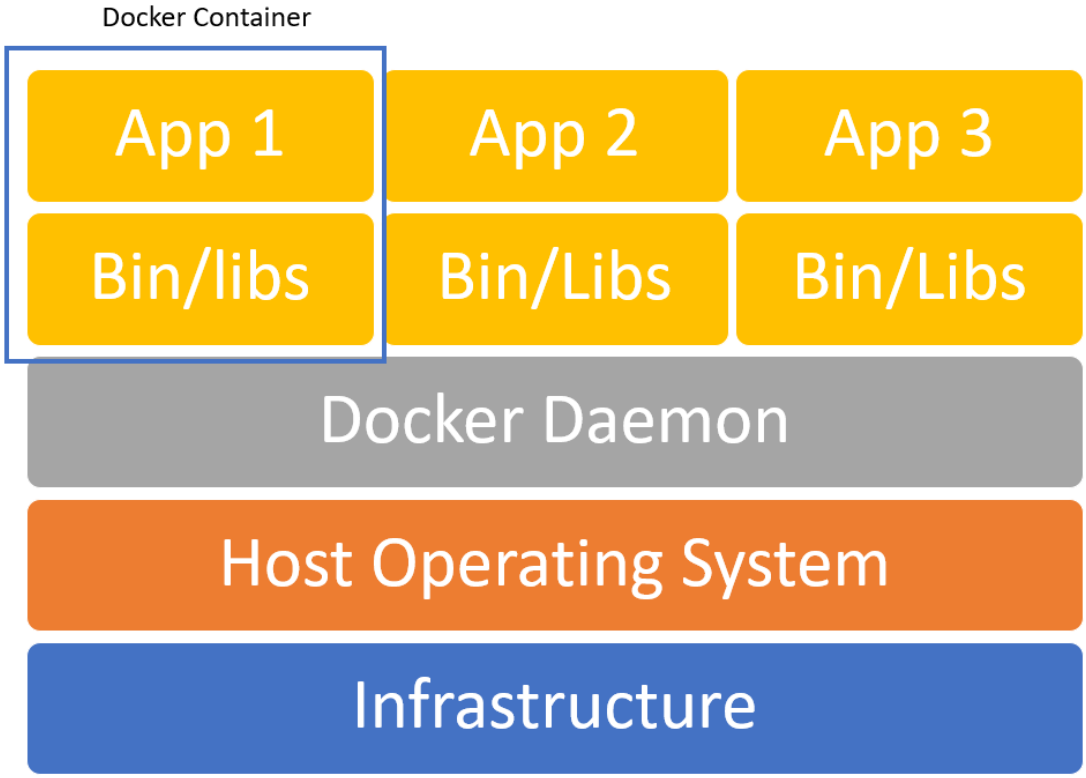
Το Docker είναι ένα εργαλείο σχεδιασμένο για να διευκολύνει τη δημιουργία, την ανάπτυξη και την εκτέλεση εφαρμογών χρησιμοποιώντας containers. Τα containers επιτρέπουν σε έναν προγραμματιστή να συσκευάσει μια εφαρμογή με όλα τα μέρη που χρειάζεται, όπως βιβλιοθήκες και άλλες εξαρτήσεις, και να την αναπτύξει ως ένα ενιαίο πακέτο. Με τον τρόπο αυτό, χάρη στο container, ο προγραμματιστής μπορεί να είναι βέβαιος ότι η εφαρμογή θα τρέξει σε οποιοδήποτε άλλο μηχάνημα Linux ανεξάρτητα από τυχόν προσαρμοσμένες ρυθμίσεις που θα μπορούσε να έχει αυτό το μηχάνημα και που διαφέρουν από το μηχάνημα που χρησιμοποιείται για το implementation.

Θα μπορούσαμε να πούμε ότι το Docker είναι ένα virtual machine. Αντίθετα όμως από ένα virtual machine, αντί να δημιουργήσει ένα ολόκληρο εικονικό λειτουργικό σύστημα, το Docker επιτρέπει στις εφαρμογές να χρησιμοποιούν τον ίδιο πυρήνα Linux με το σύστημα στο οποίο εκτελούν και απαιτεί μόνο να γίνονται deployed εφαρμογές που δεν εκτελούνται ήδη στον κεντρικό υπολογιστή. Αυτό δίνει σημαντική ενίσχυση της απόδοσης και μειώνει το μέγεθος της εφαρμογής.

Αυτή είναι μία σημαντική διαφορά και για να την καταλάβουμε ας δούμε σχηματικά τι περιλαμβάνει ένα virtual machine:



Όπως βλέπουμε περιλαμβάνει όλο το stack κάτω από την εφαρμογή και αναγκαζόμαστε να κάνουμε deploy ολόκληρο το OS. Αντίθετα, όταν χρησιμοποιούμε docker containers:



Παρατηρούμε ότι το container είναι πολύ πιο μικρό και περιλαμβάνει μόνο τα διαφορετικά stacks που διαφοροποιούν το περιβάλλον μιας εφαρμογής. Για να μπορέσουμε να χρησιμοποιήσουμε τα docker containers χρειάζονται τα εξής βήματα:

1. Δημιουργία του container
2. Αποθήκευση του container σε κάποιο container registry
3. orchestration των containers από το registry ώστε να δημιουργηθεί το τελικό architecture

8.2 Δημιουργία των container

Για να δημιουργήσουμε ένα docker container χρειάζεται να ξεκινήσουμε από κάποιο base container και να αρχίσουμε να του "προσθέτουμε" έξτρα πακέτα. Αυτά τα πακέτα είναι τα απαραίτητα για να τρέξει η εφαρμογή μας. Για παράδειγμα εάν θέλουμε να δημιουργήσουμε ένα docker container για μία εφαρμογή που τρέχει σε linux, και χρειάζεται τις βιβλιοθήκες X και Y για να τρέξει το σωστό container θα περιλαμβάνει τα εξής:

- base linux container
- βιβλιοθήκη X
- βιβλιοθήκη Y
- εκτελέσιμο της εφαρμογής

Για ευνόητους λόγους θέλουμε να κρατάμε τα containers όσο μικρότερα γίνεται και με το ελάχιστο configuration που χρειάζεται. Αυτά τα βήματα μπορούμε να τα κάνουμε κάθε φορά με το χέρι , αλλά αυτό προφανώς δεν ταιριάζει σε ένα workflow καθημερινού και business-grade development. Στην εφαρμογή μας το αντιμετωπίσαμε ως εξής:

Χρησιμοποιούμε ένα **maven plugin** ώστε όταν κάνουμε build το εκτελέσιμο της εφαρμογής μας αυτό να δημιουργεί και το σωστό docker container. Πώς όμως γνωρίζει το plugin πώς να το κάνει; Αυτό το κάνουμε με συνδυασμό δύο αρχείων: του dockerfile αλλά και του pom.xml

8.2.1 Dockerfile

Σε όλα τα microservices μας έχουμε στο path ../src/main/docker ένα αρχείο dockerfile. Αυτό περιλαμβάνει τα βήματα με τα οποία θα κατασκευαστεί το docker container για το service αυτό. Ας δούμε το dockerfile αρχείο του Renting Service:

```
FROM openjdk:8-jdk-alpine
RUN  apk update && apk upgrade && apk add netcat-openbsd
RUN mkdir -p /usr/local/rentingservice
ADD @project.build.finalName@.jar /usr/local/rentingservice/
ADD run.sh run.sh
RUN chmod +x run.sh
CMD ./run.sh
```

Αυτές είναι docker εντολές και κάνουν τα εξής:

- FROM openjdk:8-jdk-alpine : Υποδηλώνουμε από πιο base container θα ξεκινήσουμε. Διαλέγουμε ένα aline linux με openjdk toolkit για java 8. Το alpine linux είναι ένα μίνιμαλ linux distribution για να κρατήσουμε το μέγεθος μικρό. Το image αυτό θα το κατεβάσει από το docker hub.
- RUN apk update && apk upgrade && apk add netcat-openbsd: κάνουμε update τα υπάρχοντα πακέτα του alpine και προσθέτουμε το netcat-openbsd. Όπως είπαμε το alpine έχει όσο το δυνατόν λιγότερα πακέτα by default
- RUN mkdir -p /usr/local/rentingservice : δημιουργούμε μέσα στο linux το φάκελο `/usr/local/rentingservice``
- `ADD @project.build.finalName@.jar /usr/local/rentingservice/ : προσθέτουμε το .jar αρχείο της εφαρμογής στο φάκελο που μόλις δημιουργήσαμε. Το @project.build.finalName@ είναι ένα placeholder για το maven plugin ώστε να έχουμε παραμετροποιήσιμο όνομα αρχείου. Θα δούμε παρακάτω πώς γίνεται ακριβώς το configuration του plugin.
- ADD run.sh run.sh :προσθέτουμε το αρχείο run.sh που βρίσκεται στον ίδιο φάκελο
- RUN chmod +x run.sh: δίνουμε στο run.sh permissions εκτελέσιμου
- CMD ./run.sh: εκτελούμε το run.sh

Σκοπός μας όπως είδαμε είναι να δημιουργήσουμε το κατάλληλο περιβάλλον για να τρέξει το service, να βάλουμε μέσα το .jar του service και να το τρέξουμε με το run.sh. Ο λόγος που δεν τρέξαμε το jar με ένα απλό java -jar και χρησιμοποιούμε ένα επιπλέον script είναι γιατί χρειαζόμαστε λίγο περισσότερο logic. Για να το καταλάβουμε ας δούμε το run.sh του Renting service:

```
#!/bin/sh

echo "*********************************************************"
echo "Waiting for the eureka server to start on port $EUREKASERVER_PORT"
echo "*********************************************************"
while ! `nc -z eurekaserver $EUREKASERVER_PORT`; do sleep 3; done
echo "******* Eureka Server has started"


echo "*********************************************************"
echo "Waiting for the database server to start on port $DATABASESERVER_PORT"
echo "*********************************************************"
while ! `nc -z database $DATABASESERVER_PORT`; do sleep 3; done
echo "******** Database Server has started "

echo "*********************************************************"
echo "Waiting for the configuration server to start on port $CONFIGSERVER_PORT"
echo "*********************************************************"
while ! `nc -z configserver $CONFIGSERVER_PORT`; do sleep 3; done
echo "*******  Configuration Server has started"


echo "*********************************************************"
echo "Starting Renting Server with Configuration Service via Eureka :  $EUREKASERVER_URI" ON PORT: $SERVER_PORT;
echo "*********************************************************"
java -Djava.security.egd=file:/dev/./urandom -Dserver.port=$SERVER_PORT   \
     -Deureka.client.serviceUrl.defaultZone=$EUREKASERVER_URI             \
     -Dspring.cloud.config.uri=$CONFIGSERVER_URI                          \
     -Dspring.profiles.active=$PROFILE -jar /usr/local/rentingservice/@project.build.finalName@.jar
```

Όπως βλέπουμε είναι ένα bash script το οποίο περιμένει να αντιληφθεί ότι έχουν ήδη σηκωθεί τα services Eureka Server, Database και Config Server πριν εκτελέσει το γνωστό μας java -jar. Το Renting service χρειάζεται αυτά τα services να έχουν σηκωθεί πριν μπορέσει να λειτουργήσει και επομένως πρέπει να χρησιμοποιήσουμε αυτό το 'τρικ' για να βεβαιωθούμε για τη σωστή σειρά με την οποία θα τρέξουν τα services.

Στο script βλέπουμε και κάποια varibles όπως $EUREKASERVER_PORT, $DATABASESERVER_PORT, $CONFIGSERVER_PORT . Αυτά είναι εξωτερικές μεταβλητές τις οποίες θα δούμε στη φάση του orchestration πώς τις ορίζουμε.

8.2.2 Maven plugin

Για να κάνουμε trigger το dockerfile χρησιμοποιούμε ένα maven plugin όπως είπαμε. Ας δούμε πώς ακριβώς το κάνουμε αυτό. Στο pom.xml αρχείο κάνουμε import to Spotify docker-maven-plugin και του δίνουμε το κατάλληλο configuration:

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.10</version>
    <configuration>
        <imageName>${docker.image.name}:${docker.image.tag}</imageName>
        <dockerDirectory>${basedir}/target/dockerfile</dockerDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

Οι μεταβλητές που χρησιμοποιούμε δίνονται σε άλλο τμήμα του pom.xml ώστε να μην έχουμε duplication του configuration.

Σε αυτό το σημείο για να κάνουμε trigger το compilation και το packaging σε docker containers το μόνο που χρειάζεται να κάνουμε είναι:

mvn clean package docker:build

Το maven θα κάνει το compilation και ύστερα το plugin θα δημιουργήσει τα containers σύμφωνα με τα dockefiles.

8.2.3 Docker registry

Στη φάση αυτή έχουμε τα containers στον υπολογιστή που έγινε το compilation, αλλά το deployment θα το κάνουμε σε άλλους servers. Θα μπορούσαμε να κάνουμε copy τα containters αλλά υπάρχει και πιο εύκολος τρόπος: το docker registry.

Το docker registry είναι ένα stateless, highly scalable server side application που δίνει τη δυνατότητα να κάνουμε distribute docker images. Το registry είναι open source με Apache license. Κάποια εταιρία/οργανισμός έχει τις εξής επιλογές όσον αφορά το docker registry:

- να κάνει host το δικό του docker registry
- να χρησιμοποιήσει κάποιο paid plan στο Docker Hub
- να χρησιμοποιήσει κάποιο free plan στο Docker Hub (μόνο για public repositories)

Για την ανάπτυξη της εφαρμογής μας χρησιμοποιήθηκε το η 3η επιλογή και τα αντίστοιχα docker images βρίσκονται εδώ

8.2.4 Docker compose

Έχοντας δημιουργήσει τα docker containers και έχοντας τα στο docker registry (ή στο development machine) το μόνο που μένει είναι το τελικό orchestration. Η τελική σύνδεση των containers ώστε να έχουμε τη συνολική εφαρμογή μας. Ένας από τους τρόπους που μπορεί να γίνει αυτό και αυτός που επιλέχθηκε για την εφαρμογή μας είναι το **docker compose**.

Το Compose είναι ένα εργαλείο για τον καθορισμό και την εκτέλεση εφαρμογών Docker πολλαπλών κοντέινερ. Με το Compose, ο developer χρησιμοποιεί ένα αρχείο YAML για να διαμορφώσει τα services της εφαρμογής. Στη συνέχεια, με μία μόνο εντολή, δημιουργεί και ξεκινά όλα τα services από το configuration.

Η χρήση του Compose είναι βασικά μια διαδικασία τριών βημάτων:

- Ορισμός του περιβάλλοντος της εφαρμογής με ένα αρχείο Docker, ώστε να μπορεί να αναπαραχθεί οπουδήποτε
- Καθορισμός των υπηρεσιών που συνθέτουν την εφαρμογή στο docker-compose.yml έτσι ώστε να μπορούν να λειτουργούν μαζί σε ένα απομονωμένο περιβάλλον.
- Εκτέλεση της εντολής docker-compose υρκαι το Compose εκκινεί και εκτελεί ολόκληρη την εφαρμογή

Ας αναλύσουμε τώρα το αρχείο docker-compose.yml της εφαρμογής μας που βρίσκεται στο path docker/common

```
version: '2'
services:
  eurekaserver:
    image:   spirosag/eurekaserver:latest
    ports:
      - "8761:8761"
  configserver:
    image: spirosag/configserver:latest
    ports:
      - "8888:8888"
    environment:
      EUREKASERVER_URI: "http://eurekaserver:8761/eureka/"
      EUREKASERVER_PORT: "8761"
      ENCRYPT_KEY:       "IMSYMMETRIC"
  database:
    image: postgres:9.5
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=p0stgr@s
      - POSTGRES_DB=rent_flix_local
```

```
rentingservice:
    image: spirosag/renting-service:latest
    ports:
      - "8080:8080"
      - "8090:8090"
    environment:
      PROFILE: "default"
      SERVER_PORT: "8080"
      CONFIGSERVER_URI: "http://configserver:8888"
      EUREKASERVER_URI: "http://eurekaserver:8761/eureka/"
      CONFIGSERVER_PORT:   "8888"
      DATABASESERVER_PORT: "5432"
      EUREKASERVER_PORT:   "8761"
      ENCRYPT_KEY:       "IMSYMMETRIC"
movieservice:
    image: spirosag/movie-service:latest
    ports:
      - "8085:8085"
    environment:
      PROFILE: "default"
      SERVER_PORT: "8085"
      CONFIGSERVER_URI: "http://configserver:8888"
      EUREKASERVER_URI: "http://eurekaserver:8761/eureka/"
      CONFIGSERVER_PORT:   "8888"
      DATABASESERVER_PORT: "5432"
      EUREKASERVER_PORT:   "8761"
      ENCRYPT_KEY:       "IMSYMMETRIC"
```

Αρχικά ορίζουμε το version του docker compose που θα χρησιμοποιηθεί για να τρέξει η εφαρμογή. Στη συνέχεια ορίζουμε τα services που θα τρέξουν. Αντιστοιχεί ένα microservice με ένα docker container και ένα docker services. Στην εφαρμογή μας έχουμε 5 microservies και άρα 5 docker services:

1. Eureka Server
2. Config Server
3. Database
4. Movie Service
5. Renting Service

Για τον καθορισμό κάθε services χρησιμοποιούμε τα εξής configuration properties:

- services.[serviceName]: το όνομα με το οποίο θα κάνει reference το docker αυτό το service
  - image: Το docker image που θα τρέξει για να δημιουργήσει το docker container. (αυτά είναι τα containers που έχουμε ανεβάσει στο registry ή έχουμε τοπικά στο δίσκο μας)

- ports: Κάνουμε το απαραίτητο port forwarding καθώς by default όλες οι πόρτες του container είναι κλειστές. Ο καθορισμός γίνεται με το συντακτικό HOST:CONTAINER. Δηλαδή το 5000:8090 σημαίνει ότι συνδέουμε την πόρτα 5000 του host με την πόρτα 8090 του container
- environment: Θέτουμε environment variables που θέλουμε να περάσουμε στο συγκεκριμένο container. Αυτά τα variables μπορούν να γίνουν reference μέσα στο container (πχ στο run.sh αρχείο όπως είδαμε ήδη)

Έχοντας δημιουργήσει το πλήρες docker-compose.yml αρχείο μας το μόνο που μας μένει για να "σηκώσουμε" όλα τα microservices είναι να τρέξουμε την εντολή:

docker-compose -f docker/common/docker-compose.yml up

Αντίστοιχα για να σταματήσουμε τα containers:

docker-compose -f docker/common/docker-compose.yml down

Συμπέρασμα

Τα microservices είναι μία αρχιτεκτονική που ήρθε για να μείνει. Ολοένα και περισσότερες εταιρίες αρχίζουν να αντιμετωπίζουν τα συστήματα που αναπτύσσουν όχι σαν μία ενιαία μονολιθική εφαρμογή αλλά σαν ένα σύνολο από μικρότερα services. Αυτό τα κάνει πιο ευέλικτα στις αλλαγές αλλά και πιο δύσκολα στην ανάπτυξη.

Ο developer που θα αναλάβει να αναπτύξει microservices χρειάζεται να αντιμετωπίσει προβλήματα που σε μια παραδοσιακή μονολιθική εφαρμογή δεν υπάρχουν: Το πρόβλημα του service discovery, πώς δηλαδή θα «ανακαλύπτει» το ένα service τα υπόλοιπα. Το πρόβλημα του configuration, πώς δηλαδή θα αλλάζει το configuration σε μεγάλο πλήθος microservices αφού έχουν γίνει deploy. Επίσης πώς θα επικοινωνεί εύκολα με τα υπόλοιπα services.

Μαζί με αυτά τα προβλήματα θα πρέπει να αρχίσει να αντιλαμβάνεται και να χρησιμοποιεί patterns που στη μονολιθική εφαρμογή δεν υπάρχουν. Πράγματα που τα θεωρούσε δεδομένα (διευθύνσεις, instances, χρόνος απόκρισης) πλέον είναι ευμετάβλητα. Κάθε φορά που χρειάζεται να επικοινωνήσει με κάποιο άλλο microservice χρειάζεται να το αντιμετωπίζει σαν call σε κάποιο 3rd party service σε κάποιο άλλο δίκτυο για το οποίο δεν γνωρίζει σχεδόν τίποτα.

Σε όλες αυτές τις αλλαγές και τις δυσκολίες έρχεται και ένα νέο σύστημα deployment καθώς χρειάζεται να διαχειριστούμε και να κάνουμε deploy πολλές εφαρμογές ταυτόχρονα.

Σκοπός αυτής της εφαρμογής ήταν να παρουσιάσει βασικούς τρόπους να αντιμετωπιστούν αυτά τα προβλήματα. Να δώσει στον developer εργαλεία για εύκολο service discovery. Να δείξει πώς μπορεί να κάνει το configuration externalized. Να προσφέρει τρόπους για service communication χρησιμοποιώντας resilience design patterns με κατάλληλες βιβλιοθήκες. Και τέλος να δείξει πώς μπορεί ένα containerized deployment σύστημα να ενσωματωθεί με το development.

Τα microservices είναι μία αρχιτεκτονική που ήρθε για να μείνει. Έχει δυσκολίες αλλά ταυτόχρονα προσφέρει ευελιξία που ποτέ πριν δεν ήταν εφικτή. Αυτό που χρειάζεται ο developer είναι να εφοδιαστεί με τις κατάλληλες γνώσεις, εργαλεία και μεθοδολογίες για να τα κατακτήσει.

Βιβλιογραφία

- https://spring.io/microservices
- https://docs.docker.com/
- Building Microservices - Sam Newman
- Microservices Patterns - Chris Richardson
- Spring Microservices in Action - John Carnell
- https://microservices.io/patterns/microservices.html
- https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html
- https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency
- https://github.com/Netflix/eureka
- https://github.com/Netflix/Hystrix
- https://spring.io/projects/spring-cloud-config
- https://www.postgresql.org/docs/
- https://hibernate.org/orm/documentation/5.4/