ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Back-end Web εφαρμογή για Παρακολούθηση Τιμών Ξενοδοχείων»

Φοιτητής: Καλαιτσιδης Αρσλαν.

Επιβλέπων: Σαλαμπάσης Μιχάλης.

26/01/2024

Τίτλος Δ.Ε. Back-end WEB εφαρμογή για Παρακολούθηση Τιμών Ξενοδοχείων
Κωδικός Δ.Ε. 22340
Ονοματεπώνυμο φοιτητή Καλαιτσιδης Αρσλαν
Ονοματεπώνυμο εισηγητή …
Ημερομηνία ανάληψης Δ.Ε. 09-11-2022
Ημερομηνία περάτωσης Δ.Ε. …

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Καλαιτσιδης Αρσλαν που την εκπόνησε/αν. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Περίληψη

Το αντικείμενο αυτής της πτυχιακής εργασίας ήταν η ανάπτυξη web εφαρμογής για την παρακολούθηση των τιμών ξενοδοχείων/δωματίων από διαχειριστές ξενοδοχείων για να μπορούν παίρνουν τεκμηριωμένες απόφασης και να έχουν καλύτερη εικόνα του τομέα μέσα στον οποίο εργάζονται.

Η εφαρμογή αναπτύχθηκε χρησιμοποιώντας βιβλιοθήκες, εργαλεία και γλώσσες του JVM οικοσυστήματος και σε συνεργασία με άλλες ομάδες που ασχολήθηκαν με το κομμάτι του scraping και front-end.

«Back-end WEB API for hotel price monitoring»

«Kalaitsidis Arslan»

Abstract

The purpose of this project was the development of web application to allow hotel managers monitor prices for hotel rooms so that they can take informed decisions and have a better view of the sector they are working in.

The application was developed using libraries, tools and languages of JVM ecosystem and in cooperation with other teams that implemented the front-end and scraping parts of the system.

Περιεχόμενο

Περίληψη......3
Abstract......4
Συντομογραφίες......10
1 Εισαγωγή......11
1.1 Προσδιορισμός Τιμών......11
1.2 Λίστες......11
1.3 Το Σύστημα......11
1.4 Η κατάσταση των συστημάτων παρακολούθησης ξενοδοχείων......12
1.5 Κανόνες Ανάπτυξης......12
1.6 Οργάνωση......12
2 Τεχνολογίες......13
2.1 Kotlin......13
2.2 Gradle......15
2.2.1 Build.gradle.kts......15
2.2.2 Gradle Tasks......15
2.2.3 Gradle Dependencies......15
2.3 Gradle Plugins......16
2.3.1 Gradle Build Lifecycle......18
2.3.2 Gradle Incremental Builds......18
2.4 Gradle Wrapper......18
2.5 Docker......19
2.5.1 Dockerfile......19
2.5.2 Docker Registry......21
2.5.3 Docker Repository......21
2.6 Servlets......21
2.7 Spring......21
2.7.1 Spring Framework......21
2.7.1.1 *Spring* IoC......21
2.7.1.2 Spring Bean Configuration......22
2.7.1.3 Spring Component Scanning......24
2.7.1.4 Environment......24
2.7.1.4.1 Ιδιότητες......24
2.7.1.4.2 Profiles......25
2.7.1.5 Spring Transaction Management......25

2.7.2 Spring Boot....28
2.7.2.1 Spring Boot Properties....28
2.7.2.2 Spring Boot Conditional Bean Registration....28
2.7.3 Spring MVC....28
2.7.3.1 Spring MVC Handlers....29
2.7.3.2 Spring MVC εξαιρέσεις....30
2.7.4 Spring Hateoas....30
2.7.5 Spring Security....32
2.7.5.1 Authentication....32
2.7.5.2 Authentication Manager....32
2.7.5.3 SecurityContext....32
2.7.5.4 SecurityContextHolder....32
2.7.5.5 GrantedAuthority....32
2.7.5.6 AuthorizationManager....32
2.7.5.7 Spring MVC Integration....32
2.7.6 Spring AMQP....34
2.7.7 Spring Test....34
2.7.8 Spring Data JPA....34
2.7.9 Spring Validation....35
2.8 Hibernate....35
2.8.1 Hibernate ORM....35
2.8.1.1 Entity....35
2.8.1.2 Persistent Context....37
2.8.1.3 HQL/JPQL....39
2.8.1.4 Hibernate Envers....41
2.8.2 Hibernate Validation....41
2.9 Liquibase....43
2.9.1 Spring Boot Integration....44
2.9.2 Context....44
2.10 Bucket4j....44
2.10.1 Bucket....44
2.10.2 Bucket Configuration....45
2.10.3 Περιορισμοί....45
2.10.4 Στρατηγικές παραγωγής των token....45
2.10.5 JDBC Integration....45

2.11 Testing....45
2.11.1 Spring TestContext Framework....46
2.11.2 Spring Boot Test....47
2.11.3 TestContainers....47
2.11.4 Unit Testing....48
2.11.4.1 Mockk και Springmockk....49
2.11.4.2 JUnit....50
2.12 Άλλα....51
3 Υλοποίηση....52
3.1 Αρχιτεκτονική....52
3.2 Roles....52
3.3 Email....52
3.4 JWT....52
3.4.1 JWS Αυθεντικοποίηση και Εξουσιοδότηση....53
3.4.2 JWS Email Verification....53
3.4.3 JWS Password Reset....54
3.5 Rate Limiting....54
3.6 Επίπεδο Παρουσίασης....54
3.7 Επίπεδο Εφαρμογής....55
3.7.1 Εγγραφή του χρήστη στο σύστημα....55
3.7.2 Αυθεντικοποίηση....56
3.7.3 Δημιουργία Λίστας Δωματίων....56
3.7.4 Τιμές Δωματίων....56
3.7.5 Ιστορικό Ιδιοτήτων Δωματίων....56
3.7.6 Συνδρομές....56
3.8 Επίπεδο Δεδομένων....56
3.8.1 Χρήστης....57
3.8.2 Ξενοδοχείο....57
3.8.3 Δωμάτιο....57
3.8.4 Επιλογή Δωματίου....57
3.8.5 Ιδιότητα Δωματίου και Ιδιότητα Επιλογής....58
3.8.6 Λίστα Δωματίων....58
3.8.7 Τιμές....59
4 Συμπεράσματα....60

Κατάλογος Σχημάτων

Figure 1: Extension Function Bytecode....13
Figure 2: Byte Code της class που δεν απαιτείται από την Spring βιβλιοθήκη να είναι non-final....17
Figure 3: Byte code της class που χρησιμοποιείται από Spring βιβλιοθήκη....17
Figure 4: Class που χρησιμοποιεί @Serializable annotation....18
Figure 5: To byte code της class που χρησιμοποιεί @Serializable annotation για την οποία έχει δημιουργηθεί kotlinx serializer....18
Figure 6: Η λειτουργία του Spring Framework....23
Figure 7: Spring @Configuration annotation definition....26
Figure 8: Λειτουργία των Spring Proxy για διαχείριση συναλλαγών....28
Figure 9: Persistent Context Cache Miss....43
Figure 10: Persistent Context Cache Hit....43
Figure 11: Dirty Check Example....44
Figure 12: Lazy Collection Loading Example....44
Figure 13: JOIN χωρίς FETCH παράδειγμα....46
Figure 14: JOIN με FETCH παράδειγμα....46
Figure 15: To default fetch χωρίς JPQL/HQL....46
Figure 16: Παράδειγμα JWS που επιστρέφει η εφαρμογή....64

Κατάλογος Πινάκων

Table 1: Αποτελέσματα της docker build εντολής με και χωρίς caching..22
Table 2: Αποτελέσματα docker build εντολής με και χωρίς multi-staging..22

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία
IP	Internet Protocol
API	Application Programming Interface
JWT	JSON Web Token
JWS	JSON Web Signature
XML	Extensible Markup Language
JAR	Java ARchive
HATEOAS	Hypermedia as the engine of application state
AMQP	Advanced Message Queuing Protocol
JPA	Java Persistence API
ORM	Object-Relational Mapping
RDBMS	Relational Database Management System
SQL	Structured Query Language
MVC	Model-View-Controller
HTML	HyperText Markup Language
JDBC	Java Database Connectivity
HE	Hibernate Entity
JPQL	Jakarta Persistence Query Language
HQL	Hibernate Query Language
IoC	Inversion Of Control
JVM	Java Virtual Machine
ΚΑΓ	Κατευθυνόμενος Άκυκλος Γράφος
HATEOAS	Hypermedia as the engine of application state
CSRF	Cross Site Request Forgery
DTO	Data Transfer Objects
HMAC	Hashed Message Authentication Code
OTP	One Time Password
DoS	Denial of Service
DDoS	Distributed Denial of Service

1 Εισαγωγή

Στόχος της πτυχιακής εργασία ήταν η δημιουργία εργαλείου το οποίο αυτόματα συλλέγει και αναλύει δεδομένα ξενοδοχείων/δωματίων για τους διαχειριστές ξενοδοχείων προκειμένου να λαμβάνουν τεκμηριωμένες αποφάσεις. Ένα τέτοιο εργαλείο θα επιτρέπει στους επιχειρηματίες να παραμένουν ανταγωνιστικοί στον τομέα τους και να προσαρμόζονται στις αλλαγές του περιβάλλοντος μέσα στο οποίο δραστηριοποιούνται.

Η συλλογή και ανάλυση δεδομένων αφορά πρώτα από όλα τις τιμές ξενοδοχείων. Τέτοια δεδομένα και τα συμπεράσματα τα οποία κάποιος μπορεί να εξάγει από αυτά επιτρέπουν τους διαχειριστές ξενοδοχείων γρήγορα και αποτελεσματικά να αναλαμβάνουν ενέργειες προκειμένου να προσαρμόζουν τις τιμές τους βάση της γενικής εικόνας που υπάρχει ή ενδέχεται να υπάρχει και έτσι να παραμένουν ανταγωνιστική.

1.1 Προσδιορισμός Τιμών

Ο σωστός προσδιορισμός τιμής είναι διαδικασία η οποία επηρεάζεται από πολλές μεταβλητές. Δύο από αυτές τις μεταβλητές είναι η κατάσταση της αγοράς και οι τιμές των ανταγωνιστών. Το σύστημα παρακολούθησης τιμών επιτρέπει την καλύτερη κατανόηση αυτών των μεταβλητών στον τρέχοντα χρόνο, αλλά και εντοπισμό μοτίβων μεταβολής τους τα οποία επιτρέπουν την πρόβλεψη αυτών των τιμών με κάποια ακρίβεια.

1.2 Λίστες

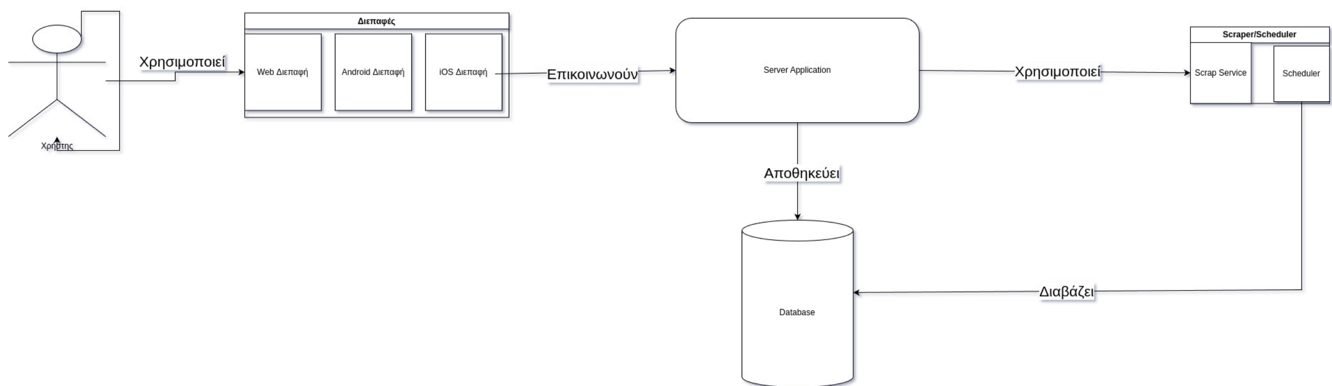
Προκειμένου η συλλογή και ανάλυση των δεδομένων να είναι χρήσιμη, θα πρέπει το σύστημα να επιτρέπει τον προσδιορισμό των δωματίων ξενοδοχείων για τα οποία θα συλλέγονται τα δεδομένα προκειμένου τα δεδομένα δωματίων για τα οποία οι διαχειριστές δεν έχουν ενδιαφέρον να μην επηρεάζουν τα αποτελέσματα.

Για αυτό το λόγο μία από τις βασικές λειτουργίες που προσφέρει το σύστημα είναι η δημιουργία λιστών δωματίων για τα οποία συλλέγονται δεδομένα. Έτσι,για παράδειγμα, ένας διαχειριστής μπορεί να δημιουργήσει λίστα παρακολούθησης δωματίων των ανταγωνιστών που έχουν ίδια ή παρόμοια χαρακτηριστικά με τα δωμάτια ξενοδοχείων που προσφέρει ή να δημιουργήσει λίστα δωματίων που έχουν επιπλέον υπηρεσίες για να δει αν είναι κερδοφόρο να προσφέρονται αυτές οι υπηρεσίες.

1.3 Το Σύστημα

Το σύστημα που αναπτύχθηκε ήταν χωρισμένο σε τρία κομμάτια τα οποία αναπτύχθηκαν σε συνεργασία προκείμενου να μπορούν να να προσφερθούν ως τελικό προϊόν. Τα τρία αυτά κομμάτια είναι:

- Διεπαφή Χρήστη
- Server εφαρμογή
- Scraper και scheduler



Το κομμάτι με το οποίο ασχολήθηκα ήταν η ανάπτυξη της server εφαρμογής και του σχήματος της βάσης δεδομένων.

1.4 Η κατάσταση των συστημάτων παρακολούθησης ξενοδοχείων

Υπάρχουν διάφορες εφαρμογές οι οποίες προσφέρουν παρακολούθηση και σύγκριση τιμών, αλλά οι δυνατότητες που προσφέρουν είναι περιορισμένες ή δεν μπορούν να χρησιμοποιηθούν για ιστοσελιδές ξενοδοχείων. Επίσης, τα περισσότερα από αυτά τα συστήματα παρακολούθησης τιμών ξενοδοχείων που υπάρχουν στην αγορά προορίζονται κυρίως σε καταναλωτές και όχι διαχειριστείς, και προσφέρουν κυρίως τη λειτουργία σύγκρισης τιμών. Άλλα προϊόντα προσφέρουν γενική παρακολούθηση τιμών με notifications και συχνά δεν δουλεύουν με τις ιστοσελίδες των ξενοδοχείων καθώς αυτές ακολουθούν διαφορετική μορφή από τις ιστοσελίδες προϊόντων όπου για την εμφάνιση των τιμών ο χρήστης πρέπει να επιλέξει ημερομηνίες.

Τέλος, πολλά εργαλεία απαιτούν χειροκίνητη εισαγωγή δεδομένων χρησιμοποιώντας αρχεία.

1.5 Κανόνες Ανάπτυξης

Για την καλύτερη ανάπτυξη της εφαρμογής και για να την αποφυγή του Over-Design του συστήματος η ανάπτυξη πραγματοποιούνταν με στόχο να προσφέρει βασικές και απλές λειτουργίες. Στόχος δηλαδή δεν ήταν η ανάπτυξη περίπλοκης εφαρμογής που θα έλυνε ένα συγκεκριμένο πρόβλημα καθώς αυτό θα οδηγούσε σε δημιουργία εφαρμογής η οποία θα ήταν δύσκολη σε κατανόηση και δύσκολο να υφίσταται αλλαγές· αλλά στόχος ήταν η δημιουργία εφαρμογής η οποία θα ήταν εύκολη στην κατανόηση,και πάνω στην οποία εύκολα θα μπορούσαν να χτιστούν άλλες εφαρμογές ή η οποία εύκολα θα μπορούσε περαιτέρω να αλλαχθεί/αναπτυχθεί για να την λύση πιο συγκεκριμένων προβλημάτων.

1.6 Οργάνωση

Το ακόλουθο κείμενο της διπλωματικής εργασίας είναι χωρισμένο σε δύο μέρη.

Το πρώτο μέρος παρουσιάζει τις τεχνολογίες και τις βιβλιοθήκες που χρησιμοποιήθηκαν κατά την υλοποίηση της εφαρμογής με εξηγήσεις και παραδείγματα από την εργασία. Στόχος του πρώτου μέρους είναι η παρουσίαση των τεχνολογιών ώστε ο αναγνώστης που έχει κάποιες βασικές γνώσεις προγραμματισμού να μπορεί να κατανοήσει τον κώδικα και τα αρχεία της εργασίας.

Το δεύτερο μέρος είναι ανάλυση του σχήματος της βάσης δεδομένων και ορισμένων λεπτομερειών της υλοποίησης της εφαρμογής.

2 Τεχνολογίες

2.1 Kotlin

Η γλώσσα προγραμματισμού που χρησιμοποιήθηκε για την υλοποίηση της εφαρμογής είναι η Kotlin.

Η Kotlin είναι γλώσσα προγραμματισμού η πρώτη έκδοση της οποίας βγήκε το 2011. Η Kotlin είναι διαλειτουργικη(interoperable) με την Java το οποίο επιτρέπει στους προγραμματιστές να χρησιμοποιούν τις βιβλιοθήκες της Java μέσα στον κώδικα της Kotlin και το αντίστροφο.

Η Kotlin έχει δυνατότητες που δεν υπάρχουν στην Java[1].Κάποιες από αυτές είναι

- Extensions(επεκτάσεις) δίνουν δυνατότητα επέκτασης μιας class ή interface με καινούριες μεθόδους ή ιδιότητες χωρίς να απαιτείται αλλαγή του κώδικα της class ή η προγραμματιστική επέκταση μέσω κληρονομικότητας/design patterns. Αυτό επιτρέπει τη δημιουργία επεκτάσεων – επιπλέον μεθόδων για classes/interfaces που είτε είναι final(δεν επιτρέπεται κληρονομικότητα) είτε είναι εκτός ελέγχου του προγραμματιστή, δηλαδή ο κώδικας της class διατηρείται από άλλους ανθρώπους(όπως η java.lang.String που ανήκει στο JDK).

  Μέσα στην επέκταση επιτρέπεται αναφορά στο αντικείμενο πάνω στο οποίο κλήθηκε η επέκταση χρησιμοποιώντας το keyword this ή χρησιμοποιώντας μόνο τα ονόματα ιδιοτήτων και μεθόδων: σε αυτές τις περιπτώσεις, το this υπονοείται.

  Στην εργασία οι επεκτάσεις χρησιμοποιήθηκαν κυρίως για δημιουργία utility συναρτήσεων.

  Ακολουθεί ένα παράδειγμα από την εργασία όπου καλείται η επέκταση withoutQueryPart() πάνω στο αντικείμενο τύπου String(η link ιδιότητα έχει τύπο String) και ο ορισμός της επέκτασης.

```
val actualURL = hotelInfoRequest.link.withoutQueryPart()
fun String.withoutQueryPart() : String = substring(0,indexOfOrFull("?"))
```

  Για την υλοποίηση των επεκτάσεων η Kotlin μετατρέπει την συνάρτηση(όπως φαίνεται στο Figure 1: Extension Function Bytecode) σε static μέθοδο με μία επιπλέον παράμετρο που είναι το αντικείμενο της class για την οποία δημιουργείται η επέκταση και πάνω στο οποίο κλήθηκε η συνάρτηση.

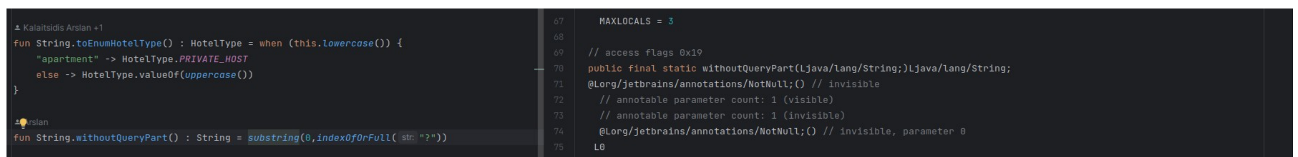


Figure 1: Extension Function Bytecode

- Null Safety: το σύστημα τύπων(type system) της Kotlin είναι πιο εκφραστικό από αυτό της Java καθώς επιτρέπει τη διάκριση μεταξύ τύπων που μπορούν να παίρνουν null τιμή και αυτών που δεν μπορούν να παίρνουν null τιμή. Αυτό με τη σειρά του επιτρέπει να αποφεύγονται τα NullPointerExceptions κατά το code compilation και την εμφάνιση τους σε Runtime.

  Για την δήλωση του τύπου μεταβλητής ή ιδιότητας που μπορεί να πάρει null τιμή χρησιμοποιείται το ερωτηματικό μετά το όνομα του τύπου. Για παράδειγμα, η μεταβλητή με τύπο String? μπορεί να πάρει null τιμή, ενώ String – όχι.

  Ακολουθεί παράδειγμα από την εργασία όπου η ιδιότητα cookies επιστρέφει nullable τύπο(τύπο που δηλώνει ότι η τιμή μπορεί να είναι null).

---

1 Σε σύγκριση με την εκδοσή Java 8/11.

Επίσης, στο ίδιο παράδειγμα χρησιμοποιείται ο safe call operator(?.) της Kotlin ο οποίος επιτρέπει να γίνεται πρόσβαση σε συναρτήσεις ή ιδιότητες αντικειμένου που μπορεί να είναι null. Συγκεκριμένα, αν η τιμή της μεταβλητής/ιδιότητας είναι null, τότε οποιαδήποτε πρόσβαση στις ιδιότητες του αντικειμένου χρησιμοποιώντας τον safe call operator επιστρέφει null χωρίς να πετάει NullPointerException.

```
request.cookies?.find{ it.name == AUTHORIZATION_COOKIE_NAME }?.value ?:
defaultBearerTokenResolver.resolve(request)
```

- Smart Casts: ο compiler της Kotlin είναι αρκετά "έξυπνος" για να ξέρει πότε το cast είναι περιττό.

  Στο παρακάτω παράδειγμα φαίνεται η λειτουργία του smart cast από nullable τύπο CustomerRegistration? σε non-nullable τύπο CustomerRegistration με αποτέλεσμα να μην χρειάζεται η χρήση του safe call operator ή cast μέσα στον κώδικα.

```
override fun isValid(value: CustomerRegistration?, context: ConstraintValidatorContext?): Boolean
{
        if(value == null) return true //after if Kotlin figures that value is not null because if
returns

        return value.password == value.repeatedPassword
}
```

- Properties(ιδιότητες). Παρά την εισαγωγή των Java Records στην έκδοση 14, η Java δεν υποστηρίζει την έννοια των ιδιοτήτων με αποτέλεσμα οι προγραμματιστές να αναγκάζονται είτε να χρησιμοποιούν πρόσθετα εργαλεία(Project Lombok) που δημιουργούν getters/setters είτε να δημιουργούν getters και setters.

  Με την εισαγωγή των Records η Java υποστηρίζει immutable ιδιότητες, αλλά δεν έχει υποστήριξή για γενικές ιδιότητες που μπορούν να είναι immutable ή mutable.

  Η Kotlin υποστηρίζει την έννοια των ιδιοτήτων. Οι ιδιότητες μπορούν να οριστούν μέσα από constructor ή στο σώμα της class.

  Η ιδιότητα μπορεί να οριστεί ως val ή var, όπου val σημαίνει immutable ιδιότητα και var mutable ιδιότητα.

```
@Serializable
data class HotelInfo(
   val name: String,
   val description: String,
   val location: Location,
   val url: String,
   val hotelType: HotelType,
   @Serializable(with = BigDecimalSerializer::class)
   val score: BigDecimal,
   val hotelID: Long,
   val rooms: Set<RoomInfo>
)
```

- Κενά στα ονόματα μεθόδων. Η Kotlin επιτρέπει τη χρήση κενού χαρακτήρα(space) μέσα στο όνομα της μεθόδου όταν η μέθοδος χρησιμοποιείται στο testing. Το όνομα πρέπει να βρίσκεται μέσα σε `` (backticks). Αυτονόητα ονόματα μεθόδων είναι σημαντικό χαρακτηριστικό των τεστ για να μπορεί κάποιος να καταλάβει το σενάριο το οποίο ελέγχεται.

```
class CustomerControllerUnitTest : AbstractControllerUnitTest() {
```

```
    @Test
    fun `should return 200(OK) response after successful customer registration`() {
        every { customerServiceMock.register(customerRegistration) } returns mockk()
    }
}
```

- Infix συναρτήσεις. Η Kotlin επιτρέπει τη χρήση infix keyword για συναρτήσεις που δέχονται μόνο μία παράμετρο. Για αυτές τις συναρτήσεις η Kotlin επιτρέπει την κλήση τους χωρίς τον dot(.) operator.

  Στο παραπάνω παράδειγμα η μέθοδος `every{}` επιστρέφει αντικείμενο το οποίο έχει infix μέθοδο returns, οπότε η Kotlin επιτρέπει η μέθοδος να κληθεί χωρίς dot operator.

  Κανονικά, αν δεν υπήρχε υποστήριξη για infix συναρτήσεις, η συνάρτηση έπρεπε να κληθεί με τον εξής τρόπο: `every { customerServiceMock.register(customerRegistration) }.returns(mockk())`. Αν και αυτό επιτρέπεται, η κλήση χρησιμοποιώντας infix notation είναι πιο ευανάγνωστη.

- Lateinit ιδιότητες. Όταν μια ιδιότητα έχει non-nullable τύπο,η τιμή της ιδιότητα πρέπει να οριστεί είτε μέσα από τον constructor της class είτε στο σώμα της class. Αυτό μπορεί να είναι ακατάλληλο για ορισμένες περιπτώσεις όπως όταν οι τιμές για αυτές τις ιδιότητες προσφέρονται σε runtime,για παράδειγμα, χρησιμοποιώντας dependency injection.

  Μία λύση είναι να χρησιμοποιήσουμε nullable τύπο και να εισάγουμε ως αρχική τιμή null. Αυτή η λύση όμως αναγκάζει τη χρήση του safe call operator εκεί όπου δεν χρειάζεται.

  Για αυτές τις περιπτώσεις η Kotlin επιτρέπει τη χρήση του lateinit keyword για var ιδιότητες με non-nullable τύπο.

  Lateinit είναι ιδιότητες οι τιμές των οποίων αρχικοποιούνται μετά τη δημιουργία του αντικειμένου. Η χρήση της ιδιότητας πριν γίνει αρχικοποίηση οδηγεί σε εξαίρεση.

  Στην εργασία lateinit μεταβλητές χρησιμοποιήθηκαν στα τεστ για ιδιότητες οι οποίο θέτονται από Spring dependency injection.

```
abstract class AbstractITest : AbstractTest(){
    @Autowired
    protected lateinit var messageSource: MessageSource
}
```

  Όπως φαίνεται και από τα παραδείγματα, οι δυνατότητες που προσφέρει η Kotlin βρίσκονται κυρίως σε επίπεδο του code compilation καθώς η Kotlin δεν μπορεί να προσθέσει καινούριες δυνατότητες στην JVM και είναι περιορισμένη από αυτά που επιτρέπει η JVM.

2.2 Gradle

Το gradle είναι build tool η κύρια αρμοδιότητα του οποίου είναι το compilation του κώδικα και πακετάρισμα της εφαρμογής ή, με άλλα λόγια, το build της εφαρμογής.

Το project που χρησιμοποιεί gradle ως build tool περιέχει τα παρακάτω στοιχεία [1]:

1. settings.gradle.kts αρχείο το οποίο δηλώνει το root project και τα sub-projects.
2. Root Project.
3. Ένα ή περισσότερα sub-projects.

Κάθε sub-project περιέχει build.gradle.kts αρχείο το οποίο ορίζει τις ενέργειες που απαιτούνται για να γίνει build το project[2].

Στην εργασία μου χρησιμοποίησα root project χωρίς sub-projects.

2.2.1 Build.gradle.kts

Το build.gradle.kts είναι το αρχείο το οποίο διαβάζει το gradle και το οποίο περιέχει:

- Ορισμό των dependencies – τα dependencies τα οποία χρησιμοποιούνται στο project.
- Plugins. Προσφέρουν tasks που υλοποιούν τις βασικές λειτουργίες όπως compilation,εκτέλεση των τεστ κτλ.
- Ορισμός των repositories από τα οποία το gradle κατεβάζει τα dependencies/plugins.
- Configuration των tasks που προσφέρονται από τα plugins.

2.2.2 Gradle Tasks

Gradle task είναι βασική μονάδα εκτέλεσης στο gradle[3].

Η εκτέλεση του task μπορεί να βασίζεται στην επιτυχής ολοκλήρωση άλλων tasks. Για παράδειγμα, το jar task που δημιουργεί το jar αρχείο του project βασίζεται στο compilation task που κάνει compilation του κώδικα.

Έτσι, όταν εκτελούμε κάποιο task, πριν την εκτέλεση του, τρέχουν το/τα task/s τα οποία έχουν δηλωθεί ως tasks από τα οποία εξαρτάται το task που τρέχουμε κ.ο.κ.

2.2.3 Gradle Dependencies

Τα dependencies της εφαρμογής ορίζονται στο build.gradle.kts αρχείο και έχουν τη μορφή group:name:version.

Επίσης, για κάθε dependency ορίζεται το επίπεδο στο οποίο χρειάζεται. Για παράδειγμα, testImplementation (group:name:version) ορίζει ότι το dependency χρειάζεται μόνο για την εκτέλεση των τεστ.

Εκτός από τα dependencies που ορίζονται μέσα στο build.gradle.kts, το gradle μπορεί να κατεβάσει επιπλέον dependencies στα οποία βασίζονται τα dependencies της εφαρμογής(transitive dependencies) κ.ο.κ. Για παράδειγμα, το spring-boot-starter-validation που χρησιμοποιείται στην εργασία έχει dependency στην βιβλιοθήκη hibernate validation.

2.3 Gradle Plugins

Τα plugins είναι η βασική μονάδα οργάνωσης και εφαρμογής build λογικής,και αποτελούν την κύρια πηγή των gradle tasks.

Στην εργασία χρησιμοποιήθηκαν τα παρακάτω gradle plugins:

- Spring Boot προσφέρει plugins τα οποία περιέχουν spring boot tasks και dependency management.

  Spring Boot dependency management plugin διαχειρίζεται τις εκδόσεις των βιβλιοθηκών του Spring Boot μέσα στην εργασία με αποτέλεσμα να μην χρειάζεται η δήλωση του version κομματιού για τα Spring boot dependencies.

  Τα βασικά tasks που προσφέρει το Spring Boot Plugin είναι:

  1. bootRun τρέχει την Spring Boot εφαρμογή.
  2. bootJar πακετάρει την Spring Boot εφαρμογή σε uber jar, το οποίο περιέχει τον κώδικα της εφαρμογής και όλα τα dependencies.

- Kotlin JVM Plugin το οποίο ορίζει την έκδοση της Kotlin που χρησιμοποιείται.

- Kotlin Spring Plugin το οποίο προσθέτει open qualifier σε classes που χρησιμοποιούνται από Spring βιβλιοθήκη.

  Στη Java όλες οι classes είναι "ανοιχτές" για κληρονομικότητα εξ' ορισμού – δεν είναι final. Στην Kotlin, όλες οι classes είναι εξ' ορισμού final.

  Η Spring βιβλιοθήκη απαιτεί ορισμένες classes να μην είναι final. Το Kotlin Spring Plugin αναγνωρίζει ποίες classes το Spring απαιτεί να μην είναι final και προσθέτει το open keyword της Kotlin που ανοίγει την class για κληρονομικότητα: σε byte code επίπεδο αυτό σημαίνει ότι η class δεν θα έχει final keyword.

  Το Figure 2: Byte Code της class που δεν απαιτείται από την Spring βιβλιοθήκη να είναι non-final. παρουσιάζει το byte code της class που δεν χρησιμοποιείται από Spring και έχει το final keyword στον ορισμό της class.

  Το Figure 3: Byte code της class που χρησιμοποιείται από Spring βιβλιοθήκη παρουσιάζει το byte code της class που χρησιμοποιείται από Spring Framework και απαιτεί να είναι non-final και δείχνει

```
Arslan
class EntityAlreadyExistsException : RuntimeException()
```

```
Decompile  Inline  Optimization  Assertions  IR JVM Target: 1.8
1 // ================iee/ihu/gr/pricemonitoringbackend/EntityAlreadyExistsException.class =================
2 // class version 52.0 (52)
3 // access flags 0x31
4 public final class iee/ihu/gr/pricemonitoringbackend/EntityAlreadyExistsException extends java/lang/RuntimeException {
5
```

Figure 2: Byte Code της class που δεν απαιτείται από την Spring βιβλιοθήκη να είναι non-final.

```
import ...

BarracudaX +3
@Configuration
class InfrastructureConfiguration {
```

```
Decompile  Inline  Optimization  Assertions  IR JVM Target: 1.8
1 // ================iee/ihu/gr/pricemonitoringbackend/configuration/InfrastructureConfiguration.class =========
2 // class version 52.0 (52)
3 // access flags 0x21
4 public class iee/ihu/gr/pricemonitoringbackend/configuration/InfrastructureConfiguration {
5
```

Figure 3: Byte code της class που χρησιμοποιείται από Spring βιβλιοθήκη

  το αποτέλεσμα του Kotlin Spring Plugin το οποίο αφαίρεσε το final keyword από τον ορισμό της class.

- Kotlin JPA Plugin προσφέρει παρόμοια λειτουργικότητα με Kotlin Spring Plugin(προσθέτει open keyword σε classes) αλλά για Hibernate framework.
- Kotlinx serialization plugin. Αυτό το Plugin αναγνωρίζει το @Serializable annotation της kotlinx serialization βιβλιοθήκης και δημιουργεί για αυτές τις classes serializers που χρησιμοποιούνται από την βιβλιοθήκη για serialization/deserialization των αντικειμένων σε διάφορα formats(πχ. JSON).

To Figure 4: Class που χρησιμοποιεί @Serializable annotation. δείχνει την HotelData class η οποία έχει το @Serializable annotation και το Figure 5: To byte code της class που χρησιμοποιεί @Serializable annotation για την οποία έχει δημιουργηθεί kotlinx serializer. δείχνει το byte code της

```kotlin
@Serializable
data class HotelData(

    @field:NotBlank(message = "...") val name: String,

    @field:NotBlank(message = "...") val description: String,

    @field:Valid val location: Location,

    @field:URL(message = "...", protocol = "https") val url: String,

    @field:NotBlank(message = "...") val hotelType: String,

    @field:DecimalMin(value = "0.0", message = "...")
    @field:DecimalMax(value = "10.0", message = "...")
    @Serializable(with = BigDecimalSerializer::class)
    val score: BigDecimal,

    val rooms: Set<ScrappedRoomInfo> = emptySet()
){
```

Figure 4: Class που χρησιμοποιεί @Serializable annotation.
class για την οποία έχει δημιουργηθεί serializer.

```java
public static final class $serializer implements GeneratedSerializer {
   @NotNull
   public static final $serializer INSTANCE;
   // $FF: synthetic field
   private static final SerialDescriptor $$serialDesc;

   private $serializer() {
   }

   static {
      $serializer var0 = new $serializer();
      INSTANCE = var0;
      PluginGeneratedSerialDescriptor var1 = new PluginGeneratedSerialDescriptor("iee.ihu.gr.pricemonitoringbackend.dto.HotelData", (GeneratedSerializer)INSTANCE, 7);
      var1.addElement("name", false);
      var1.addElement("description", false);
      var1.addElement("location", false);
      var1.addElement("url", false);
      var1.addElement("hotelType", false);
      var1.addElement("score", false);
      var1.addElement("rooms", true);
      $$serialDesc = var1;
   }

   @NotNull
   public KSerializer[] typeParametersSerializers() { return DefaultImpls.typeParametersSerializers(this); }

   @NotNull
   public SerialDescriptor getDescriptor() { return $$serialDesc; }
```

Figure 5: To byte code της class που χρησιμοποιεί @Serializable annotation για την οποία έχει δημιουργηθεί kotlinx serializer.

2.3.1 Gradle Build Lifecycle

Όπως έχει αναφερθεί, η βασική μονάδα εκτέλεσης στο gradle είναι το task. Το gradle εκτελεί τα task σε συγκεκριμένη σειρά βάση των dependencies που έχουν οριστεί μεταξύ τους.

Το gradle χρησιμοποιεί τα tasks για να δημιουργήσει κατευθυνόμενο άκυκλο γράφο(ΚΑΓ)[4].

Η εκτέλεση του build με Gradle αποτελείται από τρία στάδια[5] τα οποία εκτελούνται το ένα μετά το άλλο:

1. Initialization
   Σε αυτό το στάδιο το gradle βρίσκει και ερμηνεύει το settings.gradle.kts αρχείο στο οποίο ορίζονται τα projects που θα συμμετέχουν στο build.
2. Configuration
   Σε αυτό το στάδιο το gradle διαβάζει και ερμηνεύει τα build αρχεία των projects που συμμετέχουν στο build και δημιουργεί τον γράφο(ΚΑΓ) από tasks.
3. Execution
   Εκτελεί τα tasks με συγκεκριμένη σειρά βάση των εξαρτήσεων που έχουν οριστεί μεταξύ τους.

2.3.2 Gradle Incremental Builds

Το gradle χρησιμοποιεί ορισμένες βελτιστοποιήσεις για να μειώσει τον χρόνο που απαιτείται για το build.

Μία από τις βελτιστοποίησης του gradle ονομάζεται incremental builds[6]. Incremental build είναι βελτιστοποίηση του build στο οποίο τα tasks για τα οποία οι input παράμετροι δεν έχουν αλλάξει δεν εκτελούνται μετά την τελευταία εκτέλεσή τους αλλά χρησιμοποιείται το αποθηκευμένο output των task.

2.4 Gradle Wrapper

Gradle Wrapper είναι script αρχείο το οποίο μπορεί να χρησιμοποιηθεί για εκτέλεση των gradle task.

Το gradle wrapper κατεβάζει συγκεκριμένη έκδοση του gradle(την πρώτη φορά) και έπειτα εκτελεί το build χρησιμοποιώντας την έκδοση που κατέβηκε την πρώτη φορά.

Το script αποθηκεύεται σαν αρχείο του project και έχει όνομα gradlew(για Linux) και gradlew.bat(για Windows).

Ο κώδικας του wrapper είναι αποθηκευμένος σε gradle φάκελο και το gradle που κατεβάζει gradle wrapper αποθηκεύεται μέσα στον φάκελο .gradle.

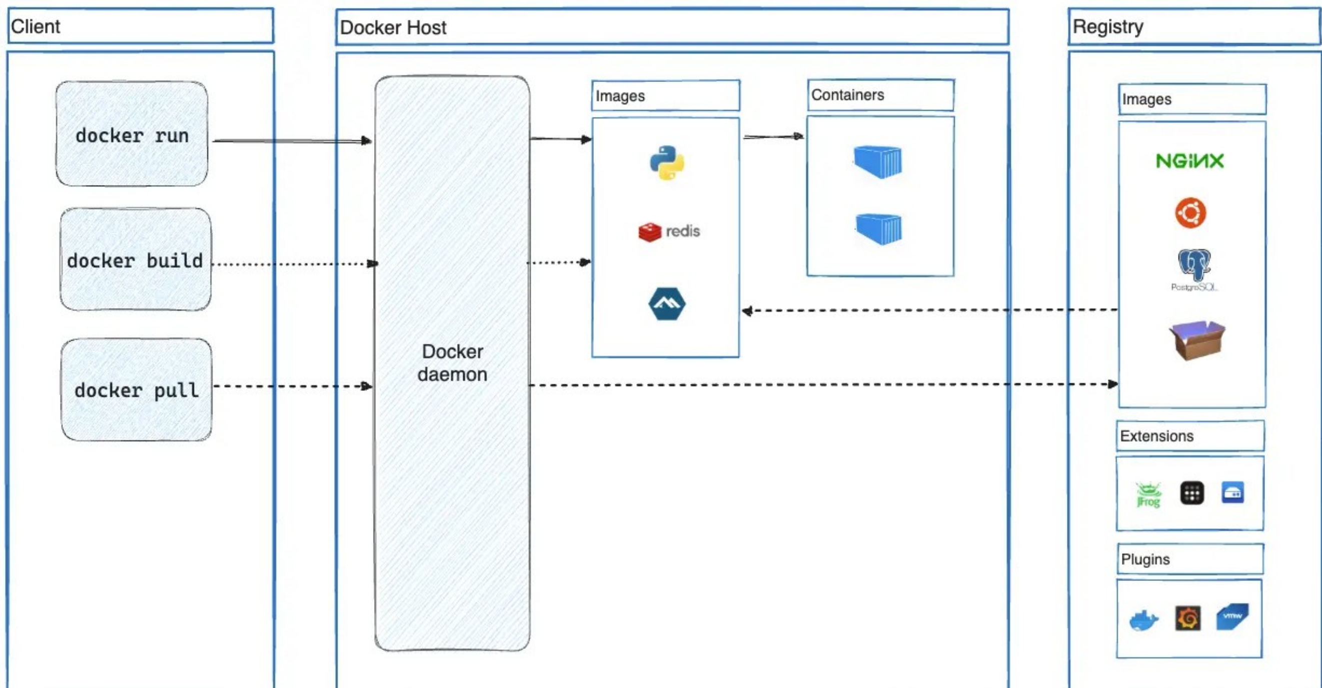
Τα πλεονεκτήματα του gradle wrapper είναι

1. Χρήστης που δεν έχει gradle μπορεί να κάνει build την εφαρμογή χρησιμοποιώντας το gradle wrapper.
2. Χρησιμοποιώντας το gradle wrapper όλοι οι χρήστες χρησιμοποιούν την ίδια έκδοση του gradle ανεξάρτητα από τις εκδόσεις που μπορεί να έχουν τοπικά στον υπολογιστή.

2.5 Docker

Το Docker επιτρέπει το πακετάρισμα των εφαρμογών σε μία συγκεκριμένη δομή η οποία στην βιβλιογραφία του Docker ονομάζονται images(εικόνες)[7],και την εκτέλεση τους μέσα σε απομονωμένα περιβάλλοντα τα οποία στην ονομάζονται containers(δοχεία)[8].

Το Docker ακολουθεί client-server αρχιτεκτονική όπου ο client επικοινωνεί με το docker process για να εκτελέσει κάποιες ενέργειες όπως δημιουργία εικόνας, εκτέλεση εικόνας μέσα σε δοχείο κ.α[9].



Έκτος από Docker, χρησιμοποιήθηκε το Docker Compose Plugin που επιτρέπει τον ορισμό των δοχείων μέσα από ένα YAML αρχείο. Έχοντας αυτό το αρχείο,το docker compose μπορεί να το διαβάσει και να ξεκινήσει όλα τα containers που έχουν οριστεί χρησιμοποιώντας το Docker.

2.5.1 Dockerfile

Το Dockerfile είναι αρχείο από το οποίο το Docker μπορεί να δημιουργήσει την εικόνα της εφαρμογής η οποία στην συνέχεια μπορεί να χρησιμοποιηθεί για την έναρξη του docker δοχείου.

Το Dockerfile αποτελείται από εντολές όπως RUN,COPY,CMD κτλ. που δηλώνουν τα βήματα για τη δημιουργία της εικόνα της εφαρμογής.

Η πρώτη εντολή του Dockerfile είναι FROM και δηλώνει την αρχική εικόνα(parent image) πάνω στην οποία θα βασίζεται η εικόνα που θα δημιουργηθεί.

Οι εικόνες στο Docker αποτελούνται από στρώματα, και σχεδόν κάθε Docker εντολή που με κάποιο τρόπο αλλάζει το περιεχόμενο της εικόνας δημιουργεί καινούριο στρώμα το οποίο τοποθετείται πάνω στα προηγούμενα δημιουργώντας έτσι stack από στρώματα.

Τα πλεονεκτήματα αυτής την αρχιτεκτονική είναι

1. Επαναχρησιμοποίηση των στρωμάτων για πιο γρήγορα build των εικόνων.

Κατά τη δημιουργία της εικόνας, αν τα αρχεία που χρησιμοποιήθηκαν για τη δημιουργία του στρώματος δεν έχουν αλλάξει, Docker χρησιμοποιεί το αποθηκευμένο στρώμα αντί να τρέχει ξανά την εντολή για να δημιουργήσει το ίδιο στρώμα.

Στο Dockerfile της εφαρμογής πρώτα κάνω copy21 τα Gradle αρχεία της εφαρμογής και τρέχω το gradle build. Ενώ το build αποτυχαίνει επειδή σε αυτό το στρώμα δεν υπάρχει ακόμη ο κώδικας της εφαρμογής,κατεβάζει όμως όλα τα dependencies της εφαρμογής και τα αποθηκεύει στο στρώμα με αποτέλεσμα τα επόμενα build της εικόνας όπου γίνονται μόνο αλλαγές του κώδικα να γίνονται πιο γρήγορα καθώς το Docker θα επαναχρησιμοποιήσει το στρώμα με τα dependencies της εφαρμογής αντί να τα ξανακατεβάζει.

2. Κάποιες εικόνες μπορούν να χρησιμοποιούν τα ίδια στρώματα με αποτέλεσμα να μειώνεται η χρήση της μνήμης για την αποθήκευσή τους καθώς κάθε καινούρια εντολή δεν αλλάζει το στρώματα, αλλά προσθέτει ένα επιπλέον στρώμα.

   Δηλαδή, τα στρώματα είναι immutable και για αυτό το λόγο επιτρέπεται κοινόχρηστη χρήση τους από διαφορετικές εικόνες

Τέλος, για τη δημιουργία του Dockerfile της εφαρμογής χρησιμοποιήθηκε το multi-stage build του Docker[10]. Αυτή η τεχνική επιτρέπει η δημιουργία της τελικής εικόνας να γίνεται σε δύο ή περισσότερα στάδια. Αυτό έχει τα εξής πλεονεκτήματα

1. Τα στάδια ή κάποια μέρη των σταδίων μπορούν να τρέξουν παράλληλα με αποτέλεσμα το build της εικόνας να γίνεται πιο γρήγορα.

   Ενώ ένα στάδιο μπορεί να χρησιμοποιεί το αποτέλεσμα κάποιου άλλου σταδίου, μέχρι να φτάσει σε αυτό το σημείο μπορεί να τρέξει παράλληλα με το άλλο στάδιο.

2. Το μέγεθος της εικόνα είναι πιο μικρό καθώς κάθε στάδιο μπορεί να χρησιμοποιεί μόνο τα εργαλεία που χρειάζονται για την εκτέλεσή του,και μόνο το τελευταίο στάδιο εμπεριέχεται στην τελική εικόνα.

   Στο Dockerfile της εφαρμογής μου υπάρχουν δύο στάδια: ένα για build της εφαρμογής(κατέβασμα των dependencies, χρησιμοποίηση του Gradle build tool για τη δημιουργία του εκτελέσιμου jar) και ένα άλλο στάδιο για την εκτέλεση του jar αρχείου που δημιουργήθηκε από το πρώτο στάδιο με αποτέλεσμα η τελική εικόνα να μην περιέχει τον κώδικα και τα εργαλεία που χρειάζονταν για το build της εφαρμογής, άλλα μόνο το τελικό εκτελέσιμο jar αρχείο και το περιβάλλον που χρειάζεται για την εκτέλεση του.

```
FROM gradle:8.2.1-jdk17-jammy as build

WORKDIR /app

COPY build.gradle.kts settings.gradle.kts gradlew gradle/ ./

RUN gradle build --debug -x bootJar || exit 0

COPY . .

RUN gradle build --debug -x test

FROM openjdk:17

COPY --from=build /app/build/libs/PriceMonitoringBackend.jar server.jar

CMD java -jar server.jar
```

Μετά την εκτέλεση benchmark για τη δημιουργία της τελικής εικόνας της εφαρμογής με dependency caching(η λήψη των dependencies και το build του jar γίνονται σε διαφορετικά στρώματα) και χωρίς

dependency caching(η λήψη των dependencies και το build γίνονται στο ίδιο στρώμα),τα αποτελέσματα ήταν τα εξής

Table 1: Αποτελέσματα της docker build εντολής με και χωρίς caching.

Χωρίς Dependency Caching	**Με Dependency Caching**
81s	32s

Ο χρόνος δημιουργίας της εικόνας με cache ήταν μικρότερος(2.5 φορές πιο γρήγορο build).

Ακολουθούν τα αποτελέσματα του μεγέθους της τελικής εικόνας με multi-stage build και χωρίς multi-stage build.

Table 2: Αποτελέσματα docker build εντολής με και χωρίς multi-staging

Χωρίς Multi-Stage Build	**Με Multi-Stage Build**
1410MB	533MB

Το τελικό μέγεθος της εικόνας με multi-stage build ήταν 2.6 φορές μικρότερο από το μέγεθος της εικόνας για την οποία δεν χρησιμοποιήθηκε το multi-stage.

2.5.2 Docker Registry

Docker registry είναι server ο οποίος χρησιμοποιείται για την αποθήκευση των docker εικόνων[11].

Για την αποθήκευση της εικόνας της εφαρμογής χρησιμοποιήθηκε Docker hub που αποτελεί Docker Registry.

2.5.3 Docker Repository

Repository είναι οντότητα που χρησιμοποιείται για την αποθήκευση εικόνων. Κάθε εικόνα στο repository έχει μοναδικό tag, και κάθε εικόνα στο Docker hub έχει μοναδικό αναγνωριστικό το οποίο αποτελείται από το όνομα του repository + το tag της εικόνας.

Για την αποθήκευση της εικόνας της εφαρμογής δημιούργησα repository στο docker hub – barracudax/iee-ihu-gr-174958.

2.6 Servlets

Η εφαρμογή υλοποιήθηκε χρησιμοποιώντας το Servlet Specification. Το Servlet Specification ορίζει Java API το οποίο μπορεί να χρησιμοποιηθεί και να υλοποιηθεί από προγραμματιστές για την υλοποίηση web εφαρμογών που τρέχουν σε JVM.

Το Servlet Specification ορίζει διάφορα interfaces τα οποία μπορούν να υλοποιηθούν για την κατασκευή web εφαρμογών. Παρακάτων αναφέρονται τα πιο σημαντικά συστατικά του Servlet Specification.

2.6.1 Servlet

Τα συστατικά τα οποία διαχειρίζονται τα requests και παράγουν responses ονομάζονται servlets και είναι classes που υλοποιούν το Servlet interface. Το interface ορίζει service(…) μέθοδο η οποία καλείται όταν το server δέχεται καινούριο request.

Συνήθως, αντί για την υλοποίηση του Servlet interface, είναι πιο εύκολο να γίνει υλοποίηση της class που κληρονομεί από Servlet interface και ονομάζεται HttpServlet. Το πλεονέκτημα δημιουργίας servlet που κληρονομεί από HttpServlet είναι ότι το HttpServlet υλοποιεί service(…) μέθοδο και προσφέρει μεθόδους που μπορούν να υλοποιηθούν και που αντιστοιχούν σε HTTP μεθόδους: doGet, doPost κτλ.

2.6.2 Servlet Container

Το συστατικό το οποίο διαχειρίζεται και καλεί τις μεθόδους του servlet όταν ο server δέχεται ένα request ονομάζεται servlet container. Servlet Container μπορεί να είναι ειδικός server ή κάποια επέκταση για ήδη υπάρχον server.

Η αρμοδιότητα του servlet container είναι να δέχεται το request και να κάλει το αντίστοιχο servlet.

2.6.3 Servlet Filter

Το servlet specification επιτρέπει τη δήλωση του filter chain – filter αντικείμενα ενωμένα σε μία εννοιολογική αλυσίδα – από το οποίο θα περάσει το request πριν φτάσει στο servlet για τελική επεξεργασία. Το filter chain αποτελείται από αντικείμενα των classes που υλοποιούν το Filter interface.

Τα φίλτρα χρησιμοποιούνται για υλοποίηση διαφόρων λειτουργιών της εφαρμογής όπως

1. Ασφάλεια. Σε αυτή την περίπτωση, το φίλτρο ελέγχει αν ο χρήστης έχει δικαίωμα να κάνει request ή όχι, και συνεχίζει την επεξεργασία μόνο αν ο χρήστης είναι εξουσιοδοτημένος να κάνει το request.
2. Συμπίεση δεδομένων. Σε αυτή την περίπτωση, το φίλτρο συμπιέζει τα δεδομένα πριν αυτά αποσταλούν στον client.

Γενικά, τα φίλτρα χρησιμοποιούνται για την υλοποίηση της λογικής η οποία μπορεί να χρησημεύσει για περισσότερα από ένα servlets.

2.6.4 Υλοποίηση της εφαρμογής

Στην εφαρμογή χρησιμοποίησα Spring MVC το οποίο δημιουργεί μοναδικό Servlet που ονομάζεται DispatcherServlet.

Ως servlet container χρησιμοποίησα το embedded tomcat servlet container που προσφέρεται από το Spring Boot. Embedded servlet container ξεκινάει μαζί με την εφαρμογή. Το πλεονέκτημα του embedded servlet container είναι ότι η εφαρμογή μπορεί να τρέξει χωρίς υπάρχον servlet container στο μηχάνημα χρησιμοποιώντας το εκτελέσιμο jar.

2.7 Spring

Το Spring αποτελεί μία οικογένεια από βιβλιοθήκες που χρησιμοποιούνται για τη δημιουργία εφαρμογών. Από όλα τα Spring Projects χρησιμοποιήθηκαν τα παρακάτω.

2.7.1 Spring Framework

2.7.1.1 *Spring* IoC

Η βασική λειτουργία του Spring Framework είναι η υλοποίηση IoC(Inversion of Control) αρχής. IoC είναι η αρχή σύμφωνα με την οποία τα αντικείμενα δηλώνουν τα dependencies(άλλα αντικείμενα που χρειάζονται για τη λειτουργία τους) μέσα από constructor παραμέτρους ή/και setters. Η αρχή έχει αυτό το όνομα επειδή είναι αντίστροφη(inverse) της αρχής βάση της οποίας τα αντικείμενα δημιουργούν τα dependencies τους.

Οπότε, η βασική λειτουργία του Spring Framework είναι η ανάγνωση του configuration της εφαρμογής και η δημιουργία των αντικειμένων της εφαρμογής ακολουθώντας IoC αρχή.

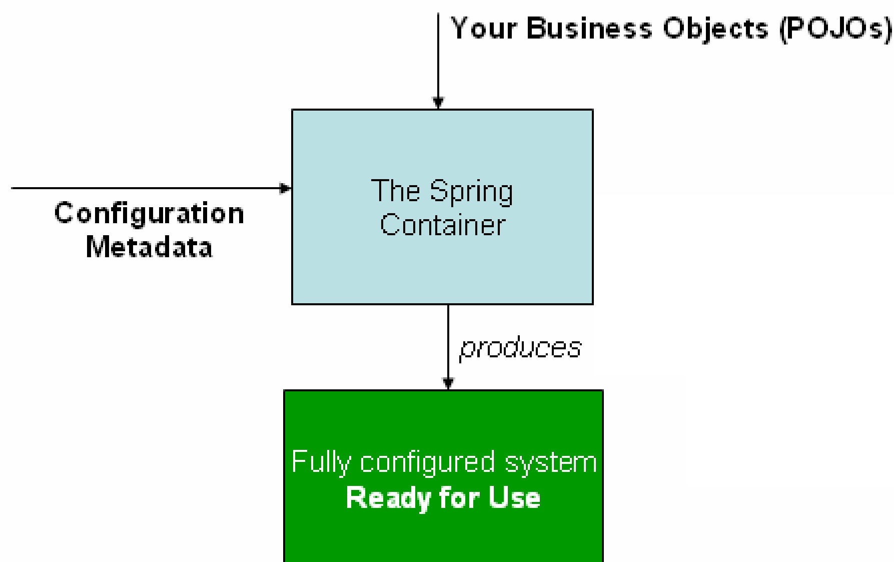


Figure 6: Η λειτουργία του Spring Framework

Τα κύρια πλεονεκτήματα της IoC αρχής είναι:

1. Ευκολότερο testing καθώς τα dependencies που δέχεται constructor/setter μπορούν να αντικατασταθούν με mock αντικείμενα.
2. Επειδή το αντικείμενο δέχεται τα dependencies από κάποιο εξωτερικό container(Spring), είναι πιο εύκολη η αλλαγή της υλοποίησης του dependency χωρίς να χρειάζεται να γίνεται αλλαγή της υλοποίησης της class. Μόνο το configuration πρέπει να αλλάξει για να οριστεί διαφορετική υλοποίηση.

Για παράδειγμα, στην εργασία η class HotelServiceImpl δηλώνει τρία dependencies μέσα από τον constructor της class.

```
@Service
@Transactional
class HotelServiceImpl(
   private val hotelRepository: HotelRepository,
   private val roomService: RoomService,
   private val scrappedHotelRepository: ScrappedHotelRepository,
) : HotelService { .... }
```

Επειδή η class δηλώνει και δέχεται τα dependencies μέσα από το constructor, αν θέλω να γράψω τεστ για την class, μπορώ να δημιουργήσω το αντικείμενο της class και να περάσω ως dependencies τεστ mocks.

Τέλος, αν για κάποιο λόγο θα πρέπει να αλλάξει η υλοποίηση του HotelRepository,δεν χρειάζεται να αλλάξω τον κώδικα του HotelServiceImpl[2], αλλά μόνο το configuration για να δηλώσω άλλη υλοποίηση του HotelRepository.

Όλα τα υπόλοιπα Spring projects βασίζονται στο Spring Framework IoC.

2.7.1.2 Spring Bean Configuration

Το Spring Configuration δηλώνει τα beans – τα αντικείμενα που διαχειρίζεται το Spring Framework IoC και τα οποία αποτελούν τα βασικά συστατικά της εφαρμογής.

Το Spring υποστηρίζει διαφορετικά formats για το configuration: XML,Groovy,Java/Kotlin classes. Στην εφαρμογή χρησιμοποίησα Kotlin classes.

Ακολουθεί InfrastructureConfiguration class η οποία περιέχει Spring configuration.

```
@Configuration
class InfrastructureConfiguration {

   @Bean
   fun base64Decoder() : Base64.Decoder = Base64.getDecoder()

   @Bean
   fun java8JacksonModule() : Module = Jdk8Module()

   @Bean
   fun jacksonMessageConverter(objectMapper: ObjectMapper) : Jackson2JsonMessageConverter = Jackson2JsonMessageConverter(objectMapper)

   @Bean
   fun kotlinJacksonModule() : Module = KotlinModule.Builder().build()

   @Bean
   fun jacksonCustomizer() : Jackson2ObjectMapperBuilderCustomizer = Jackson2ObjectMapperBuilderCustomizer { builder ->
        builder.postConfigurer { mapper -> mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,false) }
   }
```

2 Αυτό ισχύει μόνο όταν για τον ορισμό των dependencies χρησιμοποιούνται οι τύποι των interfaces και όχι οι συγκεκριμένες υλοποιήσεις. Στο παράδειγμα, αν είχα χρησιμοποιήσει RoomServiceImpl ως τύπο, τότε δεν θα μπορούσα να αλλάξω την υλοποίηση του RoomServiceImpl χωρίς να αλλάξω το κώδικα του HotelServiceImpl.

```
...
}
```

Το configuration περιέχει bean definitions σε μορφή μεθόδων τα οποία ορίζουν τα παρακάτω πράγματα για κάθε αντικείμενο/bean:

1) Τα dependencies – αλλά beans τα οποία χρειάζεται για τη λειτουργία του το bean. Σε Kotlin/Java class configuration, τα dependencies δηλώνονται μέσα από την λίστα παραμέτρων μεθόδων. Στο παραπάνω παράδειγμα, το jacksonMessageConverter δηλώνει ένα dependency στο object mapper αντικείμενο.
2) Το όνομα/id του bean. Η αναφορά στα beans γίνεται με βάση το id/όνομα του ή τύπο. Όταν χρησιμοποιείται Kotlin/Java κώδικας ως configuration, το id του bean αντιστοιχεί στο όνομα της μεθόδου, και μπορεί να αλλάξει χρησιμοποιώντας το name attribute του @Bean annotation.
3) Το scope του bean. Το scope του bean ορίζει τον κύκλο ζωής του bean μέσα στον IoC container. Στο παράδειγμα, όλα τα beans έχουν τον εξ' ορισμού singleton scope.

   Το Spring επιτρέπει τα εξής scopes:

   i) Singleton scope: δημιουργείται ένα μοναδικό bean από το bean definition.
   ii) Prototype scope: κάθε φορά που ζητείται το bean, δημιουργείται καινούριο αντικείμενο.
   iii) Request,Session,Application. Αυτά τα scopes υποστηρίζονται σε WebApplicationContext, και έχουν αντίστοιχο scope του request,session και web εφαρμογής.

Μετά την ανάγνωση του configuration, το Spring δημιουργεί application context το οποίο αποτελεί IoC container και περιέχει όλα τα beans της εφαρμογής.

Κάποιος από τις βασικές λειτουργίες του Spring application context:

1. Δυνατότητα δημιουργίας bean ή επιστροφή ήδη υπάρχοντος bean από το application context. Το bean επιστρέφεται αφού έγινε η οριστικοποίηση του: έχουν δημιουργηθεί τα άλλα beans τα οποία αποτελούν dependencies του, έχουν κληθεί Spring extension μέθοδοι(όπως για παράδειγμα, init μέθοδος).
2. Δυνατότητα αποστολής event μέσω του application context μεταξύ των beans.

2.7.1.3 Spring Component Scanning

Με το Spring component scanning, το Spring μπορεί να βρει αυτόματα τα beans κάνοντας αναζήτηση των classes που έχουν το @Component annotation χωρίς να απαιτείται η δήλωση του bean definition μέσα στο configuration αρχείο. Στον παραπάνω παράδειγμα25, το @Configuration annotation έχει το @Component annotation, όπως φαίνεται από τον source κώδικα της class Figure 7: Spring @Configuration annotation definition, οπότε αυτή η class αυτόματα εντοπίζεται όταν χρησιμοποιείται το Spring component scanning.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
```

Figure 7: Spring @Configuration annotation definition

Για την ενεργοποίηση του ComponentScan χρησιμοποιείται το @ComponentScan annotation.

2.7.1.4 Environment

To Environment interface είναι abstraction ενσωματωμένο μέσα στο Spring IoC container και χρησιμοποιείται για τη μοντελοποίηση των ιδιοτήτων εφαρμογής και των profiles.

2.7.1.4.1 Ιδιότητες

To Environment αντικείμενο αρχικοποιείται με ιδιότητες από διαφορετικές πηγές και προσφέρει μοναδικό σημείο πρόσβασης στις ιδιότητες της εφαρμογής.

Οι πηγές των ιδιοτήτων είναι

1. Μεταβλητές περιβάλλοντος συστήματος
2. Ιδιότητες του JVM(ορίζονται μέσα από command line χρησιμοποιώντας το -D option)
3. Χρησιμοποιώντας @PropertySource annotation μπορούμε να ορίσουμε properties αρχείο το οποίο θα χρησιμοποιηθεί ως πηγή ιδιοτήτων.
4. Χρησιμοποιώντας τα application.properties αρχεία. Αυτό επιτρέπεται μόνο όταν χρησιμοποιείται το Spring Boot.

Οι ιδιότητες έχουν την μορφή *όνομα.ιδιότητας=τιμή.ιδιότητας*.

Μετά την αρχικοποίηση του Environment αντικειμένου, μπορούμε να το χρησιμοποιήσουμε με τους παρακάτω τρόπους για να διαβάσουμε τις ιδιότητες μέσα στα beans

1. Χρησιμοποιώντας το Environment αντικείμενο ως dependency του bean.
2. To @Value annotation επιτρέπει το injection των ιδιοτήτων μέσα στα beans.
3. To @ConfigurationProperties του Spring boot επιτρέπει να κάνουμε inject αντικείμενα της class οι ιδιότητες της οποίας χρησιμοποιούνται για injection των ιδιοτήτων εφαρμογής. Αυτό επιτρέπεται όταν οι ιδιότητες είναι ιεραρχικές. Για παράδειγμα, όταν έχουμε ιδιότητες a.b.c,a.b.d και a.b.e, μπορούμε να φτιάξουμε class με ιδιότητες c,d και e και να χρησιμοποιήσουμε @ConfigurationProperties(“a.b”) πάνω στην class ώστε οι ιδιότητες της class να αντιστοιχίζουν στις ιδιότητες της εφαρμογής.

```
@Service
class TokenGeneratorImpl(
   private val macSigner: MACSigner,


   @Value("\${token.issuer}")
   private val issuer: String,


   @Value("\${token.duration}")
   private val tokenDuration: Duration
) : TokenGenerator { .... }
```

Στο παραπάνω παράδειγμα, χρησιμοποιώντας το @Value annotation ορίζω ότι το bean απαιτεί injection δύο ιδιοτήτων που έχουν τα ονόματα token.issuer και token.duration αντίστοιχα.

2.7.1.4.2 Profiles

Το Spring προσφέρει το @Profile annotation το οποίο δηλώνει ότι το bean πρέπει δημιουργηθεί μόνο σε περίπτωση που το profile είναι ενεργοποιημένο και προσφέρει default profile που είναι ενεργοποιημένο όταν κανένα άλλο profile δεν είναι ενεργοποιημένο.

Τα profile επιτρέπουν τον προσδιορισμό πολλαπλών bean ίδιου τύπου αλλά για διαφορετικά profiles, όπου κάθε profile μπορεί να χρησιμοποιείται σε διαφορετικό περιβάλλον εκτέλεσης της εφαρμογής.

Για παράδειγμα, όταν η εφαρμογή τρέχει με dev profile, χρησιμοποιείται ένα Email Service το οποίο κάνει log το mail στην κονσόλα και δεν στέλνει πραγματικά mail. Ενώ όταν η εφαρμογή τρέχει με default profile, χρησιμοποιείται Email Service που χρησιμοποιεί έναν email service provider για την αποστολή των mail.

Τα ενεργοποιημένα profiles μπορούν να οριστούν χρησιμοποιώντας *spring.profiles.active* ιδιότητα. Αυτή η ιδιότητα μπορεί να οριστεί για παράδειγμα μέσα από μεταβλητές περιβάλλοντος συστήματος ή χρησιμοποιώντας JVM ιδιότητες.

Τα profiles στην εργασία χρησιμοποιήθηκαν σε σύνδεση με τα Spring Boot application αρχεία όπου το Spring Boot φορτώνει διαφορετικό properties αρχείο αναλόγως με τα ενεργοποιημένα profiles. Για παράδειγμα, όταν το dev profile είναι ενεργοποιημένο, εκτός από την φόρτωση ιδιοτήτων από το αρχείο application.properties, το Spring Boot φορτώνει ιδιότητες από το αρχείο application-dev.properties αν υπάρχει.

Τα profiles που έχουν οριστεί στην εργασία είναι

1. dev: χρησιμοποιήθηκε για εκτέλεση της εφαρμογής τοπικά κατά την ανάπτυξη της εφαρμογής. Το profile χρησιμοποιεί διαπιστευτήρια και localhost για τη σύνδεση στη βάση δεδομένων, στο rabbimtq και στον scraper.
2. generate: κατά την εκτέλεση της εφαρμογής με αυτό το profile, δημιουργούνται create.sql και drop.sql αρχεία τα οποία περιέχουν SQL εντολές για τη δημιουργία και την διαγραφή του σχήματος της βάσης αντίστοιχα.

2.7.1.5 Spring Transaction Management

Το Spring προσφέρει API για διαχείριση συναλλαγών το οποίο είναι ανεξάρτητο από APIs για διαχείριση συναλλαγών που προσφέρονται από διάφορες βιβλιοθήκες(JDBC, JPA,JTA,Hibernate κτλ).

Το κεντρικό interface για τη διαχείριση συναλλαγών στο Spring είναι TransactionManager. Τα δυο interfaces που κληρονομούν από αυτό είναι

1. PlatformTransactionManager. Αποτελεί διαχειριστή συναλλαγών που χρησιμοποιείται σε imperative προγραμματισμό όπου η συναλλαγή σχετίζεται με το νήμα εκτέλεσης.
2. ReactiveTransactionManager. Αποτελεί διαχειριστή συναλλαγών που χρησιμοποιείται σε reactive προγραμματισμό όπου η συναλλαγή σχετίζεται με το context του reactive stream.

Το Spring προσφέρει δύο τρόπους διαχείρισης συναλλαγών

1. Προγραμματιστικά χρησιμοποιώντας είτε το API του PlatformTransactionManager είτε TransactionTemplate.
2. Declarative transaction management χρησιμοποιώντας @Transactional annotation.

Στην εργασία χρησιμοποίησα @Transactional για τη διαχείριση συναλλαγών.

Για τα beans τα οποία έχουν το @Transactional annotation το Spring δημιουργεί proxy το οποίο ξεκινάει καινούρια συναλλαγή κατά την κλήση της μεθόδου και εκτελεί commit ή rollback αναλόγως με το αποτέλεσμα της μεθόδου[12]. Το Figure 8: Λειτουργία των Spring Proxy για διαχείριση συναλλαγών δείχνει τη λειτουργία του proxy αντικειμένου όταν κάποιος καλεί μια μέθοδο.

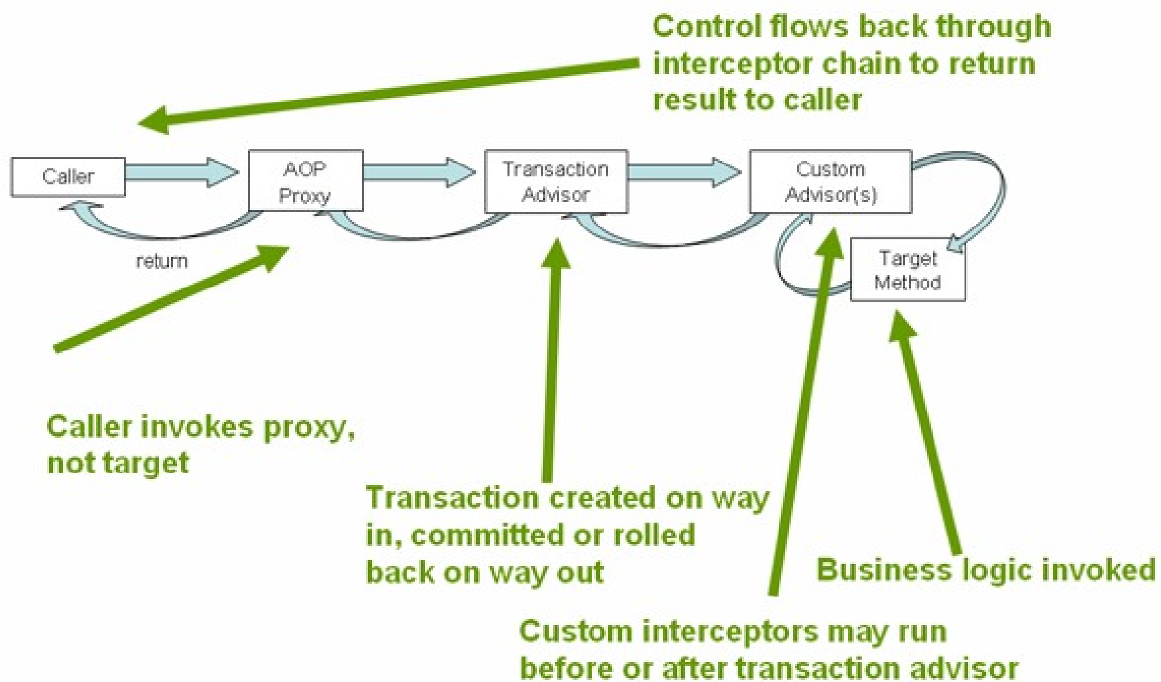


Figure 8: Λειτουργία των Spring Proxy για διαχείριση συναλλαγών

Όταν κάποιος καλεί μέθοδο πάνω σε αντικείμενο για το οποίο έχει δημιουργηθεί proxy, η κλήση γίνεται όχι στο αρχικό αντικείμενο αλλά πάνω στο proxy. Το proxy δημιουργεί καινούρια συναλλαγή αφού έχει διαβάσει και ερμηνεύσει τα @Transactional annotations της μεθόδου και της class του αντικειμένου. Μετά την έναρξη της συναλλαγής, καλεί την μέθοδο πάνω στο αρχικό αντικείμενο. Αφού τελειώσει η εκτέλεση της μεθόδου, το proxy αποφασίζει αν η συναλλαγή πρέπει να τελειώσει με commit ή rollback.

Εξ' ορισμού, αν η κλήση της μεθόδου δεν πετάει εξαίρεση και τελειώνει κανονικά, το proxy θα εκτελέσει commit. Αν η κλήση της μεθόδου τελειώνει με εξαίρεση(RuntimeException), τότε το proxy θα εκτελέσει rollback.

Τα attributes του @Transactional ελέγχουν τις ιδιότητες της συναλλαγής και τους rollback/commit κανόνες. Συγκεκριμένα

1. value/transactionManager ορίζουν τον διαχειριστή συναλλαγών που πρέπει να χρησιμοποιηθεί για τη διαχείριση της συναλλαγής όταν υπάρχουν περισσότερα από έναν bean διαχειριστή συναλλαγών στο context. Εξ' ορισμού χρησιμοποιεί τον μοναδικό διαχειριστή συναλλαγών.
2. Propagation – η propagation στρατηγική όταν η μέθοδος καλείται και υπάρχει τρέχουσα συναλλαγή. Οι επιτρεπτές τιμές είναι
   - Required(εξ' ορισμού τιμή): συμμετέχει στην ήδη υπάρχουσα συναλλαγή ή δημιουργεί καινούρια αν δεν υπάρχει.
   - Supports: συμμετέχει στην ήδη υπάρχουσα συναλλαγή αν υπάρχει. Διαφορετικά, εκτελείται χωρίς να ξεκινάει καινούρια συναλλαγή.
   - Mandatory: συμμετέχει στην ήδη υπάρχουσα συναλλαγή αν υπάρχει. Διαφορετικά, πετάει εξαίρεση.
   - Requires New: δημιουργεί καινούρια συναλλαγή και αναβάλλει την τρέχουσα συναλλαγή αν υπάρχει.
   - Not Supported: εκτελείται χωρίς να δημιουργεί συναλλαγή και αναβάλλει την τρέχουσα συναλλαγή αν υπάρχει.

- Never: εκτελείται χωρίς να δημιουργεί συναλλαγή και πετάει εξαίρεση αν υπάρχει τρέχουσα συναλλαγή.
- Nested: εκτελείται σε φωλιασμένη συναλλαγή(αν υποστηρίζεται), διαφορετικά λειτουργεί σαν το Required.

3. Isolation: το επίπεδο απομόνωσης της συναλλαγής. Οι επιτρεπτές τιμές είναι
   - Default(εξ' ορισμού τιμή): χρήση του εξ' ορισμού επιπέδου απομόνωσης. Στην MySQL το εξ' ορισμού επίπεδο είναι Repeatable Read.
   - Read Uncommitted: επιτρέπει dirty reads, non-repeatable reads και phantom reads.
   - Read Committed: αποφυγή των dirty reads. Non-repeatable reads και phantom reads επιτρέπονται.
   - Repeatable Read: αποφυγή των dirty reads και non-repeatable reads. Phantom reads επιτρέπονται.
   - Serializable: αποφυγή των dirty reads,non-repeatable reads και phantom reads.
4. Timeout – το timeout της συναλλαγής σε δευτερόλεπτα. Εξ' ορισμού, χρησιμοποιεί την τιμή που έχει οριστεί στη βάση.
5. ReadOnly(εξ' ορισμού false) ορίζει αν ο κώδικας που θα εκτελεστεί μέσα στην συναλλαγή θα κάνει μόνο ανάγνωση δεδομένων. Αυτό λειτουργεί ως υπόδειξη για την τρέχουσα τεχνολογία και μπορεί να οδηγήσει σε βελτιστοποίηση της διαχείρισης της συναλλαγής. Για παράδειγμα, στην περίπτωση της εργασίας όπου χρησιμοποιώ JPA/Hibernate για τη διαχείριση των δεδομένων, readOnly μπορεί να ερμηνευθεί από το JPA/Hibernate για να απενεργοποιηθεί το dirty checking[42] αφού με το readOnly = true υποδεικνύω ότι η συναλλαγή δεν θα αλλάξει δεδομένα.
6. rollbackFor: δηλώνει τις classes των εξαιρέσεων που προκαλούν rollback της συναλλαγής.
7. rollbackForClassName: παρόμοια λειτουργία με rollbackFor, μόνο που δήλωση των εξαιρέσεων γίνεται μέσα από string πίνακα.
8. noRollbackFor: δηλώνει τις classes των εξαιρέσεων που δεν πρέπει να προκαλούν rollback της συναλλαγής. Δηλαδή, αν ο κώδικας που εκτελείται μέσα στην συναλλαγή πετάξει μία από τις δηλωμένες εξαιρέσεις,ο διαχειριστής συναλλαγών θα εκτελέσει commit.
9. noRollabackForClassName: παρόμοια με rollbackForClassName, αλλά για noRollbackFor.

Ακολουθεί παράδειγμα από την εργασία

```
@Service
@Transactional
class HotelServiceImpl(
   private val hotelRepository: HotelRepository,
   private val roomService: RoomService,
   private val scrappedHotelRepository: ScrappedHotelRepository,
) : HotelService {

   @Transactional(readOnly = true)
   override fun getHotelInfo(hotelInfoRequest: HotelInfoRequest): Optional<HotelInfo> {
         ....
   }
```

```
    override fun createHotel(hotelData: HotelData): Hotel = with(hotelData) {
        ....
    }

    override fun updateHotel(hotelData: HotelData): Hotel {
        ....
    }
}
```

- @Transactional πάνω στην class ορίζει τις ιδιότητες συναλλαγής για όλες τις μεθόδους και τα @Transactional στη μέθοδο αντικαθιστούν ιδιότητες συναλλαγής που έχουν οριστεί σε επίπεδο της class.
- Το @Transactional στην class δεν δηλώνει attributes οπότε ισχύουν οι εξ' ορισμού τιμές.
- Το @Transactional στην μέθοδο getHotelInfo δηλώνει readOnly attribute με τιμή true που σημαίνει ότι ο κώδικας της μεθόδου θα κάνει μόνο ανάγνωση δεδομένων.

2.7.2 Spring Boot

Το Spring Boot είναι βιβλιοθήκη που μειώνει δραστικά τον χρόνο που απαιτείται για το configuration των beans προσφέροντας starters και auto-configuration classes.

Τα auto-configuration classes είναι Spring configuration classes που περιέχουν "reasonable defaults" για configuration των διαφόρων beans και επιτρέπουν να τρέξει κάποιος την εφαρμογή με ελάχιστο ή καθόλου Spring configuration.

Starters είναι dependencies τα οποία περιέχουν οτιδήποτε χρειάζεται ένας προγραμματιστής για αρχίσει να δουλεύει με κάποια τεχνολογία. Για παράδειγμα, το spring-boot-starter-web περιέχει όλα τα dependencies που χρειάζονται για να αρχίσει κάποιος την ανάπτυξη web εφαρμογής χρησιμοποιώντας Spring MVC.

Τα auto-configuration classes του Spring Boot χρησιμοποιούν @ConfigurationProperties και conditional bean definition registration.

2.7.2.1 Spring Boot Properties

Για τον προσδιορισμό των ιδιοτήτων των beans τα οποία δημιουργούνται από auto-configuration classes, το Spring Boot χρησιμοποιεί ιδιότητες εφαρμογής. Για παράδειγμα, το DataSourceAutoConfiguration το οποίο δημιουργεί το DataSource bean που χρησιμοποιείται για δημιουργία σύνδεσης στη βάση δεδομένων επιτρέπει τον ορισμό των username/password για την σύνδεση μέσα από τις ιδιότητες της εφαρμογής.

2.7.2.2 Spring Boot Conditional Bean Registration

Η εγγραφή των beans τα οποία δημιουργούνται από auto-configuration classes του Spring Boot συχνά εξαρτάται από διαφορετικές συνθήκες. Για παράδειγμα,

1. Αν υπάρχει ήδη το bean ίδιου τύπου στο context. Ελέγχοντας ως συνθήκη την ύπαρξη του bean ίδιου τύπου,το Spring Boot επιτρέπει τους προγραμματιστές να κάνουν override τα beans που δημιουργούν οι auto-configuration classes με τα beans τα οποία ορίζονται από configuration αρχεία.

2. Ύπαρξη συγκεκριμένης class στο classpath. Για παράδειγμα, το DataSourceAutoConfiguration μπορεί να δημιουργήσει διαφορετικές υλοποιήσεις του DataSource(hikari,tomcat,dbcp2,oracle UCP) αναλόγως με το ποίες βιβλιοθήκες υπάρχουν στο classpath.

2.7.3 Spring MVC

Το Spring MVC είναι framework που χρησιμοποιείται για τη υλοποίηση web εφαρμογών πάνω στο servlet specification.

Το Spring MVC υλοποιεί το front controller design pattern όπου,αντί για τη δημιουργία πολλαπλών servlets με καθένα να διαχειρίζεται διαφορετικά request, χρησιμοποιείται ένα κεντρικό servlet για τη διαχείριση όλων των request της εφαρμογής, και η επεξεργασία των request γίνεται από handlers που είναι ορισμένη στον front controller/servlet. Το πλεονέκτημα αυτού του design pattern είναι η ύπαρξη κεντρικού σημείου στην επεξεργασία των request όπου μπορεί να υλοποιηθεί κώδικας κοινός για όλα τα requests.

Στο Spring MVC, το servlet που διαχειρίζεται όλα τα requests είναι το DispatcherServlet, και οι handlers των request είναι beans τα οποία έχουν @Controller annotation.

2.7.3.1 Spring MVC Handlers

Παρακάτω ένα παράδειγμα του HotelController από την εργασία.

@RestController είναι annotation που ενώνει το @Controller annotation και @ResponseBody annotation.

Το @ResponseBody δηλώνει ότι το αποτέλεσμα της handler μεθόδου αποτελεί το σώμα του response, ενώ το @Controller χρησιμοποιείται για τον ορισμό της class που έχει request handlers μεθόδους.

Οι μέθοδοι των controllers χρησιμοποιούν το @RequestMapping annotation για request matching. Τα attributes του @RequestMapping annotation δηλώνουν τα requests τα οποία επεξεργάζεται η μέθοδος. Τα υποστηριζόμενα attributes είναι:

1. value/path: ταιριάζει το URI path του request με το δηλωμένο string.
2. method: ταιριάζει την HTTP μέθοδο του request(GET,POST κτλ) με την δηλωμένη τιμή.
3. params: ταιριάζει τις παραμέτρους του request με τα string expressions που έχουν την μορφή *παραμ=τιμή ή παραμ!=τιμή* ή !παραμ(για μη ύπαρξη της παραμέτρου).
4. headers: το ίδιο με το params αλλά για http headers.
5. consumes: ταιριάζει τα δηλωμένα media types με την Content-Type κεφαλίδα του request.
6. produces: ταιριάζει τα δηλωμένα media types με την Accept κεφαλίδα του request.

Το Spring MVC προσφέρει επιπλέον annotations για συχνά χρησιμοποιούμενα mappings: @PostMapping για POST μέθοδο, @GetMapping – για GET κτλ.

Το @RequestMapping(και τα αντίστοιχα annotations για συγκεκριμένες HTTP μεθόδους) τοποθετείται πάνω στην class ή/και μέθοδο.

Κατά τη διαδικασία του request matching,πρώτα ελέγχεται το @RequestMapping της class και μόνο μετά των μεθόδων. Έτσι, μπορεί να δηλωθεί το βασικό mapping στο @RequestMapping της class, και πιο συγκεκριμένες τιμές στα @RequestMapping των μεθόδων. Στο παρακάτω παράδειγμα, το @RequestMapping της class δηλώνει ότι όλοι οι handlers της class διαχειρίζονται τα request τα οποία περιέχουν /hotel path στο URL.

Έτσι η μέθοδος hotelInfo διαχειρίζεται τα requests που έχουν /hotel/info path, Content-Type κεφαλίδα με τιμή application/json και Accept κεφαλίδα με τιμή application/json.

```
@RequestMapping("/hotel")
```

```
@RestController
class HotelController(private val hotelService: HotelService,private val scrappingService: ScrappingService,private val scrapperDataPersistService: ScrapperDataPersistService) {


@PostMapping("/info", consumes = [MediaType.APPLICATION_JSON_VALUE], produces = [MediaType.APPLICATION_JSON_VALUE])
   fun hotelInfo(@RequestBody hotelInfoRequest: HotelInfoRequest): ResponseEntity<HotelInfo> {
      ...
   }
}
```

2.7.3.2 Spring MVC εξαιρέσεις

Το Spring MVC επιτρέπει τον ορισμό των μεθόδων που διαχειρίζονται τις εξαιρέσεις που δημιουργούνται κατά την επεξεργασία του request από handler μέθοδο. Οι μέθοδοι αυτοί πρέπει να έχουν το @ExceptionHandler annotation το οποίο δηλώνει την εξαίρεση που μπορεί να χειριστεί η μέθοδος.

Οι μέθοδοι που διαχειρίζονται τις εξαιρέσεις μπορούν να τοποθετηθούν μέσα στην @Controller class ή μέσα στην class που έχει το @ControllerAdvice annotation. Η διαφορά είναι στο ότι η μέθοδος μέσα στην @Controller class διαχειρίζεται τις εξαιρέσεις που μπορεί να πετάξει κάποια handler μέθοδος της controller class, ενώ οι μέθοδοι μέσα στην @ControllerAdvice class ισχύουν για όλους τους handlers ανεξάρτητα από τον controller μέσα στον οποίο έχουν δηλωθεί.

Στην εργασία χρησιμοποίησα @ControllerAdvice class για τον χειρισμό των εξαιρέσεων.

2.7.4 Spring Hateoas

Spring Hateoas είναι βιβλιοθήκη η οποία διευκολύνει τη δημιουργία REST εφαρμογών τα οποία ακολουθούν τις αρχές του HATEOAS.

Η εφαρμογή που ακολουθεί τις αρχές του HATEOAS επιτρέπει στις client εφαρμογές που αλληλεπιδρούν με την εφαρμογή δυναμικά να μαθαίνουν τις δυνατότητες που προσφέρονται.

Ακολουθεί response παράδειγμα από την εργασία το οποίο ακολουθεί την HATEOAS αρχή και ορίζει επιπλέον resources που μπορεί να χρησιμοποιήσει ο client και τη σχέση που έχουν αυτά με το resource που έχει ζητηθεί.

Το response περιέχει τις τιμές για δωμάτιο και εμπεριέχει links για επιπλέον resources. Συγκεκριμένα, περιέχει τα εξής links

1. first: δηλώνει το resource το οποίο επιστρέφει την πρώτη σελίδα τιμών του δωματίου.
2. self: δηλώνει το resource το οποίο επιστρέφει το ίδιο response.
3. next: δηλώνει το resource το οποίο αποτελεί την επόμενη σελίδα τιμών του δωματίου.
4. last: δηλώνει το resource το οποίο αποτελεί την τελευταία σελίδα τιμών του δωματίου.

Έτσι, ένας client μπορεί να διαβάσει το περιεχόμενο του response και να καταλάβει ποιά άλλα resources μπορεί να χρησιμοποιήσει.

Η HATEOAS αρχή χρησιμοποιήθηκε στην εργασία μόνο για pagination και δεν ακολουθείται για όλα τα endpoints της εφαρμογής.


```
{
 "_embedded":{
  "priceInfoList":[
   {
    "id":1143,
    "sleeps":2,
    "price":451,
    "quantity":4,
    "distanceDays":9,
    "cancellationPolicy":"UNKNOWN",
    "timestamp":"2023-08-14T13:02:01.703918",
    "breakfastPolicy":"unknown",
    "attributes":[
     "TEST_ROOM_VIEW_ATTR_2",
     "TEST_ROOM_VIEW_ATTR_3"
    ]
   },
   {
    "id":1141,
    "sleeps":1,
    "price":771,
    "quantity":3,
    "distanceDays":2,
    "cancellationPolicy":"UNKNOWN",
    "timestamp":"2023-08-14T13:02:01.682616",
    "breakfastPolicy":"unknown",
    "attributes":[
     "TEST_ROOM_VIEW_ATTR_1",
     "TEST_ROOM_VIEW_ATTR_2"
    ]
   }
  ]
 },
```

```
"_links":{
  "first":{
    "href":"http://localhost:8085/prices/570?page=0&size=2"
  },
  "self":{
    "href":"http://localhost:8085/prices/570?page=0&size=2"
  },
  "next":{
    "href":"http://localhost:8085/prices/570?page=1&size=2"
  },
  "last":{
    "href":"http://localhost:8085/prices/570?page=1&size=2"
  }
},
"page":{
  "size":2,
  "totalElements":4,
  "totalPages":2,
  "number":0
}
}
```

2.7.5 Spring Security

Χρησιμοποιήθηκε για configuration της ασφάλειας της εφαρμογής. Ακολουθούν τα βασικά συστατικά του Spring Security.

2.7.5.1 Authentication

Η class που υλοποιεί Authentication interface αποτελεί είτε το διαπιστευτήριο αυθεντικοποίησης (email και password,bearer token,secret key κτλ.) είτε αυθεντικοποιημένη οντότητα.

2.7.5.2 Authentication Manager

Η αρμοδιότητα της class που υλοποιεί AuthenticationManger interface είναι η αυθεντικοποίηση. Η υλοποίηση που χρησιμοποιήθηκε στην εφαρμογή είναι ProviderManager[13] που υλοποιεί την αυθεντικοποίηση χρησιμοποιώντας αντικείμενα των classes που υλοποιούν AuthenticationProvider interface. Το πλεονέκτημα του ProviderManager είναι ότι επιτρέπει την εφαρμογή να έχει περισσότερα από ένα τρόπο αυθεντικοποίησης.

2.7.5.3 SecurityContext

Container που αποθηκεύει το Authentication αντικείμενο που αναπαριστά τον αυθεντικοποιημένο χρήστη.

2.7.5.4 SecurityContextHolder

Container που αποθηκεύει το SecurityContext container.

Υπάρχουν διαφορετικές στρατηγικές για την αποθήκευση του SecurityContext. Η στρατηγική που χρησιμοποιήθηκε στην εργασία είναι η αποθήκευση του SecurityContext σε ThreadLocal μεταβλητή με αποτέλεσμα το SecurityContext να σχετίζεται με το νήμα εκτέλεσης. Αυτή είναι η εξ' ορισμού στρατηγική αποθήκευσης του SecurityContext καθώς επιτρέπει την συσχέτιση της αυθεντικοποιημένης οντότητας με το νήμα που διαχειρίζεται το request, για παράδειγμα.

2.7.5.5 GrantedAuthority

Αποτελεί κάποιο δικαίωμα που έχει ο χρήστης. Για παράδειγμα, μπορεί να αποτελεί κάποιο ρόλο του χρήστη ή δικαίωμα για κάποια ενέργεια.

2.7.5.6 AuthorizationManager

Η class που υλοποιεί αυτό το interface παίρνει την απόφαση αν ο χρήστης έχει δικαίωμα χρήσης κάποιου προστατευμένου αντικειμένου/πόρου. Το προστατευμένο αντικείμενο μπορεί να είναι μέθοδος κάποιας class,controller endpoint.

2.7.5.7 Spring MVC Integration

Το Spring Security προσφέρει integration με το Spring MVC χρησιμοποιώντας Servlet φίλτρα.

Το Spring Security προσθέτει το security σε spring mvc εφαρμογή χρησιμοποιώντας DelegatingFilterProxy και FilterChainProxy φίλτρα. Το DelegatingFilterProxy είναι φίλτρο το οποίο κάνει delegation σε άλλο φίλτρο που έχει οριστεί ως bean στο Spring context. Το FilterChainProxy είναι το φίλτρο το οποίο ορίζεται ως bean στο Spring context και το οποίο χρησιμοποιεί SecurityFilterChain beans για την εφαρμογή της ασφάλειας.

Το SecurityFilterChain interface δηλώνει δύο μεθόδους:

1. matches. Καλείται για κάθε request της εφαρμογής και δηλώνει αν το η αλυσίδα των φίλτρων ασφάλειας πρέπει να εφαρμοστεί για το request.
2. getFilters. Επιστρέφει τα φίλτρα της ασφάλειας από τα οποία πρέπει να περάσει το request όταν matches επιστρέφει true.

```
public interface SecurityFilterChain {
	boolean matches(HttpServletRequest request);
	List<Filter> getFilters();
}
```

Ακολουθεί το securityFilterChain bean definition του SecurityFilterChain της εφαρμογής το οποίο δηλώνει τα εξής χαρακτηριστικά της ασφάλειας:

1. anonymous: προσθέτει AnonymousAuthenticationFilter στο SecurityFilterChain. Αυτό το φίλτρο προσθέτει Authentication αντικείμενο που δηλώνει τον χρήστη στο security context όταν ο χρήστης δεν είναι αυθεντικοποιημένος. Αυτό έχει το πλεονέκτημα ότι το SecurityContext δεν είναι ποτέ κενό.

2. csrf: δηλώνει το κομμάτι του security configuration που αφορά το csrf. Το csrf είναι απενεργοποιημένο και αποτελεί ένα κομμάτι στο οποίο μπορεί να γίνει βελτίωση.
3. authorizeHttpRequests.Δηλώνει τους κανόνες εξουσιοδότησης: ποιο χρήστες έχουν δικαίωμα να κάνουν request στα ορισμένα endpoints. Αυτοί οι κανόνες χρησιμοποιούνται από το AuthorizationFilter για εξουσιοδότηση.
4. sessionManagement: δηλώνει το κομμάτι του security που αφορά το session management. Στην εφαρμογή θέτω stateless session management για να μη δημιουργείται session. Συγκεκριμένα, προσθέτει SessionManagementFilter φίλτρο το οποίο αποθηκεύει τον authenticated χρήστη μεταξύ των request. Επειδή δηλώνω stateless session management, το security context του χρήστη δεν θα αποθηκεύεται μεταξύ των request.
5. Oauth2ResourceServer: δηλώνει το κομμάτι του security configuration που αφορά το oauth2 resource server. Στο OAuth 2.0, resource server[14] είναι ο server που διαχειρίζεται τα resources.

   Το configuration προσθέτει BearerTokenAuthenticationFilter το οποίο εκτελεί bearer token authentication αναζητώντας bearer token στις HTTP κεφαλίδες και κάνοντας authentication χρησιμοποιώντας JwtAuthenticationProvider ο οποίο ελέγχει ότι το JWT έχει σωστή υπογραφή και δεν έχει λήξει.

```
@Bean
    fun securityFilterChain(http: HttpSecurity,decoder: JwtDecoder,customerService: CustomerService) : SecurityFilterChain =
        http
            .anonymous {}
            .csrf { csrf -> csrf.disable() }
            .logout { logoutConfig -> logoutConfig.disable() }
            .authorizeHttpRequests { authorization ->
                authorization
                        .requestMatchers(HttpMethod.POST,"/customer/login","/customer/register","/customer/email/verification","/customer/password/reset").permitAll()
                    .requestMatchers(HttpMethod.PUT,"/customer/password").permitAll()
                    .requestMatchers(HttpMethod.POST,"/scrapper/login").permitAll()
                    .requestMatchers(HttpMethod.GET,"/","/register","/login","/logout","/test").permitAll()
                         .requestMatchers("/customer/**","/hotel/info","/monitor_list/**","/prices/**","/room/history/**").hasAnyRole(Role.USER.name,Role.BASIC.name,Role.PREMIUM.name)
                    .requestMatchers(HttpMethod.POST,"/scrap").permitAll()
                    .anyRequest().permitAll()
            }
            .sessionManagement { session ->
                session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
             }.securityContext { context -> context.securityContextRepository(NullSecurityContextRepository())
}
            .oauth2ResourceServer { resourceServer ->
```

```
        resourceServer.jwt { jwt ->
          jwt.decoder(decoder)
                                        jwt.jwtAuthenticationConverter(JwtAuthenticationConverter().apply {
setJwtGrantedAuthoritiesConverter(JwtGrantedAuthoritiesConverter().apply {
setAuthorityPrefix("ROLE_") }) })
        }
 }.build()
```

2.7.6 Spring AMQP

Επιτρέπει τη ανάπτυξη εφαρμογών που χρησιμοποιούν το AMQP.

2.7.7 Spring Test

Διευκολύνει την συγγραφή των unit και integration tests για Spring εφαρμογές. Περισσότερα στο Testing.

2.7.8 Spring Data JPA

Επιτρέπει τη δημιουργία Spring Data Repositories που χρησιμοποιούν JPA για υλοποίηση των Repository μεθόδων.

Spring Data Repositories είναι interfaces τα οποία κληρονομούν από Repository interface.

Έκτος από Repository interface, το Spring Data προσφέρει άλλα interfaces που κληρονομούν από Repository και προσφέρουν κάποιες βασικές μεθόδους ανάκτησης δεδομένων ανεξάρτητα από την τεχνολογία που χρησιμοποιείται. Έτσι, το Spring Data προσφέρει

1. CrudRepository προσφέρει μεθόδους για CRUD(Create,Read,Update,Delete).
2. PaginationAndSortingRepository προσφέρει μεθόδους που επιτρέπουν σελιδοποίηση και ταξινόμηση των αποτελεσμάτων.

Το Spring Data {*Τεχνολογία*} εντοπίζει τα interfaces που κληρονομούν από Repository interface και τα υλοποιεί χρησιμοποιώντας την επιλεγμένη τεχνολογία. Στην εργασία χρησιμοποίησα Spring Data JPA.

Το Spring Data υλοποιεί τα interfaces ερμηνεύοντας τις μεθόδους των interfaces:

1. Μπορεί να δημιουργήσει υλοποίηση μεθόδου αν το όνομα της μεθόδου ακολουθεί ορισμένο format υποστηριζόμενο από την βιβλιοθήκη. Ακολουθεί παράδειγμα από την εργασία
   ```
   interface CustomerRepository : JpaRepository<Customer,Long>{
     fun findByEmail(email: String) : Optional<Customer>
   }
   ```
   Για την μέθοδο findByEmail θα δημιουργήσει υλοποίηση η οποία θα χρησιμοποιήσει SQL ερώτημα για τον πίνακα της Customer οντότητας με WHERE κομμάτι που περιέχει τον περιορισμό στο email πεδίο να είναι ίσο με την τιμή της παραμέτρου της μεθόδου. Τέλος, επειδή ο τύπος επιστροφής της μεθόδου είναι Optional<Customer>, η υλοποίηση θα επιστρέψει κενό Optional αν το αποτέλεσμα του ερωτήματος δεν θα βρει τον Customer και το Optional[3] με το Customer αν το ερώτημα θα επιστρέψει τον Customer.
2. Ένας δεύτερος τρόπος για προσδιορισμό του ερωτήματος αναζήτησης είναι χρησιμοποιώντας το @Query annotation. Ακολουθεί παράδειγμα από την εργασία

3 Optional είναι value container που μπορεί να είναι κενός ή να περιέχει τιμή.

```
interface RoomViewRepository : JpaRepository<RoomView,Long>,
RevisionRepository<RoomView,Long,Int>{
@Lock(LockModeType.PESSIMISTIC_WRITE)
 @Query("SELECT r FROM RoomView r WHERE r.sleeps = :sleeps AND r.cancellationPolicy
= :cancellationPolicy AND r.breakfast = :breakfast AND r.room = :room")
    fun findRoomViewBySleepsAndCancellationPolicyAndBreakfastAndRoom(sleeps: Int,
cancellationPolicy: CancellationPolicy, breakfast: String, room: Room) : Optional<RoomView>
}
```

Για αυτή τη μέθοδο το Spring Data θα χρησιμοποιήσει την τιμή του @Query annotation. Επίσης, επειδή χρησιμοποίησα το @Lock annotation με pessimistic write mode, το Spring Data θα χρησιμοποιήσει pessimistic locking: στην περίπτωση της RDBMS/MySQL, θα χρησιμοποιήσει FOR UPDATE στο τέλος του ερωτήματος.

Το Spring Data προσφέρει κάποιες κοινές λειτουργίες και είναι ανεξάρτητο της τεχνολογίας που χρησιμοποιείται για τη διαχείριση των δεδομένων. Για κάθε ξεχωριστή τεχνολογία υπάρχει αντίστοιχή επέκταση. Επειδή χρησιμοποίησα JPA για διαχείριση των δεδομένων, έχω προσθέσει το Spring Data JPA που υλοποιεί τα αντίστοιχα interfaces χρησιμοποιώντας το JPA API και προσφέρει επιπλέον interface(JpaRepository) με δυνατότητες που υπάρχουν μόνο όταν χρησιμοποιείται το JPA API.

2.7.9 Spring Validation

Προσφέρει εργαλεία για ανάπτυξη του validation των αντικειμένων. Περισσότερα στο Hibernate Validation.

2.8 Hibernate

Το Hibernate αποτελεί και αυτό μία οικογένεια από frameworks/βιβλιοθήκες. Από όλα τα εργαλεία που προσφέρει το Hibernate,στην εργασία χρησιμοποιήθηκαν το Hibernate ORM και Hibernate Validation.

2.8.1 Hibernate ORM

Αποτελεί μία από τις υλοποιήσεις του JPA προτύπου. Το Hibernate ORM προσφέρει αυτόματο mapping μεταξύ των αντικειμένων των classes και των πινάκων της βάσης δεδομένων. Το mapping μεταξύ των classes και πινάκων μπορεί να οριστεί με δύο τρόπους:

1. Χρησιμοποιώντας JPA/Hibernate annotations.
2. Χρησιμοποιώντας XML αρχείο.

Στην εργασία έχω προσδιορίσει το mapping χρησιμοποιώντας JPA/Hibernate annotations.

2.8.1.1 Entity

Hibernate Entity είναι class που έχει @Entity annotation. Κάθε Entity class αντιστοιχεί σε πίνακα βάσης και τα πεδία της αντιστοιχούν σε στήλες του πίνακα ή πίνακες.

Ακολουθεί παράδειγμα Entity class από την εργασία

```
@Check(constraints = "hotel_score IS NULL OR (hotel_score >= 0 AND hotel_score <= 10.0)")
@Entity
```

```
@Table(name = "hotels", uniqueConstraints = [UniqueConstraint(name = HOTELS_URL_UNIQUE_CONSTRAINT, columnNames = ["hotel_url"])])
class Hotel(
  @Column(nullable = false, name = "hotel_url")
  var url: URL,
  @Column(name = "hotel_name", nullable = false)
  var name: String,
  @Column(name = "hotel_description", length = 1000, nullable = false)
  var description: String,
  @AttributeOverrides(
    AttributeOverride(name = "address", column = Column(nullable = false)),
    AttributeOverride(name = "city", column = Column(nullable = false)),
    AttributeOverride(name = "longitude", column = Column(nullable = true)),
    AttributeOverride(name = "latitude", column = Column(nullable = true))
  )
  @Column
  var location: Location,
  @Enumerated(EnumType.STRING)
  @Column(name = "hotel_type", nullable = false)
  var hotelType: HotelType,
  @Column(name = "hotel_score", scale = 2, precision = 4, nullable = false)
  var score: BigDecimal,
  @Column(name = "id")
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  var id: Long? = null
)
```

- @Check annotation χρησιμοποιείται για προσδιορισμό SQL CHECK περιορισμών για τον πίνακα.
- @Table annotation χρησιμοποιείται για τον προσδιορισμό του ονόματος του πίνακα,των indexes και unique περιορισμών του πίνακα.
- @Column annotation χρησιμοποιείται για προσδιορισμό του mapping για την στήλη στην οποία αντιστοιχεί το πεδίο της class.
- @AttributeOverrides χρησιμοποιείται για αλλαγή του mapping των στηλών των Embeddable classes.
- @Enumarted χρησιμοποιείται για τον προσδιορισμό της στρατηγικής αποθήκευσης των τιμών των enum classes. Υπάρχουν δύο στρατηγικές: STRING και ORDINAL. To STRING αποθηκεύει το

όνομα του enum αντικειμένου, ενώ το ORDINAL αποθηκεύει το index του αντικειμένου το οποίο εξαρτάται από την σειρά των enum αντικειμένων της class.

- @Id χρησιμοποιείται για τον προσδιορισμό του primary key του πίνακα/οντότητας.
- @GeneratedValue προσδιορίζει τον τρόπο παραγωγής της τιμής του primary key.

Ορισμένα πεδία της Entity class μπορούν να αντιστοιχούν σε πίνακες όταν το πεδίο προσδιορίζει σχέση με μία ή περισσότερες οντότητες ή όταν το πεδίο έχει τύπο collection απλών τιμών. Παρακάτω ένα παράδειγμα από την εργασία όπου το πεδίο attributes αντιστοιχεί σε πίνακα και αποτελεί σχέση μεταξύ της οντότητας Room και της οντότητας RoomAttribute.

```
@Entity
@Table(name = "hotel_rooms", uniqueConstraints = [UniqueConstraint(name = HOTEL_ROOMS_UNIQUE_NAME_CONSTRAINT, columnNames = ["hotel_room_name","hotel_id"])])
class Room(
   @JoinColumn(name = "hotel_id")
   @ManyToOne(optional = false)
   var hotel: Hotel,

   @Column(name = "hotel_room_description", nullable = false, length = 10000)
   var description: String,
   @Column(name = "hotel_room_name", nullable = false)
   var name: String,
   @Audited(targetAuditMode = RelationTargetAuditMode.NOT_AUDITED)
         @JoinTable(name = "hotel_rooms_owned_attributes", joinColumns = [JoinColumn(name = "hotel_room_id")], inverseJoinColumns = [JoinColumn(name = "room_attribute_id")])
   @ManyToMany
   var attributes: MutableSet<RoomAttribute> = mutableSetOf(),
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   var id: Long? = null
)
```

@ManyToMany annotation προσδιορίζει many-to-many σχέση.

2.8.1.2 Persistent Context

Τα αντικείμενα των Entity classes δεν έχουν καμία διαφορά από τα αντικείμενα οποιασδήποτε άλλης class. Τα αντικείμενα τα οποία διαχειρίζεται και ελέγχει το hibernate ονομάζονται persistent entities. Για την μετατροπή ενός αντικείμενου σε persistent entity, πρέπει αυτό να συσχετιστεί με persistent context.

Persistent context είναι μέρος του hibernate στο οποίο αποθηκεύονται τα αντικείμενα τα οποία φορτώνονται από τη βάση στην εφαρμογή ή από την εφαρμογή στη βάση. Το persistent context λειτουργεί ως cache: όταν η εφαρμογή ζητάει το hibernate να φορτώσει κάποια οντότητα από τη βάση και η οντότητα υπάρχει ήδη στο persistent context, τότε το hibernate επιστρέφει την αποθηκευμένη οντότητα χωρίς να εκτελεί SQL εντολές.

Ακολουθεί παράδειγμα όπου χρησιμοποιείται μέθοδος της HotelRepository για αναζήτηση ξενοδοχείου βάση id. Η πρώτη κλήση προκαλεί SQL ερώτημα για τη φόρτωση του αντικειμένου όπως φαίνεται από τα logs του Figure 10: Persistent Context Cache Miss(τελευταίο log της πρώτης εικόνας), ενώ η δεύτερη κλήση δεν προκαλεί καινούριο ερώτημα και το hibernate επιστρέφει το αντικείμενο που είναι αποθηκευμένο στο persistent context όπως φαίνεται από τα logs του Figure 9: Persistent Context Cache Hit(τα τρία τελευταία logs της δεύτερης εικόνας δεν περιέχουν SQL ερώτημα).

Figure 9: Persistent Context Cache Hit

Figure 10: Persistent Context Cache Miss

Για τα persistent entities το hibernate προσφέρει ορισμένες δυνατότητες. Κάποιες από αυτές είναι:

1. Dirty Checking. Στο τέλος της SQL συναλλαγής και πριν κλίσει το persistent context, το hibernate ελέγχει όλα τα αντικείμενα που υπάρχουν στο context για να βρει αυτά που έχουν αλλάξει. Για αυτά τα αντικείμενα, εκτελεί SQL UPDATE.

2. Lazy Loading. Όταν φορτώνονται τα entities από τη βάση,το hibernate φορτώνει όλες τις ιδιότητες εκτός από τις ιδιότητες που έχουν τύπο collections. Αν το entity σχετίζεται με persistent context, η φόρτωση του πεδίου που αναπαριστά το collection θα γίνει κατά την πρώτη πρόσβαση στο πεδίο χρησιμοποιώντας ένα επιπλέον ερώτημα.

Παρακάτω το Figure 11: Dirty Check Example δείχνει στην πράξη τη λειτουργία του dirty checking. Ο κώδικας αποθηκεύει δύο οντότητες στη βάση και μετά την αποθήκευση αλλάζει url πεδίο του δεύτερου αντικειμένου. Μετά το τέλος της SQL συναλλαγής, το hibernate αναγνωρίζει ότι ένα από τα αντικείμενα έχει

Figure 11: Dirty Check Example

αλλάξει και εκτελεί SQL update όπως φαίνεται στα logs.

Το επόμενο Figure 12: Lazy Collection Loading Example δείχνει στην πράξη τη λειτουργία του lazy loading. Πρώτα φορτώνουμε Room αντικείμενο από βάση χρησιμοποιώντας το id. Αυτό οδηγεί στην εκτέλεση SQL ερωτήματος(4 log από το τέλος). Τέλος ο κώδικας καλεί println για να εμφανίσω τα attributes του δωματίου, και αυτό οδηγεί σε επιπλέον SQL ερώτημα το οποίο φορτώνει τα attributes του συγκεκριμένου δωματίου(2 log από το τέλος).

Figure 12: Lazy Collection Loading Example

Όταν δημιουργείται καινούριο αντικείμενο της Entity class, αυτό βρίσκεται σε transient state που σημαίνει ότι δεν συσχετίζεται με το persistent context και δεν αντιστοιχεί σε καμία γραμμή του πίνακα.

Για να συσχετίσω το αντικείμενο με το persistent context, πρέπει να το αποθηκεύσω χρησιμοποιώντας merge/persist μέθοδο ή,στην περίπτωση όπου χρησιμοποιώ Spring Data JPA Repository, save μέθοδο του repository. Το αντικείμενο που επιστρέφεται από τη μέθοδο είναι persistent entity το οποίο αντιστοιχεί σε γραμμή του πίνακα.

Μετά το τέλος της συναλλαγής το persistent context κλίνει και όλα τα persistent entities θεωρούνται απομονωμένα(detached). Ένα detached entity μπορεί να συσχετιστεί με καινούριο persistent context ξανά.

2.8.1.3 HQL/JPQL

JPQL είναι γλώσσα ερωτημάτων(παρόμοια με SQL) όπου αντί για πίνακες χρησιμοποιούνται οντότητες. HQL είναι επέκταση της JPQL η οποία προσφέρει ίδιες δυνατότητες με JPQL και επιπλέον δυνατότητες που υποστηρίζονται μόνο από Hibernate ORM.

Το JPQL υποστηρίζει UPDATE,DELETE και SELECT[4][15].

Ακολουθεί παράδειγμα JPQL SELECT ερώτημα από την εργασία

```
interface PriceRepository : JpaRepository<RoomViewPrice,Long>{

    @Query("SELECT p FROM RoomViewPrice p JOIN FETCH p.roomView rv LEFT JOIN FETCH rv.attributes WHERE p.id in :ids")

   fun findByIdIn(ids: Collection<Long>) : List<RoomViewPrice>

   ...

}
```

- RoomViewPrice αποτελεί οντότητα. Δεν είναι όνομα πίνακα.
- SELECT επιστρέφει αντικείμενα τύπου RoomViewPrice.
- Το ερώτημα χρησιμοποιεί αναγνωριστικό για τις οντότητες RoomViewPrice και RoomView – p και rv αντίστοιχα.
- JOIN εκτελεί SQL INNER JOIN.
- Το JPQL επιτρέπει τη χρήστη του .(dot) για πρόσβαση στα πεδία των οντοτήτων.
- FETCH μετά το JOIN δεν αλλάζει το JOIN, αλλά οι επιστρεφόμενες οντότητες του ερωτήματος θα έχουν τα αντίστοιχα πεδία φορτωμένα με τα δεδομένα από το JOIN. Χωρίς FETCH στο παραπάνω παράδειγμα, το RoomViewPrice δεν θα είχε το roomView πεδίο φορτωμένο και πρόσβαση σε αυτό

4 Το INSERT πρέπει να γίνεται μέσω του JPA API.

το πεδίο μετά την εκτέλεση του ερωτήματος θα προκαλούσε επιπλέον ερώτημα για την φόρτωση του πεδίου.

Σχετικά με το επάνω σημείωμα, η εξ' ορισμού στρατηγική(Lazy Loading) του JPA/Hibernate που έχει αναφερθεί πριν δεν ισχύει. Στην περίπτωση που χρησιμοποιούμαι JPQL/HQL, όλα τα πεδία που εκπροσωπεύον σχέση με άλλες οντότητες ή συλλογές απλών τιμών/οντοτήτων και τα οποία θέλουμε να φορτώσουμε μαζί με το φόρτωμα των οντοτήτων πρέπει να προσδιοριστούν ρητά στο JPQL/HQL με JOIN FETCH/LEFT JOIN FETCH.

Παρακάτω τρία παραδείγματα. Το Figure 13: JOIN χωρίς FETCH παράδειγμα χρησιμοποιεί JOIN χωρίς FETCH με αποτέλεσμα να εκτελείται δεύτερο ερώτημα για φόρτωση του hotel πεδίου της room οντότητας·το Figure 14: JOIN με FETCH παράδειγμα χρησιμοποιεί JOIN FETCH με αποτέλεσμα το hotel πεδίο να είναι φορτωμένο στην οντότητα που επιστρέφει το ερώτημα· το Figure 15: To default fetch χωρίς JPQL/HQL παράδειγμα χρησιμοποιεί JPA API χωρίς JPQL/HQL οπότε ισχύει η εξ' ορισμού στρατηγική φόρτωσης οντοτήτων.

```
@Test
fun `example jpql without fetch`() {
    println("Start example jpql without fetch.")
    val room :Room! = entityManager.entityManager.createQuery( qlString: "SELECT r FROM Room r JOIN r.hotel WHERE r.id = $roomID",Room::class.java).singleResult
    println("Finish example jpql without fetch.")
}
```

```
Run   Test.example jpql without fetch
Tests passed: 1 of 1 test – 892 ms
Test Results 892 ms
  Test 892 ms
    examp 892 ms
2024-01-05T21:42:54.063+02:00 DEBUG 25019 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Opened new EntityManager [SessionImpl(1714673551<open>)] for JPA transaction
2024-01-05T21:42:54.063+02:00 DEBUG 25019 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Exposing JPA transaction as JDBC [org.springframework.orm.jpa.vendor.HibernateJpaDialect$Hibernate
Start example jpql without fetch.
2024-01-05T21:42:54.466+02:00 DEBUG 25019 --- [    Test worker] org.hibernate.SQL                        : select r1_0.id,r1_0.hotel_room_description,r1_0.hotel_id,r1_0.hotel_room_name from hotel_rooms r1_
2024-01-05T21:42:54.467+02:00 DEBUG 25019 --- [    Test worker] org.hibernate.SQL                        : select h1_0.id,h1_0.hotel_description,h1_0.hotel_type,h1_0.address,h1_0.city,h1_0.latitude,h1_0.lo
2024-01-05T21:42:54.468+02:00 TRACE 25019 --- [    Test worker] org.hibernate.orm.jdbc.bind              : binding parameter [1] as [BIGINT] - [1]
Finish example jpql without fetch.
```

Figure 13: JOIN χωρίς FETCH παράδειγμα

```
@Test
fun `example jpql with fetch`() {
    println("Start example jpql with fetch.")
    val room :Room! = entityManager.entityManager.createQuery( qlString: "SELECT r FROM Room r JOIN FETCH r.hotel WHERE r.id = $roomID",Room::class.java).singleResult
    println("Finish example jpql with fetch.")
}

@AfterTransaction
fun afterTransaction(){
    roomRepository.deleteAll()
    roomAttributeRepository.deleteAll()
    hotelRepository.deleteAll()
}
}
```

```
Run   Test.example jpql with fetch
Tests passed: 1 of 1 test – 843 ms
Test Results 843 ms
  Test 843 ms
    examp 843 ms
2024-01-05T21:48:03.175+02:00 DEBUG 28334 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Closing JPA EntityManager [SessionImpl(1772055258<open>)] after transaction
2024-01-05T21:48:03.175+02:00 DEBUG 28334 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Creating new transaction with name [iee.ihu.gr.pricemonitoringbackend.Test.example jpql with fetch
2024-01-05T21:48:03.175+02:00 DEBUG 28334 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Opened new EntityManager [SessionImpl(1680462311<open>)] for JPA transaction
2024-01-05T21:48:03.175+02:00 DEBUG 28334 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Exposing JPA transaction as JDBC [org.springframework.orm.jpa.vendor.HibernateJpaDialect$Hibernate
Start example jpql with fetch.
2024-01-05T21:48:03.531+02:00 DEBUG 28334 --- [    Test worker] org.hibernate.SQL                        : select r1_0.id,r1_0.hotel_room_description,r1_0.hotel_id,h1_0.id,h1_0.hotel_description,h1_0.hotel
Finish example jpql with fetch.
```

Figure 14: JOIN με FETCH παράδειγμα

```
@Test
fun `default fetch behaviour of jpa api`() {
    println("Start default fetch behaviour example.")
    val room :Room! = entityManager.entityManager.find(Room::class.java,roomID)
    println("Finish default fetch behaviour example.")
}

@AfterTransaction
fun afterTransaction(){
    roomRepository.deleteAll()
    roomAttributeRepository.deleteAll()
    hotelRepository.deleteAll()
```

```
Run   Test.default fetch behaviour of jpa api
Tests passed: 1 of 1 test – 917 ms
Test Results 917 ms
  Test 917 ms
    defaul 917 ms
2024-01-05T21:51:07.396+02:00 DEBUG 29729 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Closing JPA EntityManager [SessionImpl(1304876392<open>)] after transaction
2024-01-05T21:51:07.396+02:00 DEBUG 29729 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Creating new transaction with name [iee.ihu.gr.pricemonitoringbackend.Test.default fetch behaviour
2024-01-05T21:51:07.396+02:00 DEBUG 29729 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Opened new EntityManager [SessionImpl(1157538178<open>)] for JPA transaction
2024-01-05T21:51:07.396+02:00 DEBUG 29729 --- [    Test worker] o.s.orm.jpa.JpaTransactionManager        : Exposing JPA transaction as JDBC [org.springframework.orm.jpa.vendor.HibernateJpaDialect$Hibernate
Start default fetch behaviour example.
2024-01-05T21:51:07.760+02:00 DEBUG 29729 --- [    Test worker] org.hibernate.SQL                        : select r1_0.id,r1_0.hotel_room_description,r1_0.hotel_id,h1_0.id,h1_0.hotel_description,h1_0.hotel
2024-01-05T21:51:07.760+02:00 TRACE 29729 --- [    Test worker] org.hibernate.orm.jdbc.bind              : binding parameter [1] as [BIGINT] - [1]
Finish default fetch behaviour example.
```

Figure 15: To default fetch χωρίς JPQL/HQL

2.8.2 Hibernate Envers

Hibernate Envers επιτρέπει το auditing των οντοτήτων: αποθήκευση διαφορετικών εκδόσεων των οντοτήτων ή ,με άλλα λογία, αποθήκευση ιστορικού οντοτήτων.

Το hibernate envers λειτουργεί χρησιμοποιώντας hibernate listeners που είναι αντικείμενα τα οποία μπορούν να δέχονται events όπως όταν αποθηκεύεται καινούρια οντότητα.

Για να ξεκινήσει το auditing των οντοτήτων μιας Entity class απαιτείται η χρήστη της @Audited annotation

Το hibernate envers για τη λειτουργία του απαιτεί την ύπαρξη στη βάση του πίνακα REVINFO. Ο πίνακας πρέπει να έχει δύο στήλες

1. REV στήλη έχει integer τύπο και δηλώνει έκδοση.
2. REVTSTMP δηλώνει το timestamp της έκδοσης.

Αυτός ο πίνακας χρησιμοποιείται για παραγωγή επόμενης έκδοσης όταν αποθηκεύεται η οντότητα: είτε είναι INSERT είτε UPDATE.

Για κάθε audited Entity class, hibernate envers επίσης απαιτεί επιπλέον πίνακα με το ίδιο όνομα που έχει ο πίνακας στον οποίο αντιστοιχεί η οντότητα + _AUD suffix.

Σε αυτόν τον πίνακα το hibernate envers αποθηκεύει τις εκδόσεις των οντοτήτων. Συγκεκριμένα, όταν δημιουργείται ή αλλάζει οντότητα, το hibernate envers δημιουργεί καινούρια γραμμή στον αντίστοιχο auditing πίνακα με το id της οντότητας,τον αριθμό της έκδοσης(στήλη REV),τον αριθμό της επόμενης έκδοσης(στήλη REVEND,η στήλη περιέχει null όταν δεν υπάρχει επόμενη έκδοση),την αιτία δημιουργίας της έκδοσης(στήλη REVTYPE) και τις τιμές των πεδίων της οντότητας που είχε η οντότητα στην συγκεκριμένη έκδοση.

Επίσης, αν το πεδίο της οντότητας αντιστοιχεί σε πίνακα και όχι στήλη του πίνακα της οντότητας, τότε για αυτό το πεδίο απαιτείται επιπλέον πίνακας με το όνομα που αντιστοιχεί στο όνομα του πίνακα που χρησιμοποιείται για την αποθήκευση των σχέσεων ή τιμών + _AUD suffix.

Η στήλη REVTYPE στους auditing πίνακες μπορεί να πάρει τιμές 0,1,2 όπου

1. 0 δηλώνει ότι η αιτία δημιουργίας της έκδοσης ήταν insertion
2. 1 δηλώνει ότι η αιτία δημιουργίας της έκδοσης ήταν modification/update
3. 2 δηλώνει ότι η αιτία δημιουργίας της έκδοσης ήταν deletion

2.8.3 Hibernate Validation

Αποτελεί μία από τις υλοποιήσεις του Jakarta Bean Validation προτύπου το οποίο επιτρέπει τον ορισμό των περιορισμών χρησιμοποιώντας annotations.

Ακολουθεί παράδειγμα από την εργασία όπου χρησιμοποιούνται constraint annotations.

```
@Serializable
data class HotelData(
    @field:NotBlank(message = "{HotelData.name.NotBlank.message}") val name: String,
    @field:NotBlank(message = "{HotelData.description.NotBlank.message}") val description: String,
    @field:Valid val location: Location,
    @field:URL(message = "{HotelData.url.URL.message}", protocol = "https") val url: String,
    @field:NotBlank(message = "{HotelData.hotelType.NotBlank.message}") val hotelType: String,
```

```
  @field:DecimalMin(value = "0.0", message = "{HotelData.score.DecimalMin.message}")
  @field:DecimalMax(value = "10.0", message = "{HotelData.score.DecimalMax.message}")
  @Serializable(with = BigDecimalSerializer::class)
  val score: BigDecimal,
  val rooms: Set<ScrappedRoomInfo> = emptySet()
)
```

- NotBlank δηλώνει ότι η τιμή του πεδίου δεν πρέπει να είναι blank string.
- Valid δηλώνει ότι τα πεδία της class του πεδίου πρέπει να ελεγχθούν αντίστοιχα.
- DecimalMin δηλώνει την μικρότερη τιμή που μπορεί να πάρει το πεδίο
- DecimalMax δηλώνει τη μεγαλύτερη τιμή που μπορεί να πάρει το πεδίο.

Κάθε constraint annotation, εκτός από τα attributes που αντιστοιχούν στο περιορισμό(πχ min τιμή στο DecimalMin),επιτρέπουν τη δήλωση του message attribute που χρησιμοποιείται για επεξήγηση της παραβίασης του περιορισμού.

To Jakarta Bean Validation πρότυπο επιτρέπει τη δημιουργία καινούριων περιορισμών.

Για τη δημιουργία καινούριου περιορισμού, απαιτείται

1. Δημιουργία annotation class με τα απαραίτητα attributes(message,payload,groups)
2. Χρησιμοποιήσει @Constraint annotation πάνω στην καινούρια class. To @Constraint annotation δηλώνει τα ονόματα των classes που υλοποιούν ConstraintValidator interface και πραγματοποιούν την λογική επικύρωσης για τους διαφορετικούς τύπους.

Ακολουθεί παράδειγμα από την εργασία όπου δημιουργήθηκε καινούριο constraint annotation.

```
@Constraint(validatedBy = [EqualPasswordsConstraintValidator::class])
annotation class EqualPasswords(val message: String,val groups: Array<KClass<*>> = [],val payload: Array<KClass<out Payload>> = [])


@Service
class EqualPasswordsConstraintValidator : ConstraintValidator<EqualPasswords,CustomerRegistration>{
  override fun isValid(value: CustomerRegistration?, context: ConstraintValidatorContext?): Boolean {
    if(value == null) return true


    return value.password == value.repeatedPassword
  }
}
```

Ο έλεγχος των περιορισμών γίνεται χρησιμοποιώντας το validation API. Στην εργασία χρησιμοποιείται το Spring validation που έχει integration με το Hibernate Validation.

To Spring Validation κάνει intercept τα calls των μεθόδων στα beans για τα οποία έχει δηλωθεί validation και χρησιμοποιεί το validation API για να κάνει validate τις παραμέτρους. Λειτουργεί χρησιμοποιώντας proxy.

Ακολουθεί παράδειγμα από την εργασία

```
@Validated
interface RoomService {
   fun createRoom(@Valid roomData:RoomData) : Room
   fun updateRoom(@Valid roomData: RoomData) : Room
   fun findHotelRooms(hotel: Hotel) : List<RoomInfo>
}
```

Επειδή το interface έχει το @Validated annotation, το Spring θα δημιουργήσει proxy για την class που υλοποιεί αυτό το interface. Το proxy χρησιμοποιεί το Jakrta Bean Validation API όταν γίνεται κλήση της μεθόδου για να εκτελέσει validation για τις παραμέτρους, και καλεί την πραγματική μέθοδο μόνο όταν το validation είναι επιτυχές.

2.9 Liquibase

Liquibase χρησιμοποιήθηκε για διαχείριση του σχήματος της βάσης. Λειτουργεί σαν git αλλά για το σχήμα της βάσης, δηλαδή κρατάει ιστορικό αλλαγών της βάσης δεδομένων.

Το liquibase χρησιμοποιεί changelog αρχεία τα οποία δηλώνουν τις αλλαγές.

Υπάρχουν διάφορα formats για τη συγγραφή των changelog αρχείων: JSON,XML,YAML,SQL. Στην εργασία χρησιμοποίησα XML.

Το changelog αρχείο αποτελείται από changesets που δηλώνουν τις αλλαγές. Τα changesets εκτελούνται σειριακά ένα μετά το άλλο όπως είναι δηλωμένα μέσα στο αρχείο. Ένα changeset μπορεί να περιέχει περισσότερες από μία αλλαγή.

Ακολουθεί παράδειγμα changeset από την εργασία

```
<changeSet id="1" author="Arslan Kalaitsidis">
    <createTable tableName="REVINFO">
                <column name="REV" type="integer" autoIncrement="true"><constraints nullable="false"
primaryKey="true"/></column>
        <column name="REVTSTMP" type="bigint" />
    </createTable>
</changeSet>
```

Το changeset πρέπει να έχει id και author attributes. Οι τιμές του id attribute,author attribute και του changelog file path χρησιμοποιούνται ως μοναδικό αναγνωριστικό του changeset από Liquibase.

Κάποια changesets που χρησιμοποιήθηκαν στην εργασία

1. createTable: δημιουργεί καινούριο πίνακα.
2. loadData: φορτώνει δεδομένα από csv αρχείο μέσα σε πίνακα.
3. addUniqueConstraint: προσθέτει SQL UNIQUE CONSTRAINT σε στήλη του πίνακα.
4. renameColumn: αλλάζει όνομα μιας στήλη.
5. insert: εκτελεί SQL INSERT.
6. addColumn: προσθέτει καινούρια στήλη στον πίνακα.
7. dropNotNullConstraint: διαγράφει NOT NULL περιορισμό από μια στήλη.
8. dropForeignKeyConstraint: διαγράφει FOREIGN KEY περιορισμό από μια στήλη.

9. sql: εκτελεί οποιαδήποτε SQL εντολή.
10. addForeignKeyConstraint: προσθέτει FOREIGN KEY περιορισμό για μια στήλη.
11. update: εκτελεί SQL UPDATE.
12. createView: δημιουργεί SQL VIEW.

Κατά την πρώτη λειτουργία του liquibase, το liquibase δημιουργεί δύο επιπλέον πίνακες στη βάση: DATABASECHANGELOG και DATABASECHANGELOGLOCK.

Ο πίνακας DATABASECHANGELOG χρησιμοποιείται για την αποθήκευση των changeset που έχουν εφαρμοστεί έτσι ώστε κατά τις επόμενες εκτελέσεις η liquibase μπορεί να αναγνωρίσει ποια changesets έχει ήδη εκτελέσει. Ο πίνακας DATABASECHANGELOGLOCK χρησιμοποιείται από την Liquibase για να εξασφαλίσει ότι μόνο μία liquibase διαδικασία μπορεί να εκτελέσει το changelog.

2.9.1 Spring Boot Integration

Το Spring Boot προσφέρει integration με liquibase. Συγκεκριμένα, κατά την έναρξη της εφαρμογής εκτελεί liquibase update χρησιμοποιώντας το Liquibase Java API.

Εξ' ορισμού, το Spring Boot εκτελεί changelog αρχείο το οποίο πρέπει να βρίσκεται στην τοποθεσία classpath:/db/changelog/db.changelog-master.yml. Αυτή η τοποθεσία μπορεί να αλλάξει χρησιμοποιώντας το spring.liquibase.change-log ιδιότητα όπως και έκανα στην εργασία μου αφού χρησιμοποίησα διαφορετικό format(xml και όχι yaml) και επειδή το root changelog αρχείο ήταν αποθηκευμένο σε διαφορετική τοποθεσία και είχε διαφορετικό όνομα: resources/liquibase/root-changelog.xml.

2.9.2 Context

Για το changeset μπορεί να οριστεί context attribute. Η τιμή του context attribute δηλώνει το context που πρέπει να είναι ενεργοποιημένο προκειμένου το changeset να εκτελεστεί. Context attribute έχει παρόμοιο λειτουργία με τα Spring Profiles μόνο που τα profiles χρησιμοποιούνται από το Spring Framework για τα beans, ενώ το context χρησιμοποιείται από το liquibase για changesets.

Το context μπορεί να οριστεί κατά την εκτέλεση της εντολής χρησιμοποιώντας την --context-filter παράμετρο. Επειδή όμως η εκτέλεση του changelog γίνεται χρησιμοποιώντας το Spring Boot και Liquibase Java API, για τη δήλωση του context χρησιμοποίησα το spring.liquibase.contexts ιδιότητα.

2.10 Bucket4j

Bucket4j είναι rate limiting βιβλιοθήκη γραμμένη σε Java η οποία βασίζεται στο Token Bucket αλγόριθμο.

Ο Token Bucket αλγόριθμος εννοιολογικά λειτουργεί με το εξής τρόπο: σε ένα κάδο συγκεκριμένου μεγέθους προσθέτονται περιοδικά X tokens με τον περιορισμό ότι ο κάδος δεν μπορεί να έχει περισσότερα tokens από ότι επιτρέπει το μέγεθος του. Ο κάδος επιτρέπει την κατανάλωση των token. Όταν ο αριθμός των token που ζητήθηκαν προς κατανάλωση είναι μεγαλύτερος από τον αριθμός των token που υπάρχουν στον κάδο, επιστρέφεται αποτυχία χωρίς να αφαιρούνται τα tokens.

Τα πλεονεκτήματα της Bucket4j είναι

1. Χρησιμοποιεί ακέραιους αριθμούς για υπολογισμούς ώστε να αποφεύγονται τα σφάλματα και ανακρίβειες που βασίζονται στην στρογγυλοποίηση.
2. Προσφέρει lock-free υλοποίηση κάδου για concurrent εφαρμογές χρησιμοποιώντας java.util.concurrent.atomic πακέτο.

3. Προσφέρουν διάφορες υλοποιήσεις για αποθήκευση των κάδων(Redis,JDBC κ.α).
4. Επιτρέπει αντικατάσταση του configuration του κάδου σε runtime με διαφορετικές στρατηγικές.
5. Επιτρέπει δημιουργία κάδου με δύο ή περισσότερους περιορισμούς.
6. Μικρό μέγεθος για την αποθήκευση της κατάστασης του κάδου: κάθε κάδος καταναλώνει το πολύ 40 bytes.

Το Bucket4j χρησιμοποιήθηκε για την προστασία της εφαρμογής από DoS/DDoS επιθέσεις.

2.10.1 Bucket

Bucket είναι rate-limiter που προσφέρει το βασικό interface.

Προσφέρει μεθόδους για

1. Κατανάλωση tokens.
2. Απόκτηση δεδομένων για την κατάσταση του κάδου(π.χ πόσα token έμειναν).
3. Προσθήκη tokens στον κάδο.
4. κ.α.

2.10.2 Bucket Configuration

Περιέχει τους περιορισμούς που πρέπει να χρησιμοποιεί ο κάδος. Το Bucket Configuration είναι immutable που σημαίνει ότι δεν μπορεί να αλλάξει κάποιος το υπάρχων bucket configuration του κάδου αλλά μόνο να το αντικαταστήσει με άλλο configuration.

2.10.3 Περιορισμοί

Κάθε περιορισμός αποτελείται από την χωρητικότητα των token, στρατηγική παραγωγής καινούριων token, τον αρχικό αριθμό των token και το αναγνωριστικό/id.

2.10.4 Στρατηγικές παραγωγής των token

Το Bucket4j υποστηρίζει διαφορετικές στρατηγικές για την παραγωγή των token των περιορισμών.

Οι στρατηγικές αυτές είναι

1. Greedy. Αυτή η στρατηγική προσπαθεί να προσθέσει tokens στον κάδο όσο το δυνατόν συντομότερα. Για παράδειγμα, αν η στρατηγική είναι να προσθέτουμε 100 tokens ανά λεπτό, η greedy στρατηγική δεν θα περιμένει 1 λεπτό για να προσθέσει απευθείας 100 tokens, αλλά κάθε 100 milliseconds θα προσπαθεί να προσθέσει tokens χωρίς να ξεπερνάει την παραγωγή των 100 token ανά 1 λεπτό.
2. Intervally. Αυτή στρατηγική προσθέτει token περιοδικά. Στο προηγούμενο παράδειγμα με 100 token ανά 1 λεπτό, η στρατηγική θα περιμένει 1 λεπτό για να προσθέσει όσα tokens λείπουν στον κάδο ή το πολύ 100 token.
3. Intervally Aligned. Αυτή η στρατηγική λειτουργεί ακριβώς όπως και Intervally με την διαφορά όμως ότι επιτρέπει τον ορισμό της χρονικής στιγμής που πρέπει να γίνει η πρώτη προσθήκη των tokens. Για παράδειγμα, να θέλουμε ένας κάδος να προσθέτει 100 tokens στην αρχή κάθε ώρας ανεξάρτητα από την στιγμή δημιουργίας του κάδου, μπορούμε να χρησιμοποιήσουμε αυτή την στρατηγική για να ορίσουμε η πρώτη προσθήκη να γίνει στην αρχή της επόμενης ώρας και έπειτα να γίνεται κάθε ώρα.

4. Intervally Aligned With Adaptive Initial Tokens. Λειτουργεί σαν το Intervally Aligned με τη διαφορά όμως ότι ο αρχικός αριθμός των token υπολογίζεται χρησιμοποιώντας τον ακόλουθο τύπο

   Math.min(capacity, Math.max(0, bandwidthCapacity - refillTokens) + (timeOfFirstRefillMillis - nowMillis)/refillPeriod * refillTokens)

2.10.5 JDBC Integration

Το Bucket4j προσφέρει JDBC integration το οποίο και χρησιμοποιήθηκε για την αποθήκευση της κατάστασης των κάδων.

Το Bucket4j JDBC απαιτεί πίνακα με δύο πεδία για αποθήκευση των κάδων

1. id στήλη που είναι το primary key του πίνακα. Δεν υπάρχει περιορισμός τύπου.
2. state με blob/binary τύπο για την αποθήκευση της κατάστασης του πίνακα.

2.11 Testing

Testing αποτελεί την συγγραφή κώδικα ή scripts που έχουν ως στόχο την επαλήθευση σωστής λειτουργίας του προγράμματος συγκρίνοντας τα αποτελέσματα των τεστ με τις απαιτήσεις και προδιαγραφές.

Τα τεστ μπορούν να επαληθεύουν διαφορετικές όψεις της σωστής λειτουργίας του προγράμματος ή και διαφορετική κλίμακα του προγράμματος, και για αυτό το λόγο υπάρχουν διαφορετικοί τύποι των τεστ με το καθένα να επιτυγχάνει διαφορετικούς στόχους.

Για παράδειγμα, performance τεστ έχει ως στόχο την επαλήθευση σωστής λειτουργίας του προγράμματος από την άποψη των απαιτήσεων απόδοσης (διεκπεραιωτικότητα, καθυστέρηση κτλ) για κάποιο κομμάτι του προγράμματος, ενώ το unit τεστ έχει ως στόχο την επαλήθευση σωστής συμπεριφοράς μιας μονάδας του προγράμματος από την άποψη των απαιτήσεων που έχουν οριστεί για την μονάδα υπό έλεγχο.

Ένα άλλο παράδειγμα αποτελούν τα unit, integration και end-to-end τεστ τα οποία έχουν ίδιο στόχο: επαλήθευση σωστής συμπεριφοράς του προγράμματος βάση των απαιτήσεων, αλλά έχουν διαφορετική κλίμακα: το unit τεστ έχει στόχο την επαλήθευση σωστής λειτουργίας μιας μονάδας εξετάζοντας τις απαιτήσεις που έχουν οριστεί, το integration τεστ έχει στόχο την επαλήθευση συμπεριφοράς συστατικού του συστήματος σε συνεργασία με άλλα συστατικά τα οποία απαιτούνται για τη λειτουργία του, τέλος το end-to-end τεστ(ή system τεστ) έχει στόχο την επαλήθευση συμπεριφοράς όλου του/της συστήματος/εφαρμογής.

Τα πλεονεκτήματα των τεστ είναι

1. Αυτοματισμός. Επειδή τα τεστ είναι σε μορφή κώδικα/script δεν απαιτείται αλληλεπίδραση με τον άνθρωπο για να γίνει επαλήθευση.
2. Ταχύτητα. Τα τεστ εκτελούνται σχετικά γρήγορα σε σύγκριση με την χειροκίνητη δοκιμή του προγράμματος από τον άνθρωπο και αποτρέπουν το ανθρώπινο λάθος που μπορεί να γίνει.
3. Επιτρέπουν να βρίσκουμε τα σφάλματα νωρίτερα. Αντί να περιμένουμε το σφάλμα να εμφανιστεί στην παραγωγή, τα τεστ βρίσκουν τα τυχόν σφάλματα μετά την εκτέλεσή τους και μας δίνουν ένα επίπεδο βεβαιότητας ότι το πρόγραμμά μας λειτουργεί σύμφωνα με τις προδιαγραφές και απαιτήσεις.
4. Αναπαραγωγιμότητα . Όταν βρούμε κάποιο σφάλμα στο πρόγραμμα, μπορούμε να γράψουμε τεστ σενάριο το οποίο προκαλεί το σφάλμα και το οποίο μπορούμε να χρησιμοποιήσουμε για αναπαραγωγή του σφάλματος. Αφού γίνε επιδιόρθωση του σφάλματος, μπορούμε να χρησιμοποιήσουμε το τεστ σενάριο που έχουμε φτιάξει για να ελέγχουμε ότι η εφαρμογή μας δεν προκαλεί το σφάλμα μετά τις μελλοντικές αλλαγές του κώδικα.

5. Documentation. Τα τεστ αποτελούν documentation του προγράμματος. Διαβάζοντας τα τεστ σενάρια, μπορούμε να μάθουμε τις απαιτήσεις και συμπεριφορά του προγράμματος. Επίσης, επειδή τα τεστ συνήθως εκτελούνται αυτόματα και αρκετά συχνά, αναπαριστούν με μεγαλύτερη ακρίβεια τις απαιτήσεις και προδιαγραφές του συστήματός απ' ότι άλλα έγγραφα.

Στην εργασία μου, έχω δημιουργήσει unit και integration τεστ.

2.11.1 Spring TestContext Framework

Το Spring TestContext Framework προσφέρει υποστήριξη για τη συγγραφή unit και integration τεστ για τις εφαρμογές που χρησιμοποιούν το Spring Framework.

Τα βασικά συστατικά του Spring TestContext Framework είναι

- TestContextManager διαχειρίζεται για το test suite το TestContext και ειδοποιεί τους TestExecutionListener για τα διάφορα events που συμβαίνουν κατά την εκτέλεση των τεστ.
- TestContext είναι το αντικείμενο το οποίο περιέχει το Spring context με τα beans της εφαρμογής μας. Επίσης, προσφέρει caching ώστε το Spring context να φορτώνεται μόνο μία φορά.
- SmartContextLoader χρησιμοποιείται από το TestContext για τη δημιουργία του Spring context.
- TestExecutionListener υλοποιούν την βασική υποστήριξη για τα τεστ προσφέροντας διαχείριση συναλλαγών στα τεστ, dependency injection για τις τεστ classes και άλλα.

Το Spring TestContext framework προσφέρει και αυτόματα ενεργοποιεί ορισμένους TestExecutionListeners.

Οι σημαντικότεροι TestExecutionListeners είναι

1. DependencyInjectionTestExecutionListener προσφέρει dependency injection για την τεστ class που εκτελείται. Συγκεκριμένα, υποστηρίζει dependency injection μέσω του constructor ή/και χρησιμοποιώντας @Autowired annotation.

   Ακολουθεί παραδείγματα από την εργασία όπου χρησιμοποιείται dependency injection

```
@TestConstructor(autowireMode = TestConstructor.AutowireMode.ALL)
@Transactional
@Testcontainers
@AutoConfigureTestEntityManager
@SpringBootTest
abstract class AbstractITest : AbstractTest(){
    @Autowired
    protected lateinit var messageSource: MessageSource

                                                          ….
}
```

   Στο παραπάνω παράδειγμα το @Autowired δηλώνει ότι το MessageSource bean πρέπει να βρεθεί στο Spring context και να ανατεθεί η αναφορά σε αυτό στην μεταβλητή messageSource της test class.

```
class HotelServiceITest @Autowired constructor(private val hotelService: HotelService) :
AbstractITest() {}
```

   Το παραπάνω παράδειγμα δείχνει το dependency injection μέσα από constructor

2. TransactionalTestExecutionListener προσφέρει εκτέλεση του κώδικα των τεστ μέσα σε συναλλαγές με εξ' ορισμούς rollback στρατηγική: εκτελεί αυτόματο rollback μετά το τέλος εκτέλεσης κάθε τεστ

μεθόδου. Για τη λειτουργία του TransactionalTestExecutionListener, το τεστ πρέπει να χρησιμοποιήσει @Transactional annotation.

2.11.2 Spring Boot Test

Προσφέρει υποστήριξη για τη συγγραφή τεστ για τις εφαρμογές που χρησιμοποιούν το Spring Boot. Βασίζεται πάνω στο Spring TestContext Framework.

Προσφέρει SpringBootTest annotation το οποίο

1. Αλλάζει τον εξ' ορισμού SmartContextLoader σε SpringBootContextLoader.
2. Αυτόματα αναζητά @SpringBootConfiguration και το χρησιμοποιεί για τη δημιουργία του Spring Context.
3. Και άλλα.

2.11.3 TestContainers

Java testcontainers βιβλιοθήκη χρησιμοποιεί Java API client[5] για την επικοινωνία με το Docker process[6]. Χρησιμοποιεί το docker client για να στέλνει requests στο docker process κατά την εκτέλεση των τεστ. Συγκεκριμένα, ξεκινάει containers που είναι ορισμένη μέσα στον κώδικα των τεστ πριν ξεκινήσει η εκτέλεσή τους επιτρέποντας την προετοιμασία του περιβάλλοντος που απαιτείται για την εκτέλεση των τεστ.

Η χρήση της βιβλιοθήκης testcontainers σε σύγκριση με χειροκίνητη δημιουργία του τεστ περιβάλλοντος έχει τα εξής πλεονεκτήματα:

1. Το περιβάλλον μέσα στο οποίο εκτελούνται τα τεστ είναι πιο κοντά στο περιβάλλον παραγωγής μέσα στο οποίο θα εκτελείται η εφαρμογή. Για παράδειγμα, αντί να χρησιμοποιείται κάποια in-memory βάση δεδομένων, testcontainers μας επιτρέπει να τρέξουμε την έκδοση της βάσης δεδομένων που θα τρέχει στο περιβάλλον παραγωγής.
2. Η testcontainers βιβλιοθήκη κάνει πιο εύκολη την προετοιμασία του τεστ περιβάλλοντος. Αντί να αποτελεί προαπαιτούμενο η ύπαρξη υποδομής που απαιτείται για να τρέξουν τα τεστ(για παράδειγμα, να τρέχει στο μηχάνημα στο οποίο εκτελούνται τα tests κάποια συγκεκριμένη έκδοση MySQL), το μόνο απαιτούμενο[7] για την εκτέλεση των τεστ είναι κάποιο μηχάνημα στο οποίο τρέχει docker.
3. Το configuration του τεστ περιβάλλοντος δηλώνεται μέσα στον κώδικα των τεστ.

Ακολουθεί παράδειγμα από την εργασία όπου χρησιμοποιήθηκε η βιβλιοθήκη testcontainers.

Ο κώδικας χρησιμοποιεί το mysql module της βιβλιοθήκης για να σηκώσει container που τρέχει την τελευταία έκδοση της mysql. Το @DynamicProperties[8] χρησιμοποιείται για αλλαγή των ιδιοτήτων σύνδεσης της εφαρμογής στην βάση καθώς testcontaiers χρησιμοποιεί κάθε φορά random port για τους containers που σηκώνει.

```
companion object{
```

5 GitHub Project

6 By default, προσπαθεί να βρει τοπική εγκατάσταση του Docker, αλλά μπορεί να γίνει παραμετροποίηση για να επικοινωνήσει με remote Docker host.

7 To docker client που χρησιμοποιεί testcontainers μπορεί να παραμετροποιηθεί να χρησιμοποιεί remote docker installation, οπότε σε αυτή την περίπτωση,το μηχάνημα στο οποίο εκτελούνται τα tests δεν απαιτείται να έχει docker.

8 DynamicProperties είναι μέρος της Spring βιβλιοθήκης,και αρχικά έχει δημιουργηθεί για πιο εύκολο integration με την βιβλιοθήκη testcontainers.

```
    private val mysqlContainer = MySQLContainer("mysql:latest").withReuse(true)
    @JvmStatic
    @DynamicPropertySource
    fun propertyInit(registry: DynamicPropertyRegistry){
      mysqlContainer.start()
      registry.add("spring.datasource.username"){ mysqlContainer.username }
      registry.add("spring.datasource.password"){ mysqlContainer.password }
      registry.add("spring.datasource.url"){ mysqlContainer.jdbcUrl }
    }
  }
```

2.11.4 Unit Testing

Στόχος του unit τεστ είναι η επαλήθευση συμπεριφοράς της μικρότερης μονάδας, εξ ου και το όνομα unit(μονάδα) testing. Στην Java και Kotlin η μικρότερη μονάδα την συμπεριφορά της οποίας μπορούμε να επαληθεύσουμε είναι η class[9].

Η συμπεριφορά μιας class ορίζεται από τις μεθόδους της. Συγκεκριμένα, κάθε μέθοδος ορίζει

1. Προϋποθέσεις(Preconditions) που πρέπει να τηρούνται για την κλήση της μεθόδου. Επίσης, ορίζεται η συμπεριφορά της μεθόδου όταν γίνεται κλήση της μεθόδου με μία ή περισσότερες προϋποθέσεις να παραβιάζονται.
   Προϋπόθεση μπορεί να αποτελεί κάποια κατάσταση στην οποία πρέπει να βρίσκεται το αντικείμενο ώστε η μέθοδος να ολοκληρωθεί επιτυχώς. Για παράδειγμα, η μέθοδος remove(index) της List έχει ως προϋπόθεση το index >= 0 και index < size(), όπου το size() εξαρτάται από την κατάσταση της λίστας.
2. Postconditions είναι συνθήκες που ισχύουν μετά την ολοκλήρωση της μεθόδου ή/και τα side effects της μεθόδου. Χρησιμοποιώντας το προηγούμενο παράδειγμα με το remove, αν η κλήση της μεθόδου επιστρέφει επιτυχώς, τότε ένα από τα postconditions που μπορούμε να ελέγξουμε είναι το size() = προηγούμενο size() - 1.

Ένα τυπικό unit test εκτελεί τα παρακάτω βήματα

1. Καλεί κάποια μέθοδο πάνω στο αντικείμενο εξετάζοντας συμπεριφορά για συγκεκριμένες προϋποθέσεις.
2. Ελέγχει τα postconditions και το αποτέλεσμα της μεθόδου.

Σπάνια όμως η λειτουργία μιας class είναι απομονωμένη από άλλες classes. Συνήθως η class απαιτεί αντικείμενα άλλων classes για τη λειτουργία της. Επειδή ο στόχος του unit τεστ είναι επαλήθευση συμπεριφοράς της απομονωμένης μονάδας, απαιτείται αντικατάσταση αυτών των αντικειμένων από τα οποία εξαρτάται η class με αντικείμενα τα οποία θα είναι υπό έλεγχο του τεστ ώστε να αποφύγουμε τις περιπτώσεις όπου

- η συμπεριφορά του αντικειμένου από το οποίο εξαρτάται η μονάδα υπό έλεγχο δεν είναι σταθερή(για παράδειγμα, εξαρτάται από την ώρα του συστήματος) με αποτέλεσμα τα τεστ μας να

9 Παρά το γεγονός ότι η Kotlin επιτρέπει συναρτήσεις που δεν εμπεριέχονται μέσα σε class, σε byte code επίπεδο η Kotlin δημιουργεί static class γιατί JVM δεν επιτρέπει “classless” συναρτήσεις.

μην εκτελούνται με ντετερμινιστικό τρόπο αλλά να εξαρτώνται από παράγοντες που είναι εκτός ελέγχου μας.

- είναι δύσκολο να φέρουμε το αντικείμενο σε κατάσταση που απαιτείται για το τεστ σενάριο που γράφουμε. Για παράδειγμα, μπορεί να ελέγχουμε σενάριο στο οποίο ένα από τα dependencies της μονάδας υπό έλεγχο πετάει εξαίρεση, ενώ να φέρουμε το dependency σε κατάσταση που να πετάει την εξαίρεση που θέλουμε είναι δύσκολο ή και αδύνατο(π.χ. η εξαίρεση να εξαρτάται από παράγοντα που είναι εκτός ελέγχου του τεστ).

Τα αντικείμενα με τα οποία αντικαθιστούμε τα πραγματικά αντικείμενα που απαιτούνται από την μονάδα υπό έλεγχο ονομάζονται test doubles.

Test double είναι αντικείμενο που έχει ίδιο interface με κάποια class από την οποία εξαρτάται η μονάδα υπό έλεγχο αλλά για το οποίο μπορούμε

- Να ορίσουμε τη συμπεριφορά. Για παράδειγμα, μπορούμε να ορίσουμε το test double να επιστρέψει συγκεκριμένο αποτέλεσμα όταν η μονάδα υπό έλεγχο καλεί κάποια μέθοδο πάνω στο mock αντικείμενο.
- Να επαληθεύσουμε το ιστορικό αλληλεπίδρασης που είχε το test double με τη μονάδα υπό έλεγχο. Αυτό συνήθως σημαίνει ότι μπορούμε να επαληθεύσουμε ότι η μονάδα υπό έλεγχο έχει καλέσει συγκεκριμένη μέθοδο με συγκεκριμένες παραμέτρους πάνω στο test double.
- Να ζητήσουμε και έπειτα να επαληθεύσουμε τις παραμέτρους που έχει περάσει η μονάδα υπό έλεγχο όταν κάλεσε κάποια μέθοδο πάνω στο test double αντικείμενο.

Υπάρχουν 4 τύποι των test doubles

1. Dummy είναι test double για το οποίο δεν ορίζεται συμπεριφορά και για το οποίο δεν ελέγχεται αλληλεπίδραση που είχε με αυτό η μονάδα υπό έλεγχο. Συνήθως είναι αντικείμενα τα οποία απαιτούνται για τη δημιουργία του αντικειμένου της class της μονάδας υπό έλεγχο ή της κλήσης της μεθόδου αλλά με τα οποία η μονάδα υπό έλεγχο δεν αλληλεπιδρά για το σενάριο το οποίο ελέγχουμε.
2. Stub είναι test double για το οποίο μπορούμε να ορίσουμε την συμπεριφορά: ποια δεδομένα επιστρέφει ή ποια εξαίρεση πετάει όταν η μονάδα υπό έλεγχο αλληλεπιδρά με αυτό.
3. Spy είναι test double το οποίο περιέχει το πραγματικό αντικείμενο από το οποίο εξαρτάται η μονάδα υπό έλεγχο και ανακατευθύνει αλληλεπίδραση στο πραγματικό αντικείμενο. Το πλεονέκτημα των spy είναι ότι προσφέρουν δυνατότητα επαλήθευσης αλληλεπίδρασης που είχε η μονάδα υπό έλεγχο με το test double ενώ ταυτοχρόνως επιτρέπουν τη χρήση πραγματικής υλοποίησης για τη συμπεριφορά τους.
4. Mock είναι test double το οποίο επιτρέπει stubbing και επαλήθευση συμπεριφοράς.

Συνήθως οι βιβλιοθήκες προσφέρουν μόνο mocks και spies, και αντί για dummy και stub μπορούμε να χρησιμοποιήσουμε το mock.

2.11.4.1Mockk και Springmockk

Για τη δημιουργία των mocks στα unit tests χρησιμοποίησα mockk και springmockk.

Mockk είναι Kotlin mocking βιβλιοθήκη. Επιτρέπει τη δημιουργία των mocks και spies και την επαλήθευση της αλληλεπίδρασης.

Ακολουθεί παράδειγμα χρήσης mockk βιβλιοθήκης από την εργασία

```
@Test
fun `should make a call to scrapper when data not available in database and save it in hotel service before re-requesting again`() {
```

```
val scrapperResult = HotelData("ANY_NAME","ANY_DESCRITPION", Location("ANY_ADDRESS","ANY_CITY",null,null),"http://test.com","ANY_HOTEL_TYPE", BigDecimal.ONE)
      every { hotelServiceMock.getHotelInfo(request) } returns Optional.empty() andThen Optional.of(hotelInfo)
      every { scrappingServiceMock.requestHotelInfo(request.toScrappingRequest()) } returns scrapperResult
    every { scrapperDataPersistServiceMock.saveScrappingResult(scrapperResult) } just runs
    .....
    verifyAll {
       hotelServiceMock.getHotelInfo(request)
       scrappingServiceMock.requestHotelInfo(request.toScrappingRequest())
       scrapperDataPersistServiceMock.saveScrappingResult(scrapperResult)
       hotelServiceMock.getHotelInfo(request)
    }
 }
```

- Η μέθοδος every χρησιμοποιείται για stubbing: δήλωση συμπεριφοράς του mock.
  - returns χρησιμοποιείται για δήλωση του αποτελέσματος που πρέπει να επιστρέψει το mock όταν κληθεί η μέθοδος δηλωμένη στο every κομμάτι.
  - just runs χρησιμοποιείται για μεθόδους του mock που δεν επιστρέφουν αποτέλεσμα(έχουν void/Unit τύπο) και δηλώνει ότι η μέθοδος πρέπει να επιστρέψει επιτυχώς.
  - andThen χρησιμοποιείται για επιστροφή περισσότερα από ένα αποτέλεσμα όταν ξέρουμε ότι η μέθοδος θα κληθεί περισσότερα από μία φορά.
  - Επιπρόσθετα, μπορούμε να χρησιμοποιήσουμε throws μέθοδο για να δηλώσουμε ότι το mock πρέπει να πετάξει εξαίρεση.
- verifyAll,verify,verifySequence,verifyOrder χρησιμοποιούνται για επαλήθευση ιστορικού αλληλεπίδρασης του mock αντικειμένου.
  - verifyAll ελέγχει ότι το ιστορικό αλληλεπίδρασης των mock περιέχει όλα τα calls τα οποία είναι δηλωμένα μέσα σε {} και ότι δεν υπήρχε καμία άλλη αλληλεπίδραση εκτός από αυτές που δηλώθηκαν.
  - verifySequence λειτουργεί όπως και το verifyAll με τη διαφορά όμως ότι ελέγχει και τη σειρά με την οποία έγιναν οι κλήσεις μεθόδων.
  - verifyOrder λειτουργεί όπως και το verifySequence με τη διαφορά όμως ότι επιτρέπει τα mocks να έχουν περισσότερες κλήσεις από αυτές που έχουν δηλωθεί μέσα στο {}.
  - verify είναι η πιο γενική μορφή η οποία επιτρέπει τη δήλωση διαφορετικών παραμέτρων.

Το Spring Boot Test προσφέρει τη δυνατότητα δημιουργίας mock τα οποία χρησιμοποιούνται για την αντικατάσταση των beans στον test context χρησιμοποιώντας @MockBean annotation πάνω στα πεδία της test class. Spring Boot Test αναγνωρίζει αυτό το annotation και αντικαθιστά τα αντίστοιχα beans στο Spring context με τα mocks. Για τη δημιουργία των mock, το Spring Boot Test χρησιμοποιεί την Mockito βιβλιοθήκη.

Το springmockk προσφέρει αντίστοιχο @MockkBean(διαφορετικό από το @MockBean) annotation με ίδια λειτουργία όπως και το Spring Boot Test @MockBean, αλλά για τη δημιουργία των mocks χρησιμοποιούν την mockk βιβλιοθήκη της Kotlin.

Η λειτουργία της springmockk βιβλιοθήκης(όπως και της αντίστοιχης λειτουργίας που προσφέρεται εξ' ορισμού από το Spring Boot Test) βασίζεται στην λειτουργία των TestExecutionListener: το springmockk προσφέρει MockkTestExecutionListener το οποίο αναγνωρίζει @MockkBean annotation και δημιουργεί mock αντικείμενα με την mockk βιβλιοθήκη. Επίσης, προσφέρει ClearMocksTestExecutionListener η κύρια λειτουργία του οποίου είναι να καθαρίζει τα mocks μετά την εκτέλεση του κάθε τεστ ώστε κάθε τεστ να ξεκινάει με καθαρά mocks – δεν περιέχουν stubbing και ιστορικό αλληλεπίδρασης που έχουν οριστεί στα προηγούμενα τεστ.

2.11.4.2JUnit

JUnit είναι framework για την συγγραφή και εκτέλεση των test. Στο JUnit, τα τεστ ορίζονται ως μέθοδοι των Java/Kotlin classes.

Παρά το όνομα JUnit, το JUnit δεν χρησιμοποιείται μοναδικά για την εκτέλεση και συγγραφή των unit τεστ. Το JUnit μπορεί να χρησιμοποιηθεί για unit,integration και end-to-end τεστ, αλλά και για άλλα τεστ αρκεί τα τεστ να μπορούν να περιγραφούν ως μέθοδοι.

Οι μέθοδοι που αποτελούν τα τεστ σενάρια πρέπει να έχουν το @Test annotation και να έχουν void τύπο επιστροφής(στην Kotlin δεν υπάρχει void τύπος και χρησιμοποιείται ο Unit τύπος).

Εκτός από @Test annotation, JUnit υποστηρίζει @ParameterizedTest. Το @ParameterizedTest είναι τεστ το οποίο εκτελείται πολλαπλές φορές με διαφορετικές παραμέτρους κάθε φορά. Η πηγή παραμέτρων για το τεστ μπορεί να δηλωθεί χρησιμοποιώντας

- @MethodSource annotation δηλώνει την static μέθοδο που επιστρέφει τις παραμέτρους που χρησιμοποιούνται για την εκτέλεση του @ParameterizedTest.
- @CsvSource/@CsvFileSource δηλώνει csv τιμές ή csv αρχείο.
- @EnumSource δηλώνει enum class. Το τεστ εκτελείται με κάθε enum αντικείμενο να χρησιμοποιείται ως παράμετρος.
- @ValueSource δηλώνει τις παραμέτρους χρησιμοποιώντας τα attributes του annotation.
- Και άλλα….

Στην εργασία, χρησιμοποίησα κυρίως @MethodSource ως πηγή παραμέτρων.

Εκτός από @Test και @ParameterizedTest, JUnit προσφέρει τα παρακάτω annotations

- @BeforeEach δηλώνει μέθοδο της test class η οποία εκτελείται πριν την εκτέλεση κάθε τεστ μεθόδου. Μπορεί να χρησιμοποιηθεί για επαναφορά των mocks ώστε ένα τεστ να μην επηρεάζει τα άλλα τεστ που εκτελούνται μετά από αυτό.
- @AfterEach δηλώνει μέθοδο της class η οποία εκτελείται μετά την εκτέλεση της κάθε τεστ μεθόδου.
- @BeforeAll δηλώνει στατική μέθοδο που εκτελείται μία φορά πριν την εκτέλεση των τεστ μεθόδων. Μπορεί να χρησιμοποιηθεί για προετοιμασία του περιβάλλοντος εκτέλεσης των τεστ.
- @AfterAll δηλώνει στατική μέθοδο που εκτελείται μία φορά μετά την εκτέλεση των τεστ μεθόδων.

Τα JUnit τεστ μπορούν να εκτελεστούν με έναν από τους παρακάτω τρόπους

1. Χρησιμοποιώντας το gradle test ή check task.
2. Χρησιμοποιώντας το integration που προσφέρουν ορισμένα IDE.
3. Μέσα από το terminal.

Στην εργασία χρησιμοποίησα τους τρόπους (1) και (2) για την εκτέλεση των τεστ.

2.12 Άλλα

- Kotlinx.serialization χρησιμοποιήθηκε για serialization/deserialization των αντικειμένων σε/από JSON format καθώς προσφέρει πιο καλή υποστήριξή της Kotlin γλώσσας σε σύγκριση με άλλες παρόμοιες Java βιβλιοθήκες.
- Kotest Assertions είναι assertion βιβλιοθήκη για testing.
- Logback, slf4j και Spring logging χρησιμοποιήθηκαν για logging.

3 Υλοποίηση

3.1 Αρχιτεκτονική

Η εφαρμογή ακολουθεί την αρχιτεκτονική τριών επιπέδων:

1. Το επίπεδο παρουσίασης περιλαμβάνει τα συστατικά της εφαρμογής με τα οποία αλληλεπιδρά ο χρήστης/client. Σε αυτό το επίπεδο περιλαμβάνονται http controllers.
2. Το επίπεδο της εφαρμογής περιλαμβάνει τα συστατικά τα οποία ορίζουν τους κανόνες και περιορισμούς που επιβάλλονται από τις απαιτήσεις που πρέπει να ικανοποιεί η εφαρμογή. Σε αυτό το επίπεδο ανήκουν οι κεντρικές classes της εφαρμογής που υλοποιούν τις βασικές απαιτήσεις της εφαρμογής.
3. Το επίπεδο των δεδομένων περιέχει τα συστατικά που χρησιμοποιούνται για την αποθήκευση και ανάκτηση των δεδομένων της εφαρμογής. Σε αυτό το επίπεδο ανήκουν η βάση δεδομένων και οι classes της εφαρμογής που επικοινωνούν με αυτήν.

Ο κώδικας που ανήκει σε κάθε επίπεδο είναι οργανωμένος σε πακέτα(packages)

- controller και dto πακέτα περιέχουν τις classes του επιπέδου παρουσίασης.
- entity και repository πακέτα περιέχουν class του επιπέδου δεδομένων.
- service πακέτο περιέχει τις classes του επιπέδου εφαρμογής.

3.2 Roles

Κάθε χρήστης στο σύστημα έχει έναν από τους τρεις ρόλους: USER, BASIC, PREMIUM.

Η αρχιτεκτονική με ρόλους χρησιμοποιήθηκε για εξουσιοδότηση των χρηστών και για προσφορά διαφορετικών περιορισμών/δυνατοτήτων στους χρήστες αναλόγως με τον ρόλο του χρήστη.

3.3 Email

Για την αποστολή των mail στην εφαρμογή χρησιμοποιήθηκε πάροχος email υπηρεσιών SendGrid.

Το SendGrid υποστηρίζει διαφορετικές συνδρομές και στην εφαρμογή χρησιμοποίησα την δωρεάν συνδρομή η οποία επιτρέπει 100 mail/μέρα.

Για την αποστολή των mails το SendGrid προσφέρει δύο τρόπους

1. Χρησιμοποιώντας Web API.
2. Χρησιμοποιώντας SMTP πρωτόκολλο και τον SMTP server του SendGrid.

Στην εφαρμογή χρησιμοποίησα το Web API καθώς το SendGrid προσφέρει για ορισμένες γλώσσες(μία από αυτές είναι Java) βιβλιοθήκες που διευκολύνουν την χρήση του Web API.

3.4 JWT

Η εφαρμογή χρησιμοποιεί εκτενώς τα JWT για διάφορες λειτουργίες. Υπάρχουν δύο τύποι JWT: JWS και JWE.

Το JWS είναι JWT το οποίο,εκτός από payload/δεδομένα, έχει υπογραφή η οποία προσφέρει integrity στο JWT, δηλαδή δεν μπορεί να αλλάξει κάποιος τα δεδομένα αποθηκευμένα μέσα στο JWT.

Το payload του JWS δεν είναι κρυπτογραφημένο οπότε δεν μπορεί να χρησιμοποιηθεί για μεταφορά ευαίσθητων δεδομένων.

JWE περιέχει κρυπτογραφημένο payload οπότε προσφέρει confidentiality.

Στην εργασία χρησιμοποίησα JWS.

Το JWS αποτελεί base64 encoded string με τρία μέρη τα οποία είναι χωρισμένα με . χαρακτήρα.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJpc3MiOiJQcmljZU1vbml0b3JpbmciLCJzdWIiOiIxIiwiZXhwIjoxNzA0OTkxMTM1LCJpYXQiOjE3MDQ5MDQ3MzUsInNjb3BlIjoiVVNFUiJ9.v_nKIcGzXKeaq-q9Dosgnwld2YBIdLlAZraWBaFs3flfSP0FSjoP8AZEzvVIZKqudLX4dJqtvvUG743AUSDWxw

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "typ": "JWT",
  "alg": "HS512"
}
```


PAYLOAD: DATA


```
{
  "iss": "PriceMonitoring",
  "sub": "1",
  "exp": 1704991135,
  "iat": 1704904735,
  "scope": "USER"
}
```


VERIFY SIGNATURE

```
HMACSHA512(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Figure 16: Παράδειγμα JWS που επιστρέφει η εφαρμογή

Το πρώτο μέρος είναι η κεφαλίδα του JWS και περιέχει λεπτομέρειες σχετικά με τον αλγόριθμο που χρησιμοποιήθηκε για την δημιουργία της ψηφιακή υπογραφή.

Το δεύτερο μέρος είναι το payload του JWS το οποίο αποτελεί json αντικείμενο. Η εφαρμογή χρησιμοποιεί τα παρακάτω json πεδία στο payload του JWS

1. iss ορίζει την οντότητα που έκδωσε το access token.
2. sub: το subject/ο χρήστης του JWS
3. exp: χρονική σήμανση μετά την οποία το token δεν θεωρείται πια έγκυρο.
4. iat: χρονική σήμανση την οποία δημιουργήθηκε το access token.
5. scope ορίζει το scope του JWS. Μπορεί να αποτελεί ρόλο,κομμάτι της εφαρμογής,κάποιο δικαίωμα κτλ.
6. name: περιέχει το όνομα του χρήστη.
7. email: περιέχει το email του χρήστη.
8. maxMonitorList: δηλώνει τον μέγιστο αριθμό των λιστών που μπορεί να δημιουργήσει ο χρήστης.
9. maxRooms: δηλώνει τον μέγιστο αριθμό των δωματίων που μπορεί να περιέχει μια λίστα.

Το τρίτο μέρος είναι η ψηφιακή υπογραφή η οποία δημιουργείται χρησιμοποιώντας ως input τα πρώτα δύο μέρη του JWS. Για τον υπολογισμό της υπογραφής στην εφαρμογή χρησιμοποιήθηκε HMAC-SHA-512.

Τα JWS χρησιμοποιήθηκαν για αυθεντικοποίηση/εξουσιοδότηση, email verification, password reset.

3.4.1 JWS Αυθεντικοποίηση και Εξουσιοδότηση

Μετά την αυθεντικοποίηση του χρήστη με email και κωδικό πρόσβασης, η εφαρμογή επιστρέφει ένα μακροχρόνιο JWS που ισχύει για 24 ώρες.

Το πλεονέκτημα χρήστης JWS για αυθεντικοποίηση/εξουσιοδότηση είναι ότι επιτρέπει την υλοποίηση stateless εφαρμογών – εφαρμογών που δεν διαχειρίζονται session για να θυμούνται τον χρήστη.

3.4.2 JWS Email Verification

Για email verification χρησιμοποιήθηκαν επίσης τα JWS. Συγκεκριμένα, για να κάνει ο χρήστης επαλήθευση του email, πρέπει να ζητήσει την εφαρμογή να στείλει στο email ένα JWS. Αυτό το JWS έχει μικρότερη διάρκεια(10 λεπτά) και περιέχει ως payload το email.

Αυτό το JWS στη συνέχεια μπορεί να χρησιμοποιηθεί για τη δημιουργία χρήστη στο σύστημα με το email που υπάρχει στο payload.

Το πλεονέκτημα χρήστης του JWS για επαλήθευση email είναι ότι αποφεύγουμε

1. Δημιουργία inactive χρήστη στη βάση πριν γίνει επαλήθευση του email.
2. Αποθήκευση του OTP κωδικού καθώς το JWS περιέχει στο payload το email.

Το μειονέκτημα της χρήστης του JWS για email verification σε σύγκριση με ένα OTP είναι ότι είναι αρκετά μεγάλο σε σύγκριση με OTP κωδικούς που είναι συνήθως 6-7 ψηφία. Αυτό το πρόβλημα μπορεί να λυθεί χρησιμοποιώντας κάποιο κουμπί στο email αντί για απλό κείμενο.

3.4.3 JWS Password Reset

JWS για password reset λειτουργεί παρόμοια με το JWS email verification. Συγκεκριμένα, για να αλλαγή ή επαναφορά κωδικού χρήστη, ο χρήστης πρέπει να ζητήσει την εφαρμογή να στείλει στο email ένα JWS. Αυτό το JWS έχει επίσης μικρή διάρκεια(10 λεπτά) και περιέχει ως payload το email.

Αυτό το JWS στη συνέχεια μπορεί να χρησιμοποιηθεί για αλλαγή κωδικού χρήστη ο οποίο προσδιορίζεται από το email το οποίο είναι αποθηκευμένο στο payload.

Το JWS για επαναφορά/αλλαγή κωδικού χρήστη έχει τα ίδια πλεονεκτήματα και μειονεκτήματα με τη χρήστη του JWS για email verification.

Η χρήση του JWS για επαναφορά/αλλαγή κωδικού έχει ένα επιπλέον μειονέκτημα ότι το JWS μπορεί να επαναχρησιμοποιηθεί πολλαπλές φορές στη διάρκεια των 10 λεπτών για αλλαγή κωδικού. Αυτό το μειονέκτημα δεν υπάρχει στο email verification επειδή η εφαρμογή δεν επιτρέπει εγγραφή του χρήστη αν υπάρχει ήδη χρήστης με το ίδιο email, δηλαδή η χρήση του email JWS δεύτερη φορά θα οδηγήσει σε εξαίρεση.

Αυτό το πρόβλημα μπορεί να λυθεί μόνο με την αποθήκευση του JWS.

3.5 Rate Limiting

Στο επίπεδο της εφαρμογής υλοποιήθηκε το rate limiting χρησιμοποιώντας την βιβλιοθήκη Bucket4j.

Το rate limit υλοποιήθηκε για τους εξής λόγους

1. Για προστασία από γενικές DoS/DDoS επιθέσεις.
2. Για προστασία από DoS/DDoS επιθέσεις στα endpoints που χρησιμοποιούν το email service. Χωρίς rate limiting, κάποιος χρήστης μπορεί εύκολα να εξαντλήσει τον αριθμό των mail που μπορούμε να στείλουμε μέσω του email service provider.

Το rate limit configuration γίνεται χρησιμοποιώντας το @ConfigurationProperties.

```
@ConfigurationProperties("rate.limit")
```

```
class ConfigurationPropertiesRateLimit(val email: RateLimit,val password: RateLimit,val api: RateLimit)


data class RateLimit(val capacity: Long,val refillStrategy: RateLimitRefillStrategy,val refillTokens: Long,val refillPeriod: Duration)
```

Στην εφαρμογή υλοποιήθηκαν οι παρακάτω περιορισμοί

1. Για κάθε IP δημιουργούνται τρεις κάδοι
    1. Ένας κάδος με μέγεθος 1 token και περίοδο ανανέωσης 1 μέρα χρησιμοποιείται για περιορισμό του χρήστη να στέλνει το πολύ ένα email verification ανά μέρα.
    2. Δεύτερος κάδος έχει ίδιο configuration με το παραπάνω και χρησιμοποιείται για περιορισμό του χρήστη να ζητάει το πολύ μία επαναφορά κωδικού επειδή η επαναφορά κωδικού απαιτεί επίσης χρήση του email service provider.
    3. Τρίτος κάδος με μέγεθος 5 tokens, περίοδο ανανέωσης 1 δευτερόλεπτο και Greedy στρατηγική ανανέωσης των token χρησιμοποιείται για όλη την εφαρμογή ως μέθοδος προστασίας από DoS/DDoS επιθέσεις.

3.6 Επίπεδο Παρουσίασης

Το επίπεδο παρουσίας περιέχει το API με όλα τα endpoints.

Τα endpoints είναι οργανωμένα σε Spring MVC controllers βάση των οντοτήτων της εφαρμογής

1. CustomerController περιέχει τα endpoints που αφορούν τους customers(χρήστες) της εφαρμογής.
2. HotelController περιέχει endpoints που αφορούν τα ξενοδοχεία.
3. MonitorListController περιέχει endpoints που αφορούν τις λίστες δωματίων.
4. PriceController περιέχει endpoints που αφορούν τις τιμές δωματίων.
5. RoomController περιέχει endpoints που αφορούν τα δωμάτια
6. SubscriptionController περιέχει endpoints που αφορούν τις συνδρομές.

Για κάθε endpoint που προσφέρει η εφαρμογή υπάρχει αντίστοιχο documentation μέσα στο αρχείο ENDPOINTS.md.

Μέσα στο αρχείο για κάθε endpoint ορίζονται τα

1. HTTP μέθοδος που υποστηρίζει.
2. Το URI path.
3. Σύντομη περιγραφή του endpoint.
4. Τον κανόνα εξουσιοδότησης που ισχύει για το endpoint.
5. Το response Content-Type.
6. Παράδειγμα επιτυχημένου response.
7. Τα δυνατά HTTP status codes που μπορεί να επιστρέψει με επεξήγηση.
8. Το υποστηριζόμενο request Content-Type.
9. Παράδειγμα σώματος request.
10. Περιγραφή για τις query παραμέτρους.

3.7 Επίπεδο Εφαρμογής

Το επίπεδο εφαρμογής περιέχει την υλοποίηση των βασικών απαιτήσεων του προγράμματος.

3.7.1 Εγγραφή του χρήστη στο σύστημα

Η εγγραφή του χρήση στο σύστημα γίνεται μέσα από CustomerService.register μέθοδο η οποία δέχεται την φόρμα εγγραφής και φτιάχνει καινούριο χρήστη στο σύστημα.

Η φόρμα εγγραφής περιέχει τα πεδία: JWT, κωδικός πρόσβασης χρήστη, όνομα χρήστη και επαναλαμβανόμενος κωδικός πρόσβασης.

Πριν γίνει εγγραφή, ο χρήστης πρέπει να επαληθεύσει ότι είναι κάτοχος του συγκεκριμένου email address όπως περιγράφεται στο JWS Email Verification.

Για την φόρμα εγγραφής επιβάλλονται οι παρακάτω κανόνες επικύρωσης

1. Το όνομα χρήστη δεν μπορεί να είναι κενό.
2. Ο κωδικός χρήστη πρέπει να έχει τουλάχιστον 8 χαρακτήρες, πρέπει να περιέχει τουλάχιστον έναν αριθμό(0,1,2..,9) και τουλάχιστον ένα αλφαβητικό(A-Z,a-z).
3. Ο επαναλαμβανόμενος κωδικός πρόσβασης πρέπει να είναι ίδιος με τον κωδικό πρόσβασης.

Η εφαρμογή δεν επιτρέπει δύο χρήστης με ίδιο email στο σύστημα.

Τέλος, η εφαρμογή δεν αποθηκεύει τον κωδικό χρήστη ως απλό κείμενο, αλλά αποθηκεύει hashed κωδικό χρησιμοποιώντας το bcrypt αλγόριθμο.

Ο bcrypt αλγόριθμος χρησιμοποιείται για hashing των κωδικών πρόσβασης. Είναι ασφαλές από rainbow table επιθέσεις επειδή χρησιμοποιεί random salt το οποίο αποθηκεύεται μαζί με την hash τιμή.

Ένα άλλο πλεονέκτημα του bcrypt είναι ότι είναι προσαρμοστικός αλγόριθμος: ο αριθμός των round που εκτελεί ο αλγόριθμος είναι παραμετροποιήσιμος και αυτή η ιδιότητα τον κάνει ασφαλές από brute force επιθέσεις. Συγκεκριμένα, αυξάνοντας τον αριθμό των round,ο αλγόριθμος εκτελείται πιο αργά με αποτέλεσμα το brute force να χρειάζεται περισσότερο χρόνο.

3.7.2 Αυθεντικοποίηση

Η αυθεντικοποίηση του χρήστη γίνεται σε δύο στάδια

1. Πρώτα ο χρήστης πρέπει να υποβάλλει φόρμα αυθεντικοποίησης χρησιμοποιώντας το email και τον κωδικό χρήστη.
2. Αν η αυθεντικοποίηση με email και κωδικό πρόσβασης είναι επιτυχημένη, η εφαρμογή επιστρέφει JWS το οποίο ο χρήστης μπορεί να χρησιμοποιήσει για διάρκεια 24 ωρών για να κάνει πρόσβαση στους προστατευμένους πόρους της εφαρμογής.

3.7.3 Δημιουργία Λίστας Δωματίων

Η εφαρμογή επιτρέπει τον αυθεντικοποιημένο χρήστη να δημιουργήσει λίστα δωματίων. Η φόρμα δημιουργίας της λίστας περιέχει το όνομα της λίστας, τα id των δωματίων και τα distances.

Η εφαρμογή δεν επιτρέπει χρήση ίδιου ονόματος λίστας ανά χρήστη. Δηλαδή, δύο χρήστες μπορούν να έχουν λίστα με ίδιο όνομα, αλλά ένας χρήστης δεν μπορεί να έχει δύο λίστες με το ίδιο όνομα.

Distances είναι πίνακας ακεραίων αριθμών με τον οποίο ο χρήστης δηλώνει πόσες μέρες μπροστά ο scraper πρέπει να κάνει scrap τα δεδομένα των δωματίων. Για παράδειγμα, αν distances είναι [7,15], τότε ο scraper που θα κάνει scrap στις 01-01-2024 για την συγκεκριμένη λίστα θα ψάξει δεδομένα δωματίων για 08-01-2024 και 21-01-2024 αντίστοιχα.

3.7.4 Τιμές Δωματίων

Η εφαρμογή επιτρέπει τον αυθεντικοποιημένο χρήστη να ζητήσει τις τιμές δωματίου που έχουν συγκεντρωθεί από τον σκράπερ. Οι τιμές επιστρέφονται σελιδοποιημένες.

3.7.5 Ιστορικό Ιδιοτήτων Δωματίων

Η εφαρμογή επιτρέπει τον αυθεντικοποιημένο χρήστη να ζητήσει ιστορικό αλλαγών ιδιοτήτων ενός δωματίου.

Το ιστορικό επιστρέφεται σελιδοποιημένο και περιέχει εγγραφές με κάθε εγγραφή να περιέχει τα εξής δεδομένα

1. Χρονική σήμανση που δηλώνει πότε έγινε η αλλαγή στις ιδιότητες.
2. Τις ιδιότητες που προστέθηκαν κατά την αλλαγή των ιδιοτήτων του δωματίου.
3. Τις ιδιότητες που αφαιρέθηκαν κατά την αλλαγή των ιδιοτήτων του δωματίου.

3.7.6 Συνδρομές

Η συνδρομή του χρήστη προσδιορίζεται από τον ρόλο που έχει. Όλοι οι χρήστες μετά την εγγραφή έχουν τον USER ρόλο. Έπειτα ο χρήστης μπορεί να αναβαθμίσει την συνδρομή του σε BASIC ή PREMIUM.

Η αναβάθμιση δεν κοστίζει τίποτα και αποτελεί προσομοίωση του τρόπου λειτουργίας της συνδρομής.

Για κάθε συνδρομή ορίζονται περιορισμοί που αφορούν τον αριθμό τον επιτρεπτών λιστών και τον αριθμό των δωματίων που μπορεί να έχει μία λίστα. Συγκεκριμένα, ισχύουν οι παρακάτω περιορισμοί.

1. Χρήστης με ρόλο/συνδρομή USER μπορεί να δημιουργήσει το πολύ 10 λίστες, και κάθε λίστα μπορεί να έχει το πολύ 10 δωμάτια.
2. Χρήστης με ρόλο/συνδρομή BASIC μπορεί να δημιουργήσει το πολύ 20 λίστες, και κάθε λίστα μπορεί να έχει το πολύ 20 δωμάτια.
3. Χρήστης με ρόλο/συνδρομή PREMIUM μπορεί να δημιουργήσει το πολύ 50 λίστες, και κάθε λίστα μπορεί να έχει απεριόριστο αριθμό δωματίων.

3.8 Επίπεδο Δεδομένων

Για την αποθήκευση δεδομένων της εφαρμογής χρησιμοποιήθηκε RDBMS MySQL βάση δεδομένων.

Το σχήμα της βάσης δεδομένων της εφαρμογής περιέχει τους παρακάτω πίνακες.

3.8.1 Χρήστης

Για κάθε χρήστη αποθηκεύω

- Μοναδικό αναγνωριστικό(id).
- Το όνομα του χρήστη.
- Το email.
- Τον κωδικό πρόσβασης(hashed).
- Τον ρόλο του χρήστη.

3.8.2 Ξενοδοχείο

Για κάθε ξενοδοχείο αποθηκεύω

- Μοναδικό αναγνωριστικό(id).
- Το όνομα του ξενοδοχείου.
- Το booking URL του ξενοδοχείου.
- Την περιγραφή του δωματίου.
- Την τοποθεσία του δωματίου.
- Τον τύπο του ξενοδοχείου: για παράδειγμα, ξενοδοχείο ή διαμέρισμα.
- Το score που έχει το ξενοδοχείο στο booking.

Το URL είναι μοναδικό για όλα τα ξενοδοχεία: δύο ξενοδοχεία δεν μπορούν να έχουν ίδιο URL.

Το σκορ του ξενοδοχείου πρέπει να είναι μέσα στο εύρος [0,10].

Πριν την αποθήκευση του URL, από το URL αφαιρείται το query κομμάτι και το κομμάτι που αφορά την γλώσσα παρουσίασης.

Για παράδειγμα, αυτά τα δύο URL αναφέρονται στο ίδιο ξενοδοχείο: https://www.booking.com/hotel/gr/alpe-luxury-accommodation-penelope-collection.en-gb.html και https://www.booking.com/hotel/gr/alpe-luxury-accommodation-penelope-collection.el.html. Η διαφορά είναι στην γλώσσα παρουσίασης.

Μετά την αφαίρεση της γλώσσας από το URL, η τιμή που αποθηκεύεται είναι https://www.booking.com/hotel/gr/alpe-luxury-accommodation-penelope-collection.html.

Για την αφαίρεση της γλώσσας από το URL, παρατήρησα ότι η γλώσσα δηλώνεται πριν το .html και χωρίζεται από το υπόλοιπο URL με τελεία(.).

3.8.3 Δωμάτιο

Για κάθε δωμάτιο αποθηκεύω

- Μοναδικό αναγνωριστικό(id).
- Το αναγνωριστικό του ξενοδοχείου που δηλώνει το ξενοδοχείο στο οποίο ανήκει το δωμάτιο.
- Περιγραφή δωματίου.
- Το όνομα του δωματίου.
- Τις ιδιότητες του δωματίου.

Το όνομα του δωματίου πρέπει να είναι μοναδικό ανά ξενοδοχείο: δύο ξενοδοχεία μπορούν να έχουν δωμάτια με ίδιο όνομα, αλλά ένα ξενοδοχείο δεν μπορεί να έχει δύο δωμάτια με ίδιο όνομα.

3.8.4 Επιλογή Δωματίου

Η επιλογή δωματίου ή προσφορά δωματίου αποτελεί προσφορά δωματίου. Ένα δωμάτιο μπορεί να προσφέρεται σε δύο ή περισσότερες επιλογές. Για παράδειγμα, μια προσφορά μπορεί να περιέχει γεύμα ενώ μια άλλη όχι.

Για κάθε προσφορά/επιλογή αποθηκεύω

- Μοναδικό αναγνωριστικό(id).
- Το αναγνωριστικό δωματίου που δηλώνει το δωμάτιο για το οποίο ισχύει η προσφορά/επιλογή.
- Τον αριθμό των επισκεπτών που μπορούν να μένουν στο δωμάτιο(sleeps).
- Ημερομηνία εισαγωγής της προφοράς στην βάση δεδομένων.

- Πολιτική ακύρωσης κράτησης που προσφέρεται από την προσφορά/επιλογή.
- Το γεύμα που προσφέρεται από την προσφορά/επιλογή.
- Τις ιδιότητες που έχει η προσφορά

Η πολιτική ακύρωσης μπορεί να είναι μία από παρακάτω

- Δωρεάν ακύρωση.
- Ακύρωση με επιστροφή χρημάτων.
- Ακύρωση χωρίς επιστροφή χρημάτων.
- Ακύρωση με επιστροφή μέρους χρημάτων.
- Άγνωστο – δεν είναι δηλωμένη ή δεν αναγνωρίζεται από την εφαρμογή.

Η πολιτική ακύρωσης και το γεύμα που προσφέρεται από την προσφορά/επιλογή αποτελούν και αυτά ιδιότητες της προσφοράς/επιλογής αλλά αποθηκεύονται ως στήλες στον πίνακα προσφορών επειδή συμμετέχουν στον ορισμού της μοναδικότητας της προσφοράς.

Ένα από τα προβλήματα που βρέθηκε κατά την ανάπτυξη της εφαρμογής ήταν ο προσδιορισμός της μοναδικότητας για την οντότητα που αναπαριστά την προσφορά δωματίου. Δηλαδή,ο προσδιορισμός των στηλών που μοναδικά προσδιορίζουν την προσφορά στον πίνακα.

Από όλες τις ιδιότητες που μπορεί να έχει μία προσφορά και από άλλα δεδομένα που αποθηκεύονται για την προσφορά, επέλεξα τις παρακάτω για τον προσδιορισμό της μοναδικότητας ως πιο σημαντικές από την άποψη του χρήστη και της τιμής της προσφοράς

- Το προσφερόμενο γεύμα: αν είναι δωρεά ή με επιπλέον πληρωμή ή αν δεν προσφέρεται καθόλου.
- Ο αριθμός των επισκεπτών που επιτρέπει η προσφορά.
- Η πολιτική ακύρωσης κράτησης.
- Το id του δωματίου για το οποίο ισχύει η προσφορά.

Αυτή η επιλογή είχε τις εξής συνέπειες

- Επιτρέπει την προσθήκη ή αφαίρεση ιδιοτήτων που δεν συμμετέχουν στον προσδιορισμό μοναδικότητας στις προσφορές χωρίς αυτό να οδηγεί στη δημιουργία καινούριας προσφοράς.
- Η αλλαγή μιας,οποιασδήποτε,ιδιότητας που συμμετέχει στον προσδιορισμό μοναδικότητας οδηγεί στη δημιουργία καινούριας προσφοράς και όλες οι καινούριες τιμές σχετίζονται με την καινούρια προσφορά.

3.8.5 Ιδιότητα Δωματίου και Ιδιότητα Επιλογής

Οι ιδιότητες δωματίων και προσφορών/επιλογών αποθηκεύονται ως ξεχωριστές οντότητες μέσα στην βάση. Ο λόγος που δεν αποθηκεύονται ως weak οντότητες[10] είναι ότι αποθηκεύοντας τις ιδιότητες σε πίνακες ως οντότητες, χρησιμοποιείται λιγότερος χώρος καθώς πολλά δωμάτια και προσφορές έχουν πολλές κοινές ιδιότητες(WiFi,TV κτλ).

Για κάθε ιδιότητα δωματίου και επιλογής αποθηκεύω

- Το αναγνωριστικό(id)
- Την τιμή της ιδιότητας

3.8.6 Λίστα Δωματίων

Οι λίστες δωματίων δημιουργούνται από τους χρήστες της εφαρμογής. Για κάθε λίστα αποθηκεύω

---

10 Weak οντότητα είναι οντότητα που δεν έχει αναγνωριστικό και η ζωή της οποίας στη βάση εξαρτάται από την οντότητα γονέα. Η weak οντότητα έχει αναγνωρίζεται στη βάση από χρησιμοποιώντας τον γονέα.

- Μοναδικό αναγνωριστικό(id).
- Το όνομα της λίστας.
- Τα δωμάτια τα οποία ανήκουν στην λίστα.
- Την κατάσταση στην οποία βρίσκεται η λίστα. Μπορεί να βρίσκεται σε μία από τις δύο καταστάσεις: ενεργοποιημένη ή απενεργοποιημένη.
- Το αναγνωριστικό του χρήστη που δημιούργησε την λίστα.
- Ημερομηνία δημιουργίας της λίστας.
- Τον τύπο παρακολούθησης της λίστας(δεν χρησιμοποιείται στην εφαρμογή)
- Τα distances που ορίζουν scraping παραμέτρους

Το όνομα της λίστας είναι μοναδικό ανά χρήστη. Αυτό σημαίνει ότι ένας χρήστης δεν μπορεί να έχει δύο λίστες με το ίδιο όνομα.

Οι λίστες άμεσα επηρεάζουν το σκράπινγ των ξενοδοχείων. Για να γίνει σκραπ κάποιο ξενοδοχείοο, πρέπει τουλάχιστον ένας χρήστης να δείξει ενδιαφέρον για αυτό.

Ο τρόπος που οι χρήστες δείχνουν ότι έχουν ενδιαφέρον ένα ξενοδοχείο να συμπεριλαμβάνεται στα ξενοδοχεία για τα οποία μαζεύονται δεδομένα είναι μέσα από τις λίστες δωματίων. Όταν ο χρήστης δημιουργεί λίστα από δωμάτια τα οποία θέλει να παρακολουθούνται, τα ξενοδοχεία στα οποία ανήκουν τα δωμάτια συμπεριλαμβάνονται στην λίστα ξενοδοχείων για τα οποία μαζεύονται δεδομένα.

3.8.7 Τιμές

Οι τιμές που μαζεύονται για τα ξενοδοχεία(πιο συγκεκριμένα, οι τιμές των προσφορών δωματίων) αποθηκεύονται ως οντότητες στη βάση. Για κάθε τιμή αποθηκεύω

1. Μοναδικό αναγνωριστικό(id).
2. Ποσότητα. Η ποσότητα δηλώνει πόσες προσφορές υπάρχουν με αυτή την τιμή.
3. Τιμή της προσφοράς.
4. Timestamp. Δηλώνει την ημερομηνία που έγινε το σκράπινγ της τιμής.
5. Απόσταση ημερών. Η απόσταση ημερών δηλώνει την ημερομηνία η οποία χρησιμοποιήθηκε στο σκράπινγ. Συγκεκριμένα,η ημερομηνία για την τιμή είναι = timestamp + απόσταση ημερών. Έτσι, αν το timestamp είναι 2023-12-01 και απόσταση 7,τότε το σκράπινγ έγινε για την ημερομηνία 2023-12-08.
6. Μέρες για την ακύρωση κράτησης. Δηλώνει πόσες μέρες έχει ο χρήστης μετά την κράτηση για να την ακυρώσει.
7. Το αναγνωριστικό της προσφοράς για την οποία ισχύει η τιμή.

4 Συμπεράσματα

Στόχος της εργασία μου ήταν η ανάπτυξη web εφαρμογής,σε συνεργασία με συμφοιτητές, η οποία θα ήταν χρήσιμη σε διαχειριστές ξενοδοχείων.

Η εργασία μου επέτρεψε να εφαρμόσω τις γνώσεις που έχω αποκτήσει από τις σπουδές μου: σχεσιακές βάσεις δεδομένων, πρωτοκολλά επικοινωνίας, ασφάλεια, γλώσσες προγραμματισμού,frameworks,testing και άλλα.

Η εργασία μπορεί να βελτιωθεί περαιτέρω. Ορισμένα πράγματα στα οποία μπορεί να βελτιωθεί η εφαρμογή

1. Ασφάλεια. Αυτό αφορά CSRF προστασία η οποία στην τρέχουσα έκδοση είναι απενεργοποιημένη, αλλά και διαχείριση των κλειδιών.
2. Καλύτερη υλοποίηση για συνδρομές καθώς η τρέχουσα έκδοση περιέχει περιορισμένη υλοποίηση για συνδρομές.
3. Βελτίωση του σχήματος της βάσης δεδομένων με τη δημιουργία των indexes για καλύτερη απόδοση των ερωτημάτων.
4. Για καλύτερη ανάλυση των δεδομένων, η εφαρμογή μπορεί να βελτιωθεί χρησιμοποιώντας εργαλεία συγκεκριμένα σχεδιασμένα για αυτό το σκοπό όπως Apache Spark, Apache Flink, Hadoop κ.α.
5. Βελτίωση της αρχιτεκτονικής της εφαρμογής ώστε να μπορεί να λειτουργεί για διαφορετικές ιστοσελίδες ξενοδοχείων και όχι μόνο για booking.com.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1: Understanding Gradle’s Project Structure , Gradle Documentation

2: Understanding the Build script , Gradle Documentation

3: Viewing available Tasks , Gradle Documentation

4: Task Graph , Gradle Documentation

5: Build Phases , Gradle Documentation

6: Incremental Builds , Gradle Documentation

7: Images , Docker Documentation

8: Docker architecture , Docker Documentation

9: , Docker Documentation

10: Docker Documentation ,

11: CNCF Distribution ,

12: Spring Framework Documentation ,

12: Spring Java Docs ,

14: RFC6749 ,

14: Statement Types , JPA Specification