ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
«Μηχανική Λογισμικού για Διαδικτυακές & Φορητές Εφαρμογές»

Αρχιτεκτονικά και Σχεδιαστικά πρότυπα ως βάση για την αρχιτεκτονική και λεπτομερή σχεδίαση, υλοποίηση & έλεγχο εφαρμογών και APIs.

Τσεντίδης Διονύσιος

Επιβλέπων Καθηγητής: Δρ. Κακαρόντζας Γεώργιος

Κριτικός Αναγνώστης: Σάββας Ηλίας

Πινακας Περιεχομενων

1. ΕΙΣΑΓΩΓΗ

1.1. Πρόλογος

Στον κλάδο της Πληροφορικής και συγκεκριμένα στην ανάπτυξη λογισμικού χρησιμοποιούμε την έκφραση αρχιτεκτονική λογισμικού. Η αρχιτεκτονική ενός προγράμματος λογισμικού θα μπορούσαμε να πούμε πως είναι μια μεταφορά ανάλογη με την αρχιτεκτονική ενός κτιρίου, σαν ένα προσχέδιο. Στόχος της είναι η οργάνωση ενός συστήματος λογισμικού. Καθώς μιλάμε για οργάνωση αναφερόμαστε στην επιλογή κατάλληλων δομικών στοιχείων, τη σύνδεση μεταξύ τους και τη συμπεριφορά τους καθώς εντάσσονται σε μεγαλύτερα υποσυστήματα.

Η αρχιτεκτονική λογισμικού αντιπροσωπεύει τα ακόλουθα:

- Πολλούς ενδιαφερόμενους: ένα σύστημα λογισμικού πρέπει να εξυπηρετεί όλους τους ενδιαφερόμενους που το χρησιμοποιούν, από το χρήστη μέχρι και τον διαχειριστή.
- Διαχωρισμός των ανησυχιών: οι ανησυχίες που εκφέρουν οι ενδιαφερόμενοι αντιμετωπίζονται με τη μοντελοποίηση.
- Ποιοτικό σχεδιασμό: η ανοχή σφαλμάτων, συμβατότητα με προηγούμενες εκδόσεις, επεκτασιμότητα, αξιοπιστία, συντήρηση, διαθεσιμότητα, ασφάλεια, χρηστικότητα είναι μερικά από τα ποιοτικά χαρακτηριστικά της αρχιτεκτονικής. Πολλές φορές οι ανησυχίες των ενδιαφερόμενων ταυτίζονται σύμφωνα με αυτά τα χαρακτηριστικά.
- Επαναλαμβανόμενο στυλ: για την αντιμετώπιση προβλημάτων τα οποία επαναλαμβάνονται έχουν δημιουργηθεί κάποιοι τρόποι αντιμετώπισής τους σε διάφορα επίπεδα αφαίρεσης.
- Εννοιολογική ακεραιότητα: όταν υπάρχουν νέες προσθήκες στο σύστημα, ο αρχιτέκτονας πρέπει να διασφαλίσει ότι είναι σύμφωνες με την αρχιτεκτονική.
- Γνωστικοί περιορισμοί: οι οργανισμοί που σχεδιάζουν συστήματα υποχρεούνται να μοντελοποιούν τις δομές και τους μηχανισμούς επικοινωνίας ώστε να αντικατοπτρίζονται οι περιορισμοί του περιβάλλοντος.

Συνήθης φαινόμενο στην αρχιτεκτονική λογισμικού είναι να χρησιμοποιούμε κάποια πρότυπα. Πρότυπο είναι ένα πλαίσιο το οποίο μας βοηθάει στο να διασπάσουμε ένα μεγάλο και σύνθετο πρόβλημα που αντιμετωπίζουμε, σε μικρότερα και απλά μέρη. Στην ουσία μας παρέχει μια γενική λύση για μια κατηγορία προβλημάτων, την οποία μπορούμε να υιοθετήσουμε και να την προσαρμόσουμε στο πρόβλημα που αντιμετωπίζουμε. Σε γενικές γραμμές τα

Αρχιτεκτονικά πρότυπα είναι η βάση σε ένα πρόγραμμα, ενώ τα Σχεδιαστικά πρότυπα χρησιμοποιούνται σε μικρότερο βαθμό στο προγραμματισμό.

Η χρησιμότητα των Αρχιτεκτονικών προτύπων είναι η βοήθεια που μας παρέχουν για την επίλυση κάποιων προβλημάτων σε μια αρχιτεκτονική λογισμικού, σε ένα συγκεκριμένο περιβάλλον. Ένα αρχιτεκτονικό πρότυπο είναι μια επιτυχημένη τεχνική η οποία έχει δοκιμαστεί σε επαρκή χρονικό διάστημα και έχει αποδείξει την αξία της. Τα αρχιτεκτονικά πρότυπα χρησιμοποιούνται για τη θεμελιώδη δομική οργάνωση για συστήματα λογισμικού. Με αυτά μπορούμε να αντιμετωπίσουμε διάφορα ζητήματα στη μηχανική λογισμικού, όπως περιορισμούς από την απόδοση των υλικών ενός υπολογιστή και την μείωση επιχειρηματικού κινδύνου.

Από την άλλη τα Σχεδιαστικά πρότυπα, έχουν δημιουργηθεί για την επίλυση προβλημάτων που αντιμετωπίζουμε σε επίπεδο λογισμικού σε ένα περιβάλλον. Θεωρούνται ως οι καλύτερες τεχνικές όπου μπορεί να τις αξιοποιήσει ένας προγραμματιστής για να λύσει κάποιο πρόβλημα που αντιμετωπίζει στο πρόγραμμα σε επίπεδο λογισμικού. Ο προγραμματιστής πρέπει να προσαρμόσει το πρότυπο στα δικά του δεδομένα και περιβάλλον. Τα Σχεδιαστικά πρότυπα μας βοηθάνε στο να γράφουμε αποτελεσματικό κώδικα, εύκολο στη συντήρηση και την ανάγνωση και με υψηλή επαναχρησιμοποίηση.

1.2. Περίληψη

Ο σκοπός αυτής της διπλωματικής εργασίας είναι η μελέτη της εφαρμογής αρχιτεκτονικών και σχεδιαστικών προτύπων. Θα γίνει υλοποίηση αυτών των προτύπων σε μία σύγχρονη εφαρμογή λογισμικού, η οποία θα παρέχει προγραμματιστικές διασυνδέσεις. Πιο συγκεκριμένα θα αναπτυχθεί μια εφαρμογή για συνταγές με σκοπό να γίνει αντιληπτή η αρχιτεκτονική σχεδίαση και η υλοποίηση ολόκληρης της εφαρμογής από την αρχή, για τη διαχείριση και εκτέλεση συνταγών μαγειρικής. Η εφαρμογή αυτή θα τρέχει σε κινητές συσκευές που υποστηρίζουν Android. Οι συνταγές θα ανακτώνται από έναν εξυπηρετητή με τη χρήση API που θα αναπτυχθεί για το σκοπό αυτό. Στον εξυπηρετητή οι συνταγές θα εισάγονται σε μια τοπική βάση δεδομένων από το χρήστη. Δεν θα αναπτυχθεί εξωτερική βάση δεδομένων που χρειάζεται πρόσβαση στο διαδίκτυο.

Για την ανάπτυξη μιας εφαρμογής χρησιμοποιούμε διάφορες τεχνολογίες και εργαλεία τα οποία συνήθως είναι ευρέως γνωστά ανάμεσα στους προγραμματιστές. Πιο συγκεκριμένα θα γίνει αναφορά και ανάλυση σε εργαλεία και τεχνολογίες για κινητές εφαρμογές, οι οποίες χρησίμευσαν στην εφαρμογή.

Η αρχιτεκτονική σχεδίαση θα γίνει με βάση τα κρίσιμα ποιοτικά χαρακτηριστικά και με βάση αρχιτεκτονικά πρότυπα που ικανοποιούν αυτά τα χαρακτηριστικά. Θα διατυπωθούν οι αρχιτεκτονικές (ποιοτικές) απαιτήσεις που είναι σημαντικές για την εφαρμογή(π.χ. ευχρηστία, ταχύτητα κ.α). Στη συνέχεια

θα δοθεί με σαφήνεια η σχεδίαση σε αρχιτεκτονικό επίπεδο με βάση επιλεγμένα σχετικά αρχιτεκτονικά πρότυπα και να δειχθεί με σαφή τρόπο πως αυτά τα πρότυπα ικανοποιούν τα ποιοτικά χαρακτηριστικά.

Εκτός από την αρχιτεκτονική σχεδίαση, θα γίνει σχεδίαση σε επίπεδα κλάσεων με γνώμονα την ευχρηστία και εξέλιξη της εφαρμογής. Για αυτό το σκοπό θα γίνει χρήση σχεδιαστικών προτύπων (design patterns). Θα αναλυθεί ο τρόπος με τον οποίο τα σχεδιαστικά πρότυπα διευκολύνουν την εξέλιξη και τη συντήρηση της εφαρμογής ή ο τρόπος με τον οποίο επιλύουν συγκεκριμένα σχεδιαστικά προβλήματα.

Επίσης θα γίνει αναφορά στη τοπική βάση δεδομένων που θα χρησιμοποιηθεί. Θα δούμε τη σχεδίαση και την υλοποίηση της, χρησιμοποιώντας τη βιβλιοθήκη Room και ποια τα πλεονεκτήματα της. Επιπρόσθετα θα επισημανθεί πως συνδέεται η τοπική βάση δεδομένων με την εφαρμογή και με ποιους τρόπους μπορούμε να ανακτήσουμε τις πληροφορίες που περιέχει.

2. ΤΕΧΝΟΛΟΓΙΕΣ ΠΟΥ ΧΡΗΣΙΜΟΠΟΙΗΘΗΚΑΝ

Για την ανάπτυξη της εφαρμογής χρειάστηκε να χρησιμοποιηθούν κάποιες τεχνολογίες και εργαλεία. Σε αυτό το κεφάλαιο θα γίνει επισήμανση και ανάλυση αυτών.

2.1. Android Software Development Kit (SDK)

Το Android Software Development Kit είναι μία συλλογή από εργαλεία και βιβλιοθήκες το οποίο αναπτύχθηκε από την εταιρεία Google για συσκευές που χρησιμοποιούν τη πλατφόρμα Android. Περιλαμβάνει όλα τα απαραίτητα εργαλεία για την ανάπτυξη μιας εφαρμογής σε Android, όπως πρόγραμμα εντοπισμού σφαλμάτων, βιβλιοθήκες, εξομοιωτή συσκευών, δείγματα κώδικα και οδηγούς. Η πρώτη διαθέσιμη έκδοση ήταν τον Οκτώβριο του 2009. Έχει γραφτεί στην αντικειμενοστραφή γλώσσα προγραμματισμού Java. Κάθε φορά που η Google βγάζει μια νέα έκδοση για το Android ή κάποια ενημέρωση, ένα αντίστοιχο SDK κυκλοφορεί επίσης το οποίο είναι διαθέσιμο για να κατεβάσουν και να εγκαταστήσουν οι προγραμματιστές.

2.2. Android Studio

Το Android Studio αναπτύχθηκε από μία Τσέχικη εταιρεία την JetBrains s.r.o (πρώην IntelliJ Software s.r.o) της οποίας τα εργαλεία απευθύνονται σε προγραμματιστές λογισμικού και διαχειριστές έργων. Μέχρι τα τέλη του 2014 οι

προγραμματιστές για την ανάπτυξη λογισμικού σε Android χρησιμοποιούσαν ως ολοκληρωμένο περιβάλλον ανάπτυξης(IDE) το Eclipse με τη βοήθεια εργαλείων ανάπτυξης Android(ADT). Από το 2015 η Google ανακοίνωσε πως το Android Studio θα είναι το επίσημο ολοκληρωμένο περιβάλλον ανάπτυξης εφαρμογών για Android, αντικαθιστώντας το Eclipse.

Ένα κύριο χαρακτηριστικό του Android Studio είναι η αυτόματη αποθήκευση κάθε αλλαγής που γίνεται στο project. Μερικά άλλα χαρακτηριστικά είναι:

- Ανακατασκευή και γρήγορες επιδιορθώσεις.
- Εργαλεία για την απόδοση, τη χρηστικότητα, τη συμβατότητα έκδοσης και άλλα προβλήματα.
- Οδηγούς για κοινά σχέδια και στοιχεία.
- Μια διάταξη όπου ο προγραμματιστής μπορεί να κάνει drag and drop στοιχεία διεπαφής χρήστη.
- Προεπισκόπηση διατάξεων σε πολλές οθόνες.
- Υποστήριξη για τη δημιουργία εφαρμογών Android Wear.
- Εικονική συσκευή Android(Emulator) για εκτέλεση και εντοπισμό σφαλμάτων.

Το περιβάλλον αυτό είναι διαθέσιμο για προγραμματιστές που χρησιμοποιούν Windows, macOS και Linux.

2.3. AndroidX

Το AndroidX περιλαμβάνει πολλές από τις βιβλιοθήκες που χρησιμοποιούνται στην εφαρμογή. Είναι η τελευταία έκδοση στην αρχική βιβλιοθήκη υποστήριξης Android, η οποία δεν διατηρείται πλέον. Οι βιβλιοθήκες του AndroidX αποστέλλονται ξεχωριστά από την πλατφόρμα Android. Επίσης παρέχουν συμβατότητα προς τα πίσω σε όλες τις εκδόσεις Android. Τα πακέτα του AndroidX αντικαθιστούν πλήρως τη παλιά βιβλιοθήκη υποστήριξης του Android συν ότι υπάρχουν και νέες βιβλιοθήκες. Όλα τα πακέτα αρχίζουν με τη συμβολοσειρά androidx.*. Επιπρόσθετα υπάρχει πλήρη χαρτογράφηση όλων των παλαιών κλάσεων και στοιχείων με τα νέα. Τα AndroidX πακέτα διατηρούνται και ενημερώνονται τακτικά σε σχέση με τη παλιά βιβλιοθήκη Android, η οποία σταμάτησε να ενημερώνεται.

2.4. Η γλώσσα προγραμματισμού Java

Η Java είναι μια γλώσσα προγραμματισμού που σχεδιάστηκε από την εταιρεία πληροφορικής Sun Microsystems και συγκεκριμένα πατέρας της θεωρείτε ο James Gosling. Η επίσημη εμφάνιση της καταγράφηκε το Μάρτιο του 1995 στο συνέδριο Sun World 1995. Αυτή η γλώσσα κατατάσσεται στις αντικειμενοστραφείς γλώσσες προγραμματισμού και είναι μια από τις δημοφιλέστερες γλώσσες στην εποχή μας. Στις 27 Απριλίου του 2010 η εταιρία λογισμικού Oracle Corporation εξαγοράζει την εταιρία Sun Microsystems καθώς και όλες τις τεχνολογίες της. Η εξαγορά αυτή θεωρήθηκε σημαντική για το μέλλον της Java.

Ως αντικειμενοστραφής γλώσσα βασίζεται σε κλάσεις και αντικείμενα. Η κλάση είναι σαν ένα σχεδιάγραμμα για τη δημιουργία αντικειμένων. Μια κλάση είναι σαν να περιγράφει μια οντότητα. Η κλάση περιλαμβάνει πεδία(fields) και μεθόδους(methods). Τα πεδία αντιστοιχούν σε ιδιότητες της οντότητας ενώ οι μέθοδοι είναι οι λειτουργίες που μπορεί να κάνει μια οντότητα. Κάθε μέλος της κλάσης προσδιορίζεται από έναν προσδιοριστή προσπέλασης(access modifier). Επίσης κάθε κλάση μπορεί να έχει έναν ή περισσότερους κατασκευαστές για τη δημιουργία αντικειμένων.

Παρακάτω (εικόνα 1) απεικονίζεται ένα παράδειγμα μια κλάσης η οποία παριστάνει την οντότητα χρήστης (User). Ο κάθε χρήστης προσδιορίζεται από τα πεδία userId, name και age. Στην κλάση περιλαμβάνονται μέθοδοι ανάγνωσης και τροποποίησης των πεδίων της. Επίσης υπάρχουν και δύο κατασκευαστές για τη δημιουργία νέων αντικειμένων User.

```java
public class User {
    //Πεδία της κλάσης
    private long userId;
    private String name;
    private int age;

    //Κατασκευαστής χωρίς παραμέτρους
    public User() {
    }
    //Κατασκευαστής με παραμέτρους
    public User(long userId, String name, int age) {
        this.userId = userId;
        this.name = name;
        this.age = age;
    }
    //Μέθοδοι που επιστρέφουν τις τιμές των πεδίων του αντικειμένου
    public long getUserId() {
        return userId;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
    //Μέθοδοι που θέτουν τιμές στα πεδία του αντικειμένου
    public void setUserId(long userId) {
        this.userId = userId;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Εικόνα 1 – Java class example

Όλα τα εργαλεία που χρειάζεται κάποιος για να γράψει σε Java είναι δωρεάν. Υπάρχει άφθονο υλικό από οδηγούς μέχρι και βιβλιοθήκες που μπορούν να φανούν χρήσιμα. Το μόνο που χρειάζεται είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης(IDE) που βοηθάει τη δουλειά του προγραμματιστή, ιδιαίτερα στον εντοπισμό σφαλμάτων.

2.5. Αποθήκευση δεδομένων - Room

Σε εφαρμογές που χρησιμοποιείται μεγάλος όγκος δεδομένων επωφελούμαστε σε μεγάλο βαθμό από τη διατήρηση αυτών των δεδομένων σε μια τοπική βάση.

Ένα τρανταχτό παράδειγμα είναι η προσωρινή αποθήκευση κάποιων δεδομένων στη τοπική βάση έτσι ώστε όταν δεν υπάρχει πρόσβαση στο διαδίκτυο, ο χρήστης να μπορεί να δει τα τελευταία δεδομένα που τον ενδιέφεραν και να μπορεί να περιηγηθεί σε αυτά ενώ είναι εκτός διαδικτύου.

Η βιβλιοθήκη Room παρέχει ένα επίπεδο αφαίρεσης με τη χρήση της SQLite για να επιτρέψει τη πλήρη πρόσβαση στη βάση δεδομένων. Συγκεκριμένα αυτή η βιβλιοθήκη παρέχει τα ακόλουθα οφέλη:

- Επαλήθευση των ερωτημάτων SQL την ώρα που γράφονται.
- Χρήση των annotations για ευκολία, ελαχιστοποίηση επαναλαμβανόμενου κώδικα και αποφυγή λαθών.
- Βελτιωμένες διαδρομές βάσης δεδομένων.

2.6. Transformations

Είναι πιθανόν να χρειαζόμαστε να κάνουμε κάποιες αλλαγές στα LiveData δεδομένα μας. Επίσης τυγχάνει να συγκρίνουμε με βάση κάποια δεδομένα ή ακόμη να θέλουμε να επιστρέψουμε κάποιον άλλο τύπο δεδομένων προτού αυτά γίνουν διαθέσιμα στους παρατηρητές(observers) μας. Όλα αυτά γίνονται με τη βοήθεια των Transformations, μια κλάση που περιέχει μεθόδους που εξυπηρετούν τέτοια σενάρια.

Δύο από τις συχνότερες μεθόδους χρήσης είναι:

- **Transformations.map()** όπου μπορούμε να τροποποιήσουμε τα δεδομένα ενός LiveData αντικειμένου και να διαθέσουμε τα αποτελέσματα.

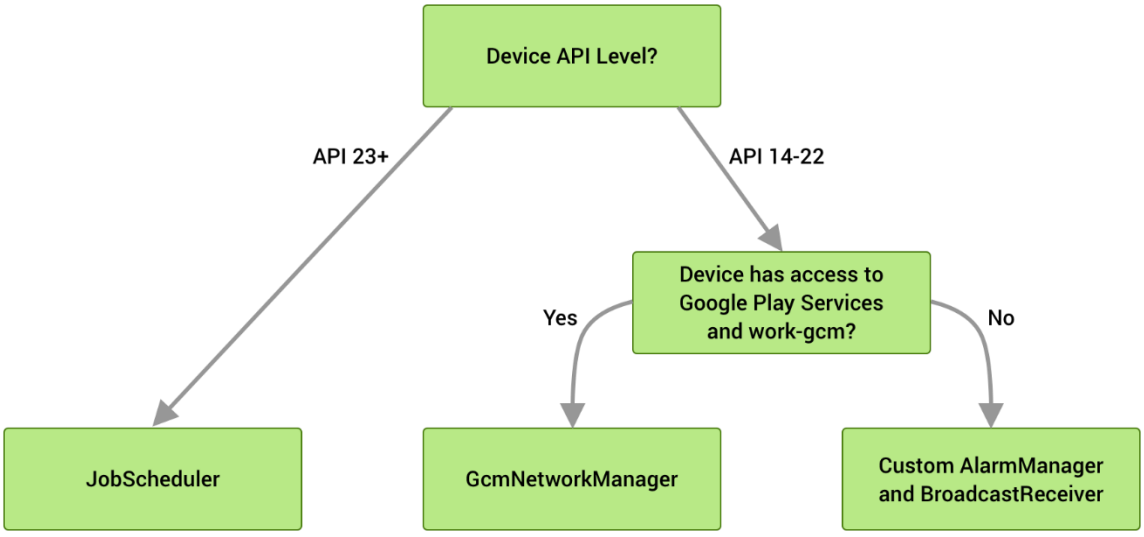
- **Transformations.switchmap()** όπου πάλι τροποποιούμε τα δεδομένα ενός LiveData αντικειμένου, μπορούμε να τα διασπάσουμε και να τα διαθέσουμε.

Τα Transformations μπορούν να χρησιμοποιηθούν για τη διάδοση πληροφοριών στο κύκλο ζωής του παρατηρητή. Δεν υπολογίζονται εκτός αν ένας παρατηρητής(observer) παρακολουθεί το LiveData αντικείμενο.

2.7. WorkManager

Το WorkManager είναι ένα API το οποίο χρησιμοποιείται για το προγραμματισμό ασύγχρονων εργασιών που θέλουμε να εκτελεστούν. Με το WorkManager ακόμη κι αν η εφαρμογή τερματιστεί ή η συσκευή επανεκκινηθεί οι προγραμματισμένες ασύγχρονες εργασίες μας θα εκτελεστούν. Είναι ο αντικαταστάτης όλων των προηγούμενων API εκτέλεσης προγραμματισμένων εργασιών στο παρασκήνιο όπως FirebaseJobDispatcher, GcmNetworkManager και Job Scheduler. Η συμβατότητά του ξεκινάει από το API 14 και μεταγενέστερα. Επίσης λαμβάνει υπόψιν και τη διάρκεια ζωής της μπαταρίας.

Με βάση κάποια κριτήρια το WorkManager χρησιμοποιεί μια υπηρεσία αποστολής εργασίας, όπως φαίνεται στη παρακάτω εικόνα:



Εικόνα 2 - WorkManager

Πέρα ότι είναι ένα εύχρηστο και απλό API προσφέρει κι άλλα οφέλη όπως:

- **Περιορισμούς εργασίας(Work Constraints):** Μπορούμε να δηλώσουμε κάποιους περιορισμούς για την εκτέλεση της εργασίας μας(π.χ. όταν το Wi-Fi της συσκευής είναι ανοιχτό).
- **Ισχυρό προγραμματισμό(Robust Scheduling):** Επιτρέπει την εκτέλεση της εργασίας για μία φορά ή επανειλημμένα. Η εργασία μπορεί να επισημανθεί(tag) και να ονομαστεί, με σκοπό να είναι μοναδική και αντικαταστάσιμη. Η προγραμματισμένη εργασία αποθηκεύεται εσωτερικά σε μια βάση δεδομένων SQLite για να διασφαλιστεί πως θα εκτελεστεί σε τυχόν επανεκκίνηση της συσκευής.
- **Ευέλικτη πολιτική επανάληψης(Flexible Retry Policy):** Μερικές φορές η εργασία αποτυγχάνει. Το WorkManager προσφέρει ευέλικτες πολιτικές επανάληψης.
- **Αλυσίδα εργασιών(Work Chaining):** Για πολλές και πολύπλοκες εργασίες προσφέρεται η επιλογή δημιουργίας μιας αλυσίδας σχετικών εργασιών. Επιλέξτε ποια κομμάτια εκτελούνται διαδοχικά και ποια παράλληλα.

Το WorkManager είναι κατάλληλο για εργασία με δυνατότητα αναβολής και για αξιόπιστη εκτέλεση ακόμη και αν η εφαρμογή τερματιστεί ή η συσκευή επανεκκινηθεί. Ένα χρήσιμο παράδειγμα είναι η αποστολή αρχείων καταγραφής σε υπηρεσίες.

Να σημειωθεί πως το WorkManager δεν προορίζεται για εργασία που τρέχει στο παρασκήνιο που μπορεί να τερματιστεί με ασφάλεια εάν η εφαρμογή τερματιστεί ή για εργασία που απαιτεί άμεση εκτέλεση.

2.8. RxJava

Η RxJava είναι μια εφαρμογή JVM(Java Virtual Machine) του ReactiveX. Το ReactiveX είναι μια βιβλιοθήκη που συνδυάζει τις ασύγχρονες ενέργειες και τα συμβάντα ενός προγράμματος με τη χρήση παρατηρήσιμων ακολουθιών.

Τα στοιχεία που αποτελούν την RxJava είναι τα παρατηρήσιμα (Observables) και οι συνδρομητές(Subscribers). Ένα παρατηρήσιμο στοιχείο εκπέμπει αντικείμενα και ο συνδρομητής χρησιμοποιείται για την κατανάλωση αυτών των αντικειμένων.

Μοιάζει πολύ με το τυπικό παρατηρήσιμο πρότυπο όπου θα αναφερθούμε αργότερα στην ενότητα(3.2.3 - LiveData) με περισσότερες επιλογές, σαν μια

καλύτερη αναβάθμιση. Όπως για παράδειγμα τα παρατηρήσιμα δεν εκπέμπουν αντικείμενα μέχρι κάποιος να εγγραφεί σε αυτά, πράγμα θετικό διότι δεν καταναλώνουμε πόρους τους οποίους δε θα αξιοποιήσουμε και ίσως μας χρειαστούν.

2.8.1. Χρήση της RxJava

Για να χρησιμοποιήσουμε την RxJava χρειαζόμαστε τα δομικά στοιχεία όπως αναφέραμε προηγουμένως, παρατηρήσιμα(Observables) και συνδρομητές (Subscribers). Το στοιχείο συνδρομητής κάνει εγγραφή στο παρατηρήσιμο στοιχείο. Το παρατηρήσιμο στοιχείο εκτελεί τη μέθοδο *onNext()* για όλα τα αντικείμενα. Αν όλα εκτελεστούν σωστά καλείται η μέθοδος *onComplete()*. Στην αντίθετη περίπτωση εκτελείται η μέθοδος *onError().*

Στη συνέχεια (εικόνα 3) απεικονίζεται ένα παράδειγμα στο οποίο δημιουργούμε ένα παρατηρήσιμο(Observable) αλφαριθμητικό(String) αντικείμενο, που περιέχει τη τιμή “Alfa Romeo”. Το επόμενο βήμα είναι να κάνουμε εγγραφή ένα νέο παρατηρητή(Observer). Αν όλα πάνε καλά, θα εκτελεστεί η *onNext()* και στο τέλος η *onComplete()* μέθοδος.

```
Observable<String> observable = Observable.just("Alfa Romeo");
observable.subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(@NonNull Disposable d) {
    }

    @Override
    public void onNext(@NonNull String s) {
        System.out.println(s);
    }

    @Override
    public void onError(@NonNull Throwable e) {
    }

    @Override
    public void onComplete() {
        System.out.println("Completed!");
    }
});
```

Εικόνα 3 - RxJava Observable example

Πράγματι παρακάτω(εικόνα 4) φαίνεται πως το παράδειγμα εκτελέστηκε και ολοκληρώθηκε χωρίς κανένα σφάλμα.

```
Alfa Romeo
Completed!

Process finished with exit code 0
```

Εικόνα 4 - RxJava Observable example result

2.8.2. Χειριστές της RxJava

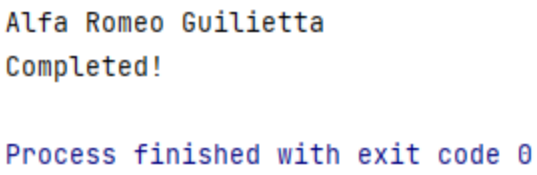
Η RxJava είναι διάσημη για την πληθώρα χειριστών που περιέχει. Οι χειριστές(Operators) είναι μέθοδοι που δημιουργήθηκαν για την επίλυση μετασχηματισμών(transformations) και τον χειρισμό προβλημάτων κλήσεων από API.

Για να γίνει πιο αντιληπτή η ιδέα ενός χειριστή(operator) θα κάνουμε ένα παράδειγμα. Στο παρακάτω παράδειγμα θα χρησιμοποιήσουμε το χειριστή Map που είναι ευρέως γνωστός. Αυτός ο χειριστής μπορεί να αλλάξει τα δεδομένα με βάση τη λογική που του δίνουμε.

```java
public static void main(String[] args) {
    Observable<String> observable = Observable.just("Alfa Romeo")
            .map(new Function<String, String>() {
                @Override
                public String apply(String s) throws Throwable {
                    return s + " Guilietta";
                }
            });
    observable.subscribe(new Observer<String>() {
        @Override
        public void onSubscribe(@NonNull Disposable d) {
        }

        @Override
        public void onNext(@NonNull String s) {
            System.out.println(s);
        }

        @Override
        public void onError(@NonNull Throwable e) {
        }

        @Override
        public void onComplete() {
            System.out.println("Completed!");
        }
    });
```

Εικόνα 5 - RxJava Map Operator

Αφού δημιουργήσουμε το αλφαριθμητικό παρατηρήσιμο αντικείμενο, χρησιμοποιούμε το χειριστή map με τον οποίο τροποποιούμε το αντικείμενο μας ανάλογα με το τι θέλουμε να κάνουμε. Σε αυτό το παράδειγμα απλά θα προσθέσουμε το αλφαριθμητικό “Guilietta” στο τέλος και θα το διαθέσουμε στον συνδρομητή(Subscriber) για εγγραφή.



Εικόνα 6 - RxJava Map Operator result

Το αποτέλεσμα (εικόνα 6) όπως ήταν αναμενόμενο με την προσθήκη του χειριστή Map χωρίς κάποιο σφάλμα.

Υπάρχουν πάρα πολλοί χειριστές και μπορούν να χρησιμοποιηθούν σε οποιοδήποτε αντικείμενο όπως αλφαριθμητικό(String), ακέραιο(Integer), δεκαδικό(decimal), λίστες(List) και πολλά άλλα. Είναι ένα πολύ ισχυρό εργαλείο για τη μετατροπή αντικειμένων και την τροποποίηση δεδομένων.

2.9. ViewPager2

Το ViewPager2 είναι μία βιβλιοθήκη μέρος του AndroidX με την οποία ο χρήστης μπορεί να περιηγηθεί από μία οθόνη στην άλλη με έναν πιο ενδιαφέρον και ελκυστικό τρόπο.

Αυτή η βιβλιοθήκη δεν είναι μέρος κάποιου αρχιτεκτονικού ή σχεδιαστικού προτύπου. Ο ρόλος της θα λέγαμε πως σχετίζεται κυρίως με τη διεπαφή χρήστη. Χρησιμοποιείται ώστε ο χρήστης να έχει μια διαφορετική εμπειρία περιήγησης από τη κλασική λειτουργία που προσφέρει το Android. Δημιουργεί μια πιο εντυπωσιακή και ευχάριστη εμπειρία για το χρήστη.

2.10. RecyclerViewSwipeDecorator

Επίσης μια βιβλιοθήκη όπου έχει να κάνει με το UI(διεπαφή χρήστη) και όχι με την αρχιτεκτονική ή σχεδιαστική μορφή της εφαρμογής.

Με αυτό το εργαλείο όταν ο χρήστης μετακινεί το στοιχείο μια ανακυκλώσιμης λίστας δεξιά ή αριστερά αλλάζει το φόντο της γραμμής του στοιχείου. Υπάρχει η επιλογή να προσθέσεις κάποιο εικονίδιο και ετικέτα σε κάθε στοιχείο της λίστας ανάλογα με τη κατεύθυνση που το ωθείς.

Η χρήση αυτής της βιβλιοθήκης μας βοηθάει να κάνουμε πιο αρεστή και εύκολη τη διαχείριση μιας λίστας δεδομένων στην εφαρμογή, δίνοντας στο χρήστη κάποιες επιλογές για το στοιχείο της λίστας.

2.11. Picasso

Το Picasso είναι ένα εύχρηστο εργαλείο για τη διαχείριση των εικόνων. Μας επιτρέπει να φορτώσουμε τις εικόνες στην εφαρμογή από διάφορες πηγές σχεδόν με μία γραμμή κώδικα και μας αποτρέπει από το να χειριστούμε τυχόν συνήθη προβλήματα όπως τη μετατροπή μιας εικόνας, τη προσωρινή αποθήκευση στη μνήμη και το χειρισμό μιας ανακυκλώσιμης λίστας(recyclerview) που περιέχει εικόνες.

3. ΑΝΑΛΥΣΗ ΚΑΙ ΣΧΕΔΙΑΣΗ ΕΦΑΡΜΟΓΗΣ

Για την ανάπτυξη μιας εφαρμογής παίζουν μεγάλο ρόλο οι απαιτήσεις του χρήστη και η σχεδιαστική δομή της εφαρμογής. Στο κεφάλαιο αυτό Θα διατυπωθούν οι απαιτήσεις που πρέπει να έχει η εφαρμογή καθώς και το αρχιτεκτονικό πρότυπο με το οποίο σχεδιάστηκε. Στη συνέχεια θα αναφερθούμε στη σχεδίαση σε επίπεδο κλάσεων ώστε να εκπληρώνει τις απαιτήσεις μας. Τέλος θα δούμε την υλοποίηση της βάσης δεδομένων για την εφαρμογή.

3.1. Ανάλυση απαιτήσεων

Ένα έργο, όπως μια εφαρμογή, έχει κάποιες απαιτήσεις. Απαίτηση είναι μια δυνατότητα όπου το σύστημα πρέπει να παρέχει. Οι απαιτήσεις αρχικά περιγράφονται από την επιχείρηση και στη συνέχεια προσθέτονται και οι τεχνικές απαιτήσεις(απαιτήσεις συστήματος).

Θα μπορούσε να ειπωθεί πως η ανάλυση απαιτήσεων είναι ένα συμβόλαιο ανάμεσα σε πελάτη και κατασκευαστή, που καταγράφεται ότι χρειάζεται το έργο με μια μορφή ξεκάθαρη για τους ενδιαφερόμενους.

Οι απαιτήσεις χωρίζονται σε δύο υποκατηγορίες:

- Λειτουργικές απαιτήσεις: όπου περιγράφουν τι πρέπει να κάνει το σύστημα, συνήθως από την επιχείρηση.
- Μη λειτουργικές απαιτήσεις: απαιτήσεις που ικανοποιούν κάποια ποιοτικά χαρακτηριστικά που πρέπει να έχει το έργο. Μερικά από αυτά τα χαρακτηριστικά είναι:
    - Αξιοπιστία: αν υπάρξει κάποιο πρόβλημα το σύστημα πρέπει να επανέλθει χωρίς να υπάρχουν απώλειες.

- Απόδοση: οι χρόνοι απόκρισης της εφαρμογής δε θα πρέπει να ξεπερνάνε κάποια όρια.
- Ασφάλεια: θα πρέπει τα προσωπικά δεδομένα και τα δεδομένα που ανακτώνται από κάποιον εξυπηρετητή να προστατεύονται.
- Ευχρηστία: αφορά την καλή επικοινωνία της εφαρμογής με το χρήστη για την εύκολη πλοήγηση και αξιοποίηση των δυνατοτήτων της εφαρμογής.
- Διαθεσιμότητα: η εφαρμογή να είναι πάντα διαθέσιμη, χωρίς περιορισμούς.

3.1.1. Περιπτώσεις Χρήσης

Το μοντέλο περιπτώσεων χρήσης είναι ένας μηχανισμός καταγραφής των απαιτήσεων σε μορφή κειμένου που μπορούν να απεικονισθούν και με διαγράμματα. Είναι κείμενο που περιγράφει τη χρήση του συστήματος ώστε να επιτευχθεί ο στόχος του χρήστη. Οι περιπτώσεις χρήσης είναι κυρίως λειτουργικές απαιτήσεις.

Μία περίπτωση χρήσης(use case) περιλαμβάνει διάφορα σενάρια επιτυχίας και αποτυχίας, που περιγράφουν ένα χρήστη που χρησιμοποιεί το σύστημα για την διεκπεραίωση μιας λειτουργίας.

Το σενάριο (scenario) αποτελείται από ενέργειες και αλληλεπιδράσεις ανάμεσα στο χειριστή και το σύστημα.

Υπάρχουν τρεις μορφές περιπτώσεων χρήσεων:

- Συνοπτικές: περίληψη σε μία παράγραφο το σενάριο επιτυχίας (happy path).
- Απλές: πολλές παράγραφοι που καλύπτουν διάφορα σενάρια.
- Πλήρεις: όλα τα βήματα και τις παραλλαγές που υπάρχουν.

Σε αυτή τη διπλωματική εργασία θα αναφερθούμε μόνο στη συνοπτική μορφή. Στη συνέχεια αναγράφονται μερικές περιπτώσεις χρήσης για την εφαρμογή.

- Δημιουργία νέας συνταγής: ο χρήστης ανοίγει την εφαρμογή. Επιλέγει να δημιουργήσει μία νέα συνταγή. Συμπληρώνει τα απαραίτητα στοιχεία όπως το όνομα της συνταγής, τη κατηγορία όπου ανήκει, τα υλικά που την αποτελούν και τα βήματα εκτέλεσης. Η εφαρμογή αποθηκεύει τη νέα συνταγή και ενημερώνει το χρήστη πως επιτεύχθηκε η καταχώριση της.
- Αναζήτηση μιας συνταγής: ο χρήστης ανοίγει την εφαρμογή. Από την αρχική οθόνη επιλέγει την αναζήτηση. Πληκτρολογεί το όνομα της

συνταγής που αναζητεί. Η εφαρμογή ανταποκρίνεται με μια λίστα από τις πιθανές συνταγές που αναζητεί ο χρήστης. Στη συνέχεια ο χρήστης διαλέγει την συνταγή που τον ενδιαφέρει.

- Εκτέλεση της συνταγής: ο χρήστης ανοίγει την εφαρμογή. Στην αρχική οθόνη εμφανίζονται όλες οι συνταγές. Ο χρήστης επιλέγει την συνταγή που τον ενδιαφέρει. Η εφαρμογή εμφανίζει λεπτομέρειες της συνταγής καθώς και την επιλογή για την εκτέλεση των βημάτων της.

Σκοπός των περιπτώσεων χρήσεων είναι η σύλληψη των στόχων των χρηστών. Μας βοηθάνε να κατανοήσουμε τι πραγματικά θέλει ο χρήστης να πετύχει μέσα από τις απόψεις του.

3.2. Αρχιτεκτονική σχεδίαση – Το μοντέλο MVVM

Στην αρχιτεκτονική σχεδίαση σημαντικό ρόλο παίζουν οι μη λειτουργικές απαιτήσεις. Είναι οι απαιτήσεις που ικανοποιούν κάποια ποιοτικά χαρακτηριστικά που πρέπει να έχει μια εφαρμογή όπως αξιοπιστία, απόδοση, διαθεσιμότητα στα οποία έγινε αναφορά προηγουμένως.

Επίσης δε θα μπορούσαμε να παραλείψουμε ένα πολύ σημαντικό κανόνα το διαχωρισμό ανησυχιών(separation of concerns). Ένα συνήθης λάθος φαινόμενο είναι να γράφουμε όλο το κώδικα σε μια κλάση δημιουργώντας μια ‘God Class’. Πιο συγκεκριμένα, αφού μιλάμε για Android εφαρμογές, οι κλάσεις όπως Δραστηριότητες(Activities) και Αποσπάσματα(Fragments) θα πρέπει να περιέχουν κώδικα που αφορά μόνο τις αλληλεπιδράσεις διεπαφής χρήστη και λειτουργικού συστήματος.

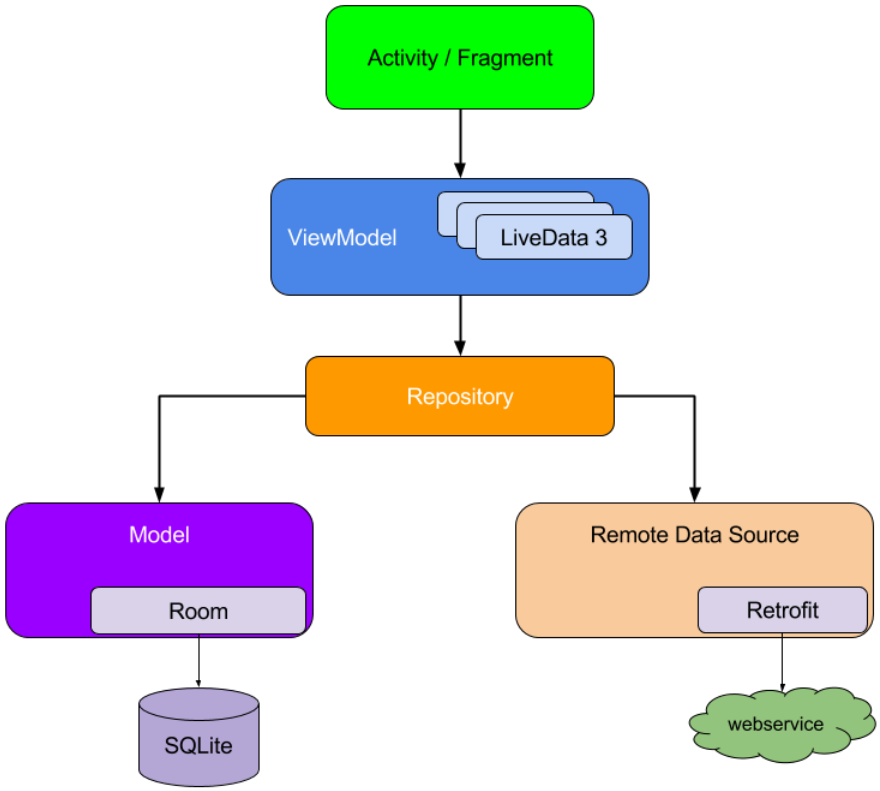
Μια άλλη σημαντική αρχή είναι ότι το περιβάλλον χρήστη πρέπει να οδηγείτε από ένα μοντέλο. Τα μοντέλα(models) είναι συνήθως κλάσεις με τα οποία μπορούμε να διαχειριστούμε δεδομένα στην εφαρμογή μας. Προτιμότερο είναι η χρήση επίμονων μοντέλων(persistent model).

Με τη χρήση επίμονων μοντέλων:

- Οι χρήστες δεν χάνουν δεδομένα εάν το Android OS καταστρέψει την εφαρμογή για την απελευθέρωση πόρων.
- Η εφαρμογή συνεχίζει να λειτουργεί σε περίπτωση όπου χαθεί η σύνδεση στο δίκτυο.

Με βάση τα προαναφερόμενα και την επίσημη ομάδα ανάπτυξης Android, το προτεινόμενο αρχιτεκτονικό μοντέλο ανάπτυξης εφαρμογών για φορητές συσκευές που υποστηρίζουν Android είναι το MVVM(Model-View-ViewModel).

Παρακάτω απεικονίζεται πως τα δομικά στοιχεία που αποτελούν το αρχιτεκτονικό μοντέλο MVVM αλληλοεπιδρούν μεταξύ τους.



Εικόνα 7 – Αλληλεπίδραση δομικών στοιχείων του μοντέλου MVVM

Κάθε δομικό στοιχείο εξαρτάται μόνο από ένα άλλο στοιχείο κάτω από αυτό. Μόνο η αποθήκη(repository) εξαρτάται από ένα επίμονο μοντέλο(persistent model) και από μία απομακρυσμένη πηγή δεδομένων(remote data source).

Με βάση αυτό το αρχιτεκτονικό μοντέλο, κάθε φορά που γυρίζει ο χρήστης στην εφαρμογή, είτε μετά από λίγα λεπτά είτε μετά από μέρες, πάντα θα εμφανίζονται οι πληροφορίες που υπάρχουν στη τοπική βάση δεδομένων. Εάν οι πληροφορίες είναι παλιές τότε η αποθήκη(repository) ενημερώνει τις πληροφορίες.

3.2.1. Η κλάση ViewModel

Η κλάση ViewModel μας επιτρέπει να αποθηκεύονται και να διαχειρίζονται πληροφορίες που σχετίζονται με τη διεπαφή χρήστη(User Interface) με τέτοιο τρόπο ώστε να γίνεται αντιληπτός από τον κύκλο ζωής(lifecycle) της συσκευής μας.

Το Android διαχειρίζεται τον κύκλο ζωής των ελεγκτών διεπαφής(UI controllers) Δραστηριοτήτων(Activities) και Αποσπασμάτων(Fragments). Ανάλογα με τον τρόπο που διαχειρίζεται ο χρήστης μια εφαρμογή, τις επιλογές που κάνει ή κάποια συμβάντα συσκευής, το Android αποφασίζει αν θα καταστραφεί ή αν θα αναδημιουργηθεί ο ελεγκτής διεπαφής.

Αυτό μπορεί να μας επιφέρει διάφορα προβλήματα στην εφαρμογή μας, αν δε βρούμε λύση να τα αντιμετωπίσουμε, όπως:

- Αν το σύστημα καταστρέψει ή αναδημιουργήσει τον ελεγκτή διεπαφής, τότε ότι πληροφορίες έχουμε αποθηκεύσει σε αυτόν θα χαθούν. Για παράδειγμα η εφαρμογή μας περιλαμβάνει μια λίστα με υλικά. Αν ο χρήστης προσθέσει ένα νέο υλικό και στη συνέχεια γυρίσει τη συσκευή σε οριζόντια θέαση(landscape), το σύστημα πρέπει να αναδημιουργήσει τη Δραστηριότητα από την αρχή, χάνοντας τη πληροφορία που είναι το νέο υλικό που πρόσθεσε ο χρήστης.
- Πολλές φορές ο ελεγκτής διεπαφής χρειάζεται να κάνει ασύγχρονες ενέργειες οι οποίες παίρνουν κάποιο χρόνο μέχρι να ολοκληρωθούν. Αυτές οι ενέργειες διαχειρίζονται από τον ελεγκτή διεπαφής και πρέπει να διασφαλίσει πως το σύστημα θα τις καθαρίσει αφού καταστραφεί για να αποτρέψει πιθανές διαρροές από τη μνήμη. Αυτή η διαδικασία απαιτεί πολλή συντήρηση και σε περίπτωση κάποιας αλλαγής διαμόρφωσης είναι χάσιμο πόρων αφού ο ελεγκτής θα πρέπει να ξανά κάνει αυτές τις ασύγχρονες ενέργειες τις οποίες είχε εκτελέσει ήδη.

Οι ελεγκτές διεπαφής(Activities, Fragments) είναι υπεύθυνοι ώστε να

- εμφανίζουν τις πληροφορίες στην οθόνη.
- ανταποκρίνονται στις ενέργειες του χρήστη.
- διαχειρίζονται την επικοινωνία του λειτουργικού συστήματος, όπως αιτήματα άδειας.

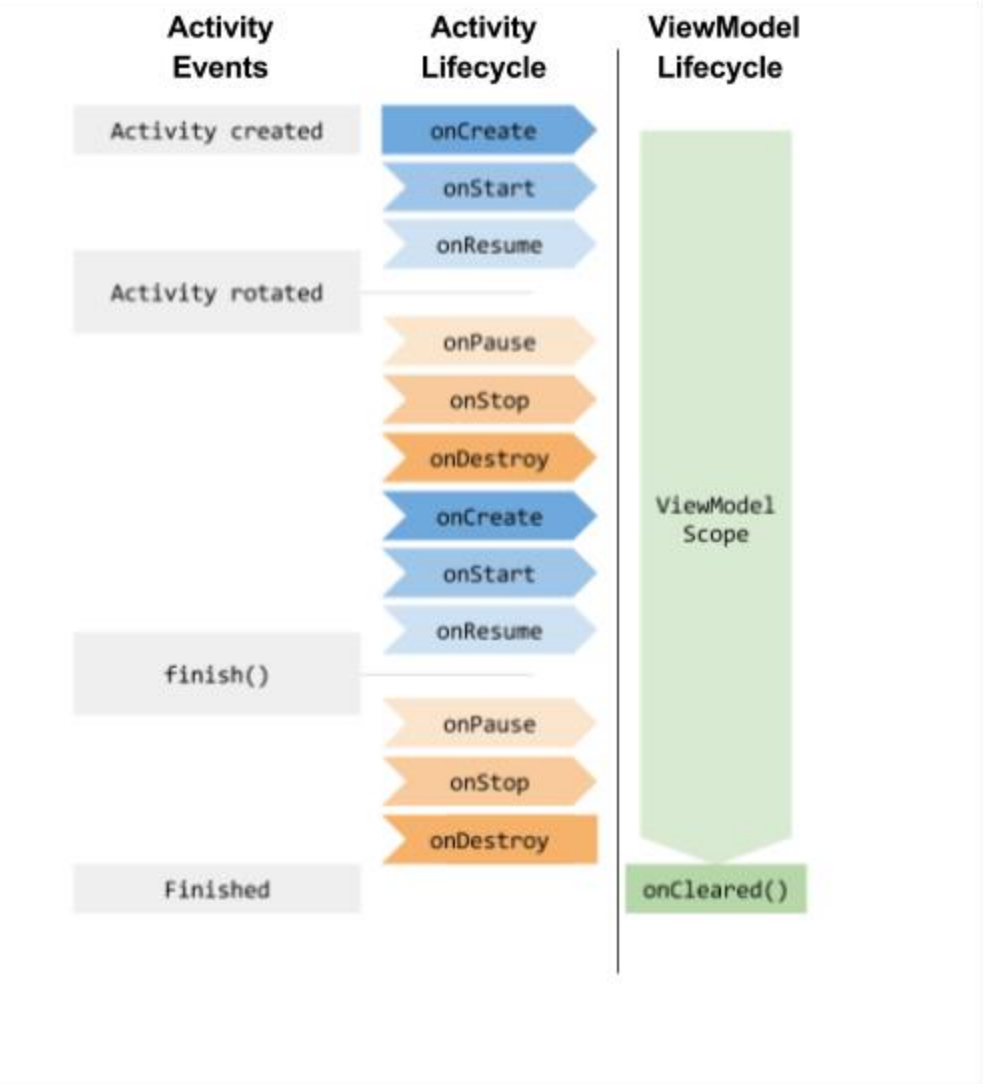
Η ανάθεση ενός ελεγκτή διεπαφής όπου μαζεύει τις πληροφορίες από μια βάση ή από το διαδίκτυο επιφέρει μεγαλύτερο φορτίο στον ελεγκτή. Έτσι θα είχαμε μια τεράστια κλάση(god class) που θα ήταν υπεύθυνη για να κάνει πολλές δουλειές από μόνη της, αντί να αναθέτει δουλειά σε άλλες κλάσεις. Επιπλέον ο

κώδικας μας δε θα ήταν ευανάγνωστος, δύσκολος στον εντοπισμό σφαλμάτων και testing.

Γι’ αυτό το λόγο είναι προτιμότερο να διαχωρίζουμε τα δεδομένα που πρέπει να προβάλλουμε από τη λογική του ελεγκτή, με τη βοήθεια της κλάσης ViewModel.

3.2.2. Ο κύκλος ζωής της κλάσης ViewModel

Τα αντικείμενα που βρίσκονται στη κλάση ViewModel ακολουθούν τον κύκλο ζωής του ViewModelProvider, μια κλάση βοηθητικών εργαλείων. Η κλάση ViewModel παραμένει στη μνήμη μέχρι να ολοκληρωθεί ο κύκλος ζωής της. Σε μια δραστηριότητα όταν τελειώσει, ενώ σε ένα απόσπασμα όταν αποκολληθεί. Στην παρακάτω εικόνα φαίνεται ο κύκλος ζωής μιας δραστηριότητας(Activity) σε σχέση με τη διάρκεια ζωής μιας κλάσης ViewModel, όταν ο χρήστης κάνει περιστροφή της συσκευής.



Εικόνα 8 – Κύκλος ζωής της ViewModel

Παρόμοια συμβαίνει και στο κύκλο ζωής ενός αποσπάσματος (Fragment). Στη παραπάνω εικόνα παρατηρούμε πως η κλάση ViewModel καλείται όταν δημιουργείται η Δραστηριότητα, στη μέθοδο onCreate(). Κάθε φορά που η οθόνη της συσκευής περιστρέφεται η Δραστηριότητα αναδημιουργείται, εκτελείται η onCreate() και καλείται η κλάση ViewModel που είχε δημιουργηθεί για πρώτη φορά. Το αποτέλεσμα είναι οι πληροφορίες μας να είναι πάντα ενημερωμένες και έγκυρες.

3.2.3. Η κλάση LiveData

Η LiveData είναι μια παρατηρήσιμη κλάση η οποία περιέχει δεδομένα. Πιο απλά, στοιχεία όπως οι ελεγκτές διεπαφής(Activites και Fragments) μπορούν να

παρατηρούν τα δεδομένα της κλάσης LiveData. Σε σχέση με άλλες παρόμοιες κλάσεις, αυτήν έχει επίγνωση του κύκλου ζωής άλλων στοιχείων της εφαρμογής, όπως είναι οι δραστηριότητες και αποσπάσματα. Χάρις αυτή την επίγνωση ενημερώνονται μόνο τα στοιχεία που είναι ενεργά στο κύκλο ζωής. Ενεργά(Active) εννοούμε τα στοιχεία που βρίσκονται σε καταστάση εκκίνησης(STARTED) ή συνέχειας(RESUME). Τα στοιχεία που έχει τελειώσει ο κύκλος ζωής(Inactive) τους δεν ενημερώνονται, καταστρέφονται(DESTROYED), πράγμα που μας αποτρέπει από διάφορες διαρροές.

Πλεονεκτήματα από τη χρήση της LiveData:

- Διασφαλίζει ότι τα δεδομένα που προβάλλονται στη διεπαφή χρήστη(UI) ταιριάζουν με τη κατάσταση δεδομένων. Η κλάση LiveData παρατηρεί τα δεδομένα αν αλλάξουν και ενημερώνει τα αντικείμενα-παρατηρητές που ενδιαφέροντα για αυτά. Με αυτό τον τρόπο δε χρειάζεται να ενημερώνουμε χειροκίνητα τυχόν αλλαγές που πραγματοποιούνται κάθε φορά.
- Δεν υπάρχουν διαρροές μνήμης. Οι παρατηρητές είναι συνδεδεμένοι με το κύκλο ζωής των αντικειμένων, έτσι καταστρέφονται όταν καταστραφεί ο κύκλος ζωής του αντικειμένου που εξαρτώνται.
- Η εφαρμογή δε τερματίζει χωρίς λόγο επειδή κάποια δραστηριότητα σταμάτησε. Όταν μια δραστηριότητα είναι σε Inactive κατάσταση τότε δε λαμβάνει συμβάντα από τη κλάση LiveData.
- Δε χρειάζεται χειροκίνητος χειρισμός του κύκλου ζωής. Αυτόματη διαχείριση καθώς υπάρχει επίγνωση του κύκλου ζωής.
- Πάντα ενημερωμένα δεδομένα. Όταν κάποιο επανέρχεται σε ενεργή κατάσταση λαμβάνει τα τελευταία δεδομένα.
- Σωστές αλλαγές διαμόρφωσης. Όταν ένας ελεγκτής διεπαφής(Activity/Fragment) αναδημιουργείται(π.χ. περιστροφή συσκευής), ενημερώνεται με τα τελευταία δεδομένα.
- Κοινή χρήση πόρων. Μπορούμε να χρησιμοποιήσουμε τη LiveData σε ένα αντικείμενο και πολλές δραστηριότητες ή αποσπάσματα να παρατηρούν την ίδια αναφορά.

3.2.4. Αποθήκη δεδομένων – Repository

Σε μια εφαρμογή συνήθως συλλέγουμε δεδομένα από πολλές και διάφορες πηγές, είτε είναι απομακρυσμένες είτε τοπικές. Προκειμένου να συγκεντρώσουμε τα δεδομένα και να αποκτήσουμε πρόσβαση, χρειαζόμαστε μία ενιαία πηγή αλήθειας(Single Source Of Truth).

Η αποθήκη δεδομένων είναι υπεύθυνη για την ανάκτηση των δεδομένων από τις πηγές δεδομένων αλλά και για το πως θα αποθηκεύονται τα διατηρήσιμα δεδομένα μας. Οι συνηθέστερες πηγές δεδομένων είναι μια τοπική βάση δεδομένων, μια απομακρυσμένη βάση, ένα αρχείο xml, ένα αρχείο json είτε κάποια άλλη πηγή.

Το πρότυπο αποθήκης δεδομένων(Repository) είναι ένα ευρέως γνωστό πρότυπο που χρησιμοποιείται για να μας δώσει λύσεις σε σχεδιαστικά προβλήματα που ερχόμαστε αντιμέτωποι. Το χρησιμοποιούμε για να διαχωρίσουμε τη λογική που ανακτά τα δεδομένα και τα αντιστοιχίζει στο μοντέλο οντότητας από την επιχειρησιακή λογική που λειτουργεί στο μοντέλο.

3.3. Σχεδίαση αντικειμένων

Κατά την ανάπτυξη δημιουργούμε διάφορα διαγράμματα, με τη βοήθεια εργαλείων, για τα δύσκολα σημεία του έργου. Τα διαγράμματα αυτά ενδέχεται να τροποποιηθούν όσο το έργο εξελίσσεται και γίνεται πιο κατανοητό.

Σε δύο κύριες κατηγορίες χωρίζεται η μοντελοποίηση:

1. Στατική και,
2. Δυναμική

Στη στατική μοντελοποίηση ανήκουν τα διαγράμματα όπου διακρίνονται πακέτα, κλάσεις, ιδιότητες και μέθοδοι. Σε αυτή τη κατηγορία εντάσσονται τα διαγράμματα:

- Κλάσεων (Class Diagram)
- Πακέτων (Package Diagram)
- Ανάπτυξης (Deployment Diagram)

Αντίθετα στη δυναμική μοντελοποίηση ανήκουν τα διαγράμματα που βοηθούν στη σχεδίαση της λογικής και συμπεριφοράς του κώδικα στο εσωτερικό των μεθόδων. Τα διαγράμματα αυτά είναι:

- Ακολουθίας (Sequence Diagram)
- Επικοινωνίας (Communication Diagram)
- Μηχανής καταστάσεων(State Machine Diagram)
- Δραστηριότητας (Activity Diagram)

3.3.1. Διαγράμματα κλάσεων

Όπως αναφέρθηκε στην πρώτη ενότητα, η εφαρμογή έχει αναπτυχθεί με βάση τη γλώσσα προγραμματισμού Java που είναι μια αντικειμενοστραφή γλώσσα.

Μια αντικειμενοστραφή γλώσσα χρησιμοποιεί αντικείμενα τα οποία προκύπτουν από κλάσεις. Η κλάση είναι μια δομή δεδομένων που αποτελείται από ιδιότητες και λειτουργίες που τη περιγράφουν.

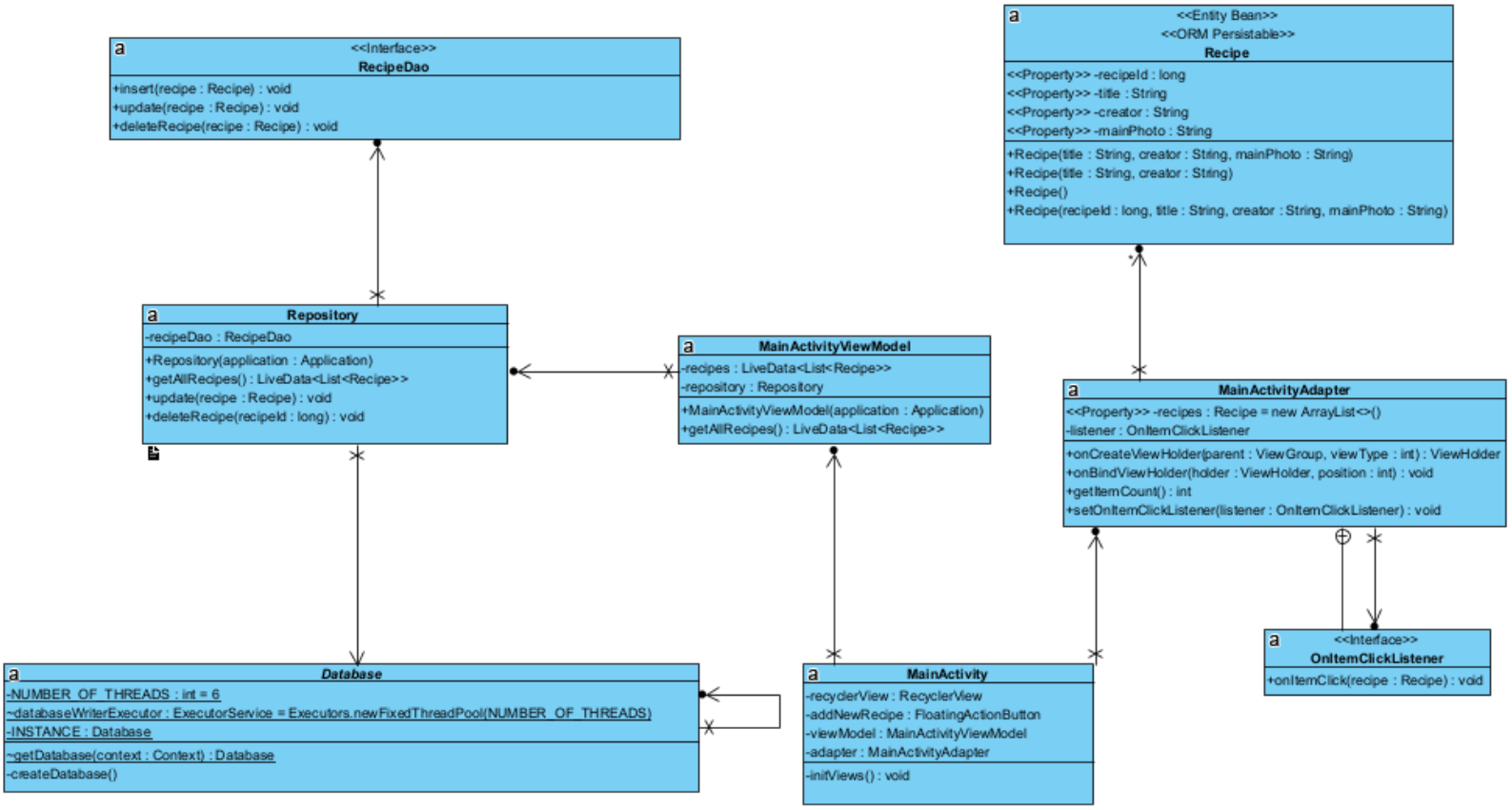
Το διάγραμμα των κλάσεων είναι ένα από τα βασικότερα διαγράμματα της UML. Δείχνει τα χαρακτηριστικά από τα οποία αποτελούνται οι κλάσεις του συστήματος και τις σχέσεις μεταξύ τους.

Σε ένα διάγραμμα κλάσεων συνήθως θα βρούμε τα εξής στοιχεία:

- Κλάσεις (Classes)
- Διεπαφές(Interfaces)
- Συνεργασίες(Collaborations)
- Συσχετίσεις(Relationships)

Χρησιμοποιείται κατά την διάρκεια της ανάλυσης για να περιγράψει τις λειτουργικές απαιτήσεις. Επίσης μεγάλο ρόλο παίζει στη φάση του σχεδιασμού ώστε να περιγράψει το λεξιλόγιο του συστήματος, τις συνεργασίες και το λογικό σχήμα της βάσης δεδομένων.

Το παρακάτω διάγραμμα κλάσης απευθύνεται σε μια δραστηριότητα της εφαρμογής.



Εικόνα 9 - Διάγραμμα κλάσεων της MainActivity

Στο διάγραμμα διακρίνονται οι κλάσεις που χρησιμοποιήθηκαν: Repository, MainActivityViewModel, MainActivity, MainActivityAdapter, Recipe, Database. Επίσης περιέχει τα RecipeDao και OnItemClickListener Interfaces. Το διάγραμμα απεικονίζει πως όλα αυτά συσχετίζονται μεταξύ τους.

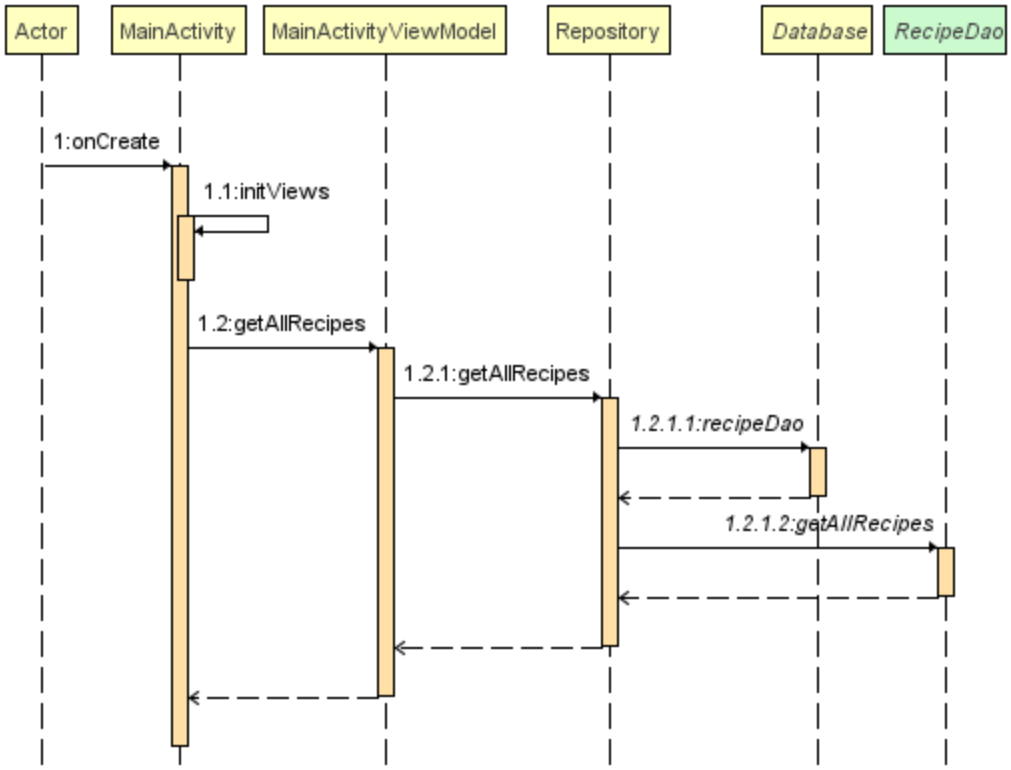
3.3.2. Διαγράμματα ακολουθίας

Σε ένα διάγραμμα ακολουθίας(Sequence Diagram) παρατηρούμε πως αλληλοεπιδρούν τα αντικείμενα μεταξύ τους μέσω της ανταλλαγής μηνυμάτων. Κύριος στόχος τους είναι να γίνει αντιληπτός ο τρόπος με τον οποίο εξελίσσονται τα γεγονότα μέσα σε μια χρονική περίοδο.

Σε ένα διάγραμμα ακολουθίας θα εντοπίσουμε τα εξής:

- Χειριστές
- Αντικείμενα
- Μηνύματα

Στον οριζόντιο άξονα του διαγράμματος εντοπίζονται τα αντικείμενα που λαμβάνουν μέρος, ενώ στον κάθετο άξονα τα γεγονότα που εμπλέκονται σε αυτή τη χρονική περίοδο.



Εικόνα 10 – Διάγραμμα ακολουθίας

Ένα απλό παράδειγμα στο οποίο απεικονίζεται η ανταλλαγή μηνυμάτων στην κύρια οθόνη της εφαρμογής.

Ο χρήστης ανοίγει την εφαρμογή, αμέσως καλείται η μέθοδος onCreate της MainActivity και η initViews όπου αρχικοποιεί τη διεπαφή χρήστη. Στη συνέχεια εκτελείται ασύγχρονα η μέθοδος getAllRecipes της κλάσης MainActivityViewModel όπου θα φέρει τα δεδομένα που χρειαζόμαστε για να προβληθούν στο χρήστη. Η MainActivityViewModel με τη σειρά της χρησιμοποιεί την αποθήκη δεδομένων, που χρησιμεύει σα κόμβος που αντλεί τα δεδομένα από διάφορες πηγές. Η αποθήκη δεδομένων επικοινωνεί με τη τοπική βάση δεδομένων της εφαρμογής και επιστρέφει μια αναφορά της. Με τη βοήθεια του Interface RecipeDao λαμβάνουμε τα δεδομένα στη κατάλληλη μορφή όπου και καταλήγουν στη διεπαφή χρήστη.

Το παραπάνω διάγραμμα δημιουργήθηκε με σκοπό να γίνει καλύτερα αντιληπτή η ιδέα πως δουλεύει το αρχιτεκτονικό μοντέλο MVVM μέσα σε μια εφαρμογή και ο τρόπος με τον οποίο εξελίσσονται τα γεγονότα.

3.3.3. Διαγράμματα Οντοτήτων – Συσχετίσεων

Το διάγραμμα οντοτήτων – συσχετίσεων είναι ένα αφαιρετικό μοντέλο δεδομένων, όπου μας περιγράφει τη δομή για τη σχεδίαση μιας βάσης δεδομένων.

Χρησιμοποιείται στο αρχικό στάδιο σχεδίασης ενός συστήματος και σκοπός του είναι να μας περιγράψει τις αναγκαίες πληροφορίες που πρέπει να αποθηκευτούν σε μία βάση δεδομένων.

Σε ένα τέτοιο μοντέλο θα παρατηρήσουμε τα εξής κύρια χαρακτηριστικά:

- Οντότητα(entity): συνήθως είναι ένα αντικείμενο ή πράγμα το οποίο έχει ιδιαίτερη σημασία στο πρόβλημα(π.χ. φοιτητής, βιβλίο, συνταγή).
- Χαρακτηριστικά(attributes): κάθε οντότητα περιέχει διάφορα στοιχεία-χαρακτηριστικά που την περιγράφουν.
- Συσχέτιση(relationship): ονομάζεται η σύνδεση ανάμεσα σε δύο ή περισσότερες οντότητες μεταξύ τους.
- Πολυπλοκότητα μιας συσχέτισης: πολυπλοκότητα ή βαθμός μιας συσχέτισης είναι ο αριθμός των οντοτήτων που παίρνουν μέρος σε μια συσχέτιση. Ο συνηθέστερος βαθμός συσχέτισης είναι το 2.
- Πληθικότητα(cardinality): περιγράφει τον αριθμό των στιγμιοτύπων μιας οντότητας που μπορεί να αντιστοιχίζονται με τον αριθμό των στιγμιοτύπων μιας άλλης οντότητας.

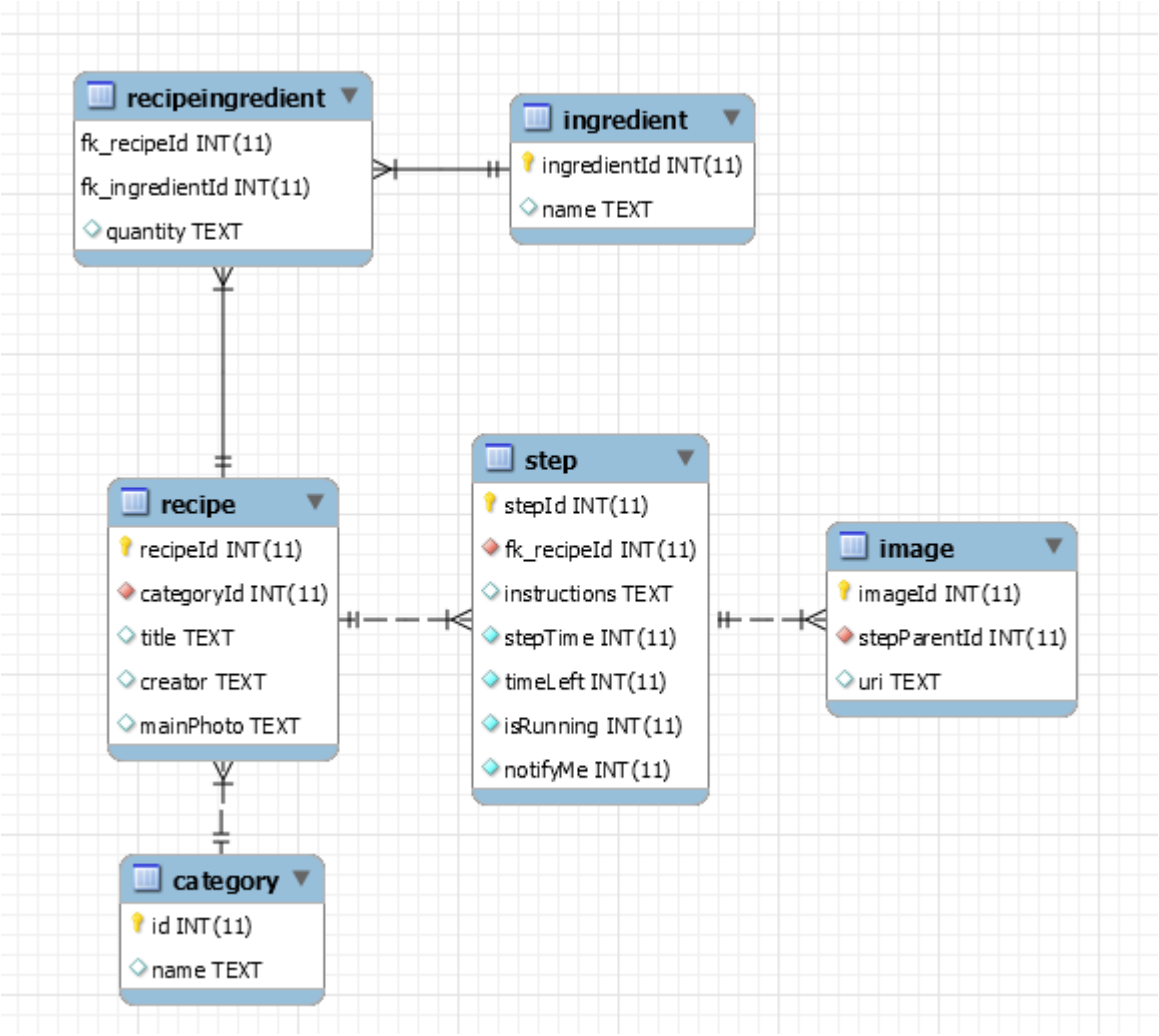
Εκτός από τα παραπάνω κύρια χαρακτηριστικά που συναντάμε σε ένα μοντέλο οντοτήτων-συσχετίσεων, υπάρχουν κι άλλα όπως:

- Υποκλάσεις
- Υπερκλάσεις
- Κληρονομικότητα
- Ασθενής τύπος οντότητας
- Και άλλα

Στο διάγραμμα της εικόνας 10 είναι ένα στιγμιότυπο της σχεσιακής βάσης δεδομένων της εφαρμογής. Το διάγραμμα παρήχθη με τη βοήθεια του εργαλείου MySQL Workbench και της λειτουργίας reverse-engineer που παρέχει.

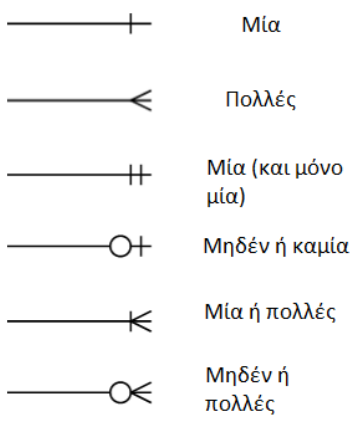
Τα κουτάκια είναι οι λεγόμενες οντότητες(entities) της βάσης δεδομένων. Η κάθε οντότητα περιγράφεται από χαρακτηριστικά. Ανάμεσα στα χαρακτηριστικά διακρίνονται τα στοιχεία που έχουν καθοριστεί ως πρωτεύον(κίτιρινο) και ξένο(κόκκινο) κλειδί αντίστοιχα όπου υπάρχουν. Επίσης μια οντότητα συσχετίζεται με μία άλλη ή και περισσότερες (οντότητες) αναδεικνύοντας το

βαθμό πολυπλοκότητας. Τέλος σε κάθε άκρη μιας συσχέτισης εκφράζεται η πληθικότητα(cardinality).



Εικόνα 11 – Διάγραμμα οντοτήτων-συσχετίσεων της βάσης δεδομένων

Οι διακεκομμένες γραμμές στις συσχετίσεις φανερώνουν μία μη ταυτοποίηση(non identifying) ανάμεσα στις οντότητες. Χρησιμοποιείται μόνο για οπτικούς λόγους καθώς δεν αλλάζει κάτι στο σύνολο της βάσης, ούτε ανάμεσα στις συσχετίσεις. Αυτός ο όρος μας υποδεικνύει ότι το πρωτεύον κλειδί του γονέα δε χρησιμοποιείται ως πρωτεύον κλειδί στην οντότητα παιδί.



Εικόνα 12 - Πίνακας συμβόλων πληθικότητας(cardinality)

3.4. Αποθήκευση διατηρήσιμων δεδομένων με τη χρήση Room

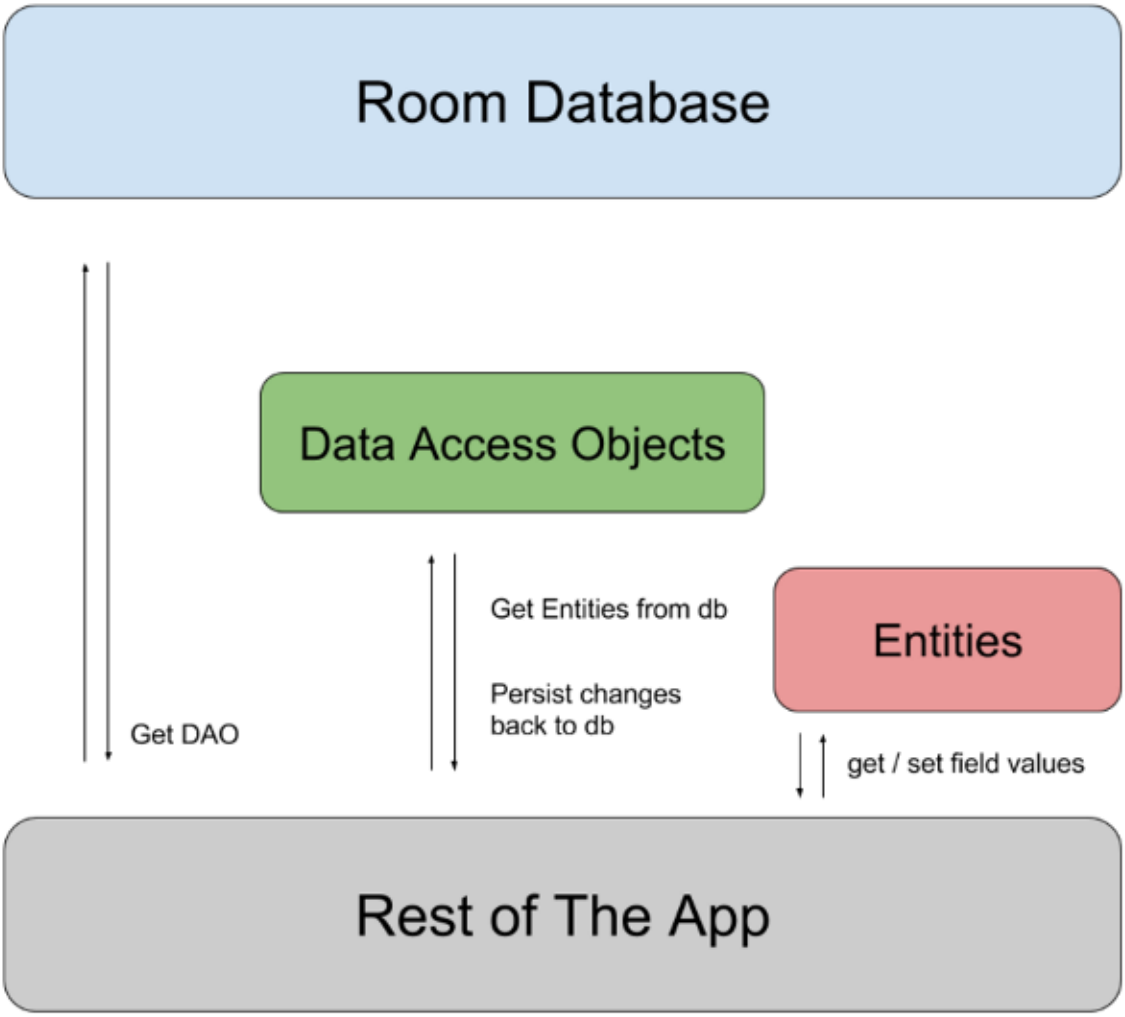
Οι εφαρμογές συνήθως χρησιμοποιούν δομημένα δεδομένα για να επωφεληθούν από αυτά. Συνήθως τα δεδομένα αυτά ανακτώνται από διαδικτυακούς εξυπηρετητές ή μια τοπική βάση. Αν ο χρήστης δεν έχει πρόσβαση στο διαδίκτυο, η συσκευή χρησιμοποιεί τη τοπική βάση ώστε να προβάλλει κάποια δεδομένα.

3.4.1. Κύρια συστατικά

Η βιβλιοθήκη Room αποτελείται από τρία κύρια συστατικά:

- Η κλάση βάσης δεδομένων που διατηρεί τη βάση δεδομένων και χρησιμεύει για την σύνδεση με τα δεδομένα μας.
- Οντότητες(entities) δεδομένων που αντιπροσωπεύουν πίνακες στη βάση δεδομένων.
- Αντικείμενα πρόσβασης δεδομένων(Data Access Objects-DAOs) που παρέχουν μεθόδους τις οποίες χρειάζεται η εφαρμογή μας(π.χ. εισαγωγή, ενημέρωση, διαγραφή).

Η κλάση βάσης δεδομένων παρέχει στην εφαρμογή περιπτώσεις DAOs που σχετίζονται με την βάση δεδομένων. Στη συνέχεια, η εφαρμογή μπορεί να χρησιμοποιήσει τα DAOs για να ανακτήσει δεδομένα από τη βάση δεδομένων στη μορφή που σχετίζονται τα αντικείμενα οντότητας δεδομένων.



Εικόνα 13 – Δομικά στοιχεία του Room Api

Προηγουμένως απεικονίζεται η σχέση μεταξύ των συστατικών της βιβλιοθήκης Room.

3.4.2. Ασύγχρονα ερωτήματα

Η βιβλιοθήκη Room δεν επιτρέπει την πρόσβαση της βάσης δεδομένων στο κύριο νήμα(main thread), διότι τα ερωτήματα μπορεί να μπλοκάρουν το UI(User Interface). Γι’ αυτό το λόγο τα ερωτήματα μας στη DAO κλάση πρέπει να είναι ασύγχρονα.

Υπάρχουν τρεις κατηγορίες ασύγχρονων ερωτημάτων:

- Εγγραφής μιας λήψης (one-shot write) με τα οποία μπορούμε να κάνουμε εισαγωγή, διαγραφή και ενημέρωση της βάσης.
- Ανάγνωσης μια λήψης(one-shot read), τα οποία διαβάζουν μια φορά τη βάση δεδομένων και επιστρέφουν ένα στιγμιότυπο εκείνης της στιγμής.
- Παρατηρήσιμα(observables), που διαβάζουν από τη βάση δεδομένων κάθε φορά που γίνεται μια αλλαγή στα δεδομένα των πινάκων.

Η Room μπορεί να συνδυαστεί με συγκεκριμένες γλώσσες, βιβλιοθήκες και πλαίσια(frameworks). Ο παρακάτω πίνακας δείχνει τους τύπους επιστροφής με βάση τα ερωτήματα και το πλαίσιο(framework) που θέλουμε να χρησιμοποιήσουμε.

Query type	Kotlin language features	RxJava	Guava	Jetpack Lifecycle
One-shot write	Coroutines (`suspend`)	`Single<T>`, `Maybe<T>`, `Completable`	`ListenableFuture<T>`	N/A
One-shot read	Coroutines (`suspend`)	`Single<T>`, `Maybe<T>`	`ListenableFuture<T>`	N/A
Observable read	`Flow<T>`	`Flowable<T>`, `Publisher<T>`, `Observable<T>`	N/A	`LiveData<T>`

Εικόνα 14 - Υποστήριξη Room και τύποι επιστροφής

Υπάρχουν τρεις πιθανοί τρόποι με τους οποίους μπορούμε να χρησιμοποιήσουμε ασύγχρονα ερωτήματα:

1) Με τη γλώσσα Kotlin.
2) Τη γλώσσα Java και το πλαίσιο RxJava.
3) Πάλι με τη γλώσσα Java σε συνδυασμό με τη βιβλιοθήκη LiveData και Guava.

Επιπρόσθετα μπορούμε να χρησιμοποιήσουμε το ExecutorService Interface στο οποίο δίνουμε μια εργασία για να εκτελεστεί από ένα νήμα(thread) ασύγχρονα.

4. ΥΛΟΠΟΙΗΣΗ – ΟΔΗΓΟΣ ΧΡΗΣΗΣ

Σε αυτή την ενότητα θα ακολουθήσει μια επίδειξη όλων όσων ειπώθηκαν προηγουμένως βάση της εφαρμογής που υλοποιήθηκε. Πιο συγκεκριμένα, θα γίνει λόγος σε γραμμές κώδικα που ικανοποιούν τις αρχές της αρχιτεκτονικής MVVM καθώς και επεξηγήσεις που θα ακολουθήσουν. Έπειτα θα ακολουθήσει ένας οδηγός χρήσης της εφαρμογής ώστε να γίνει κατανοητή ως προς τη χρήση της.

4.1. Εφαρμογή της κλάσης ViewModel

Η ViewModel είναι υπεύθυνη στο να ετοιμάσει τις πληροφορίες ώστε να εμφανιστούν στην οθόνη του χρήστη. Στο παρακάτω παράδειγμα θα δούμε πως τα αντικείμενα και πληροφορίες που περιλαμβάνει μια κλάση ViewModel διατηρούνται όταν αλλάζει η διαμόρφωση της συσκευής. Στο συγκεκριμένο παράδειγμα η εφαρμογή μας θα κρατάει το σκορ για δύο ομάδες.

Για την υλοποίηση της ViewModel χρειάζονται τρία βήματα:

1. Διαχωρίζουμε τα δεδομένα μας από τον ελεγκτή διεπαφής(UI controller) δημιουργώντας μια κλάση η οποία θα επεκτείνει τη ViewModel.
2. Να δημιουργήσουμε την επικοινωνία μεταξύ ViewModel και UI controller.
3. Να χρησιμοποιήσουμε τη ViewModel στον ελεγκτή διεπαφής.

1ο Βήμα: Δημιουργία της κλάσης ViewModel

Θα δημιουργήσουμε μια κλάση η οποία θα επεκτείνει την κλάση ViewModel. Μέσα σε αυτή θα αποθηκεύονται οι πληροφορίες που χρειαζόμαστε.

```java
public class MainActivityViewModel extends AndroidViewModel{
private List<Recipe> allRecipes;
    public MainActivityViewModel(@NonNull Application application) {
        super(application);
        allRecipes = initRecipes();
    }

    public List<Recipe> getAllRecipes(){
        return allRecipes;
    }

    private List<Recipe> initRecipes(){
        return Arrays.asList(new Recipe( title: "Τίτλος συνταγής1", creator: "Δημιουργός", mainPhoto: "φωτογραφία εξωφύλλου"),
                             new Recipe( title: "Τίτλος συνταγής2", creator: "Δημιουργός", mainPhoto: "φωτογραφία εξωφύλλου"));
    }
}
```

Εικόνα 15 - ViewModel κλάση

Όπως φαίνεται στην εικόνα 15 δημιουργήσαμε μια κλάση με το όνομα MainActivityViewModel. Η κλάση αυτή περιέχει τη μεταβλητή allRecipes, έναν κατασκευαστή, μια μέθοδο που επιστρέφει τις συνταγές και μια μέθοδο με την οποία αρχικοποιούμε κάποιες συνταγές.

Στο παραπάνω παράδειγμα χρησιμοποιούμε την υποκλάση AndroidViewModel, αντί της ViewModel, καθώς μας φανερώνει την παράμετρο application(με την οποία θα κάνουμε τη σύδεσή μας αργότερα με την αποθήκη δεδομένων-Repository).

2ο Βήμα: Σύνδεση του ελεγκτή διεπαφής(UI) με τη ViewModel κλάση

Ο ελεγκτής διεπαφής(Activity ή Fragment) που χρησιμοποιούμε θα πρέπει να γνωρίζει για τη κλάση ViewModel που δημιουργήσαμε, για να μπορεί να εμφανίσει τις συνταγές αλλά και να ενημερώσει αν προκύψουν τυχόν αλλαγές. Από την άλλη μια κλάση ViewModel δε πρέπει να γνωρίζει τίποτε για στοιχεία που υπάρχουν σε ένα ελεγκτή διεπαφής καθώς αυτό μπορεί να μας δημιουργήσει προβλήματα όπως διαρροή μνήμης.

Στην παρακάτω εικόνα φαίνεται πως επιτυγχάνεται η σύνδεση ελεγκτή διεπαφής και ViewModel με τη χρήση της ViewModelProvider.

```java
public class MainActivity extends AppCompatActivity {

    private RecyclerView recyclerView;
    private MainActivityViewModel viewModel;
    private MainActivityAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        viewModel = new ViewModelProvider( owner: this).get(MainActivityViewModel.class);
    }
```

Εικόνα 16 - Σύνδεση της ViewModel με τη MainActivity

Η πρώτη φορά που καλείται η ViewModelProvider δημιουργεί μια νέα αναφορά στην ViewModel κλάση μας, τη MainActivityViewModel. Κάθε φορά που καλείται η onCreate() μέθοδος του ελεγκτή διεπαφής επιστρέφει την αναφορά που δημιουργήθηκε πρώτη φορά. Με αυτό τον τρόπο διατηρούνται τα δεδομένα μας.

3ο Βήμα: Χρήση της ViewModel στον ελεγκτή διεπαφής

Αφού επιτεύχθηκε η σύνδεση μεταξύ ViewModel και ελεγκτή διεπαφής, μπορούμε τώρα να χρησιμοποιήσουμε τα δεδομένα που υπάρχουν μέσα στη κλάση ViewModel.

```java
public class MainActivity extends AppCompatActivity {

    private RecyclerView recyclerView;
    private MainActivityViewModel viewModel;
    private MainActivityAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initViews();

        viewModel = new ViewModelProvider( owner: this).get(MainActivityViewModel.class);
        adapter.setRecipes(viewModel.getAllRecipes());

        }

    private void initViews(){
        recyclerView = findViewById(R.id.recyclerView);
        recyclerView.setLayoutManager(new LinearLayoutManager(getBaseContext()));
        recyclerView.setHasFixedSize(true);
        adapter = new MainActivityAdapter();
        recyclerView.setAdapter(adapter);
    }
```

Εικόνα 17 - Στιγμιότυπο της MainActivity

Στην εικόνα 17 βλέπουμε την τελική έκδοση της MainActivity κλάσης. Περιλαμβάνει τις αρχικοποιήσεις των στοιχείων διάταξης που χρησιμοποιούμε (μια ανακυκλώσιμη λίστα που θα προβάλλει όλες τις συνταγές) και τη σύνδεση της ViewModel κλάσης με τη διεπαφή.

Καθώς ο χρήστης γυρίσει τη συσκευή οριζόντια ή σε πορτρέτο η MainActivity θα αναδημιουργείται με βάση την κλάση ViewModel.

4.2. Χρήση των LiveData αντικειμένων

Ο γενικός κανόνας για τη δημιουργία LiveData αντικειμένων είναι:

1. Μέσα στη ViewModel κλάση δημιουργούμε μια αναφορά LiveData για ένα συγκεκριμένο τύπο δεδομένων.
2. Δημιουργούμε ένα αντικείμενο παρατηρητή(Observer) που ορίζει τη μέθοδο onChanged(), η οποία ελέγχει τι γίνεται όταν τα δεδομένα LiveData αλλάζουν. Αυτό συμβαίνει συνήθως στον ελεγκτή διεπαφής(Activity/Fragment).

3. Στη συνέχεια συνδέουμε το αντικείμενο παρατηρητή που δημιουργήσαμε με το αντικείμενο LiveData με τη μέθοδο observe(). Η σύνδεση αυτή γίνεται στον ελεγκτή διεπαφής.

Όταν ενημερώσουμε μια τιμή στο αντικείμενο LiveData, τότε ειδοποιούνται όλοι οι παρατηρητές για τις αλλαγές, αρκεί ο κύκλος ζωής να είναι σε ενεργή κατάσταση.

Χρησιμοποιώντας LiveData ο ελεγκτής διεπαφής επιτρέπει στους παρατηρητές να λαμβάνουν ενημερώσεις. Όταν αλλάξουν τα δεδομένα του αντικειμένου LiveData, η διεπαφή χρήστη ενημερώνεται αυτόματα.

Δημιουργία αντικειμένου LiveData

Αντικείμενα LiveData μπορούμε να χρησιμοποιήσουμε σε οποιονδήποτε τύπο δεδομένων. Τα αντικείμενα αυτά βρίσκονται μέσα στη κλάση ViewModel και μπορούμε να αποκτήσουμε πρόσβαση χρησιμοποιώντας τη μέθοδο getter, όπως στο επόμενο παράδειγμα:

```
public class MainActivityViewModel extends AndroidViewModel{
private MutableLiveData<List<Recipe>> allRecipes;
    public MainActivityViewModel(@NonNull Application application) {
        super(application);
        allRecipes.setValue(initRecipes());
    }

    public LiveData<List<Recipe>> getAllRecipes(){
        return allRecipes;
    }

    private List<Recipe> initRecipes(){
        if (allRecipes == null) {
            allRecipes = new MutableLiveData<List<Recipe>>();
        }
            return Arrays.asList(new Recipe( title: "Τίτλος συνταγής1", creator: "Δημιουργός", mainPhoto: "φωτογραφία εξωφύλλου"),
                                 new Recipe( title: "Τίτλος συνταγής2", creator: "Δημιουργός", mainPhoto: "φωτογραφία εξωφύλλου"));
    }
}
```

Εικόνα 18 – Δημιουργία LiveData αντικειμένων

Τυλίγουμε τη λίστα από αντικείμενα Recipe με τη MutableLiveData, η οποία είναι μια υποκλάση της κλάσης LiveData. Για να έχουμε πρόσβαση στη

μεταβλητή μας από τον ελεγκτή διεπαφής δημιουργούμε μια μέθοδο getter, τη getAllRecipes(), που επιστρέφει ένα αντικείμενο LiveData.

Στο κατασκευαστή χρησιμοποιούμε μια μέθοδο όπου ελέγχει αν η μεταβλητή allRecipes είναι άδεια και στη συνέχεια τη γεμίζει με δύο διαφορετικές συνταγές.

Παρατηρώντας τα αντικείμενα LiveData

Συνήθως αρχίζουμε τη παρατήρηση στην onCreate() μέθοδο για τους εξής λόγους:

- Για να διασφαλίσουμε ότι το σύστημα δεν κάνει περιττές κλήσεις της μεθόδου onResume() σε μια δραστηριότητα(Activity ή ένα απόσπασμα(Fragment).
- Για να διασφαλίσουμε πως η διεπαφή χρήστη(Activity/Fragment) θα εμφανίσει τα δεδομένα όσο το δυνατόν γρηγορότερα γίνει ενεργή.

Η LiveData γενικά παρέχει ενημερώσεις μόνο όταν αλλάζουν τα δεδομένα και μόνο σε ενεργούς παρατηρητές(Observers). Επίσης ενημερώσεις γίνονται όταν η κατάσταση του παρατηρητή αλλάζει από ανενεργή(Inactive) σε ενεργή(Active).

Στο παράδειγμα που ακολουθεί απεικονίζεται ο τρόπος με τον οποίο αρχίζουμε τη παρατήρηση σε ένα αντικείμενο LiveData:

```
public class MainActivity extends AppCompatActivity {

    private RecyclerView recyclerView;
    private MainActivityViewModel viewModel;
    private MainActivityAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initViews();
        viewModel = new ViewModelProvider( owner: this).get(MainActivityViewModel.class);
        viewModel.getAllRecipes().observe( owner: this, new Observer<List<Recipe>>() {
            @Override
            public void onChanged(List<Recipe> recipes) {
                adapter.setRecipes(recipes);
            }
        });
    }
```

Εικόνα 19 – Δημιουργία παρατηρητή

Αφού κάνουμε τη σύνδεση της ViewModel κλάσης με τον ελεγκτή διεπαφής, καλούμε τη μέθοδο getAllRecipes() και χρησιμοποιούμε τη μέθοδο observe(). Η μέθοδος observe() χρειάζεται ως πρώτη παράμετρο έναν κύκλο ζωής(συγκεκριμένα της δραστηριότητας που χρησιμοποιούμε) που ορίζεται με τη λέξη κλειδί this και ως δεύτερη ένα αντικείμενο παρατηρητή. Κάθε φορά που αλλάζει η τιμή allRecipes η onChanged() μέθοδος καλείται ενημερώνοντας τη διεπαφή χρήστη. Αν η τιμή του LiveData αντικειμένου δεν αλλάξει, τότε δεν καλείται η onChanged().

Ενημέρωση των LiveData αντικειμένων

Χρησιμοποιώντας τη κλάση LiveData δε μπορούμε να ενημερώσουμε τα αποθηκευμένα δεδομένα. Η κλάση MutableLiveData μας παρέχει δύο μεθόδους τη setValue(T) και postValue(T) μες τις οποίες μπορούμε να επεξεργαστούμε ένα LiveData αντικείμενο. Συνήθως χρησιμοποιούμε MutableLiveData αντικείμενα στη ViewModel κλάση μας και εκθέτουμε αμετάβλητα LiveData αντικείμενα στους παρατηρητές.

```
public void addNewRecipe() {
    if (allRecipes.getValue() != null) {
        ArrayList<Recipe> recipes = new ArrayList<>(allRecipes.getValue());
        recipes.add(
                new Recipe( title: "Τίτλος συνταγής3", creator: "Δημιουργός", mainPhoto: "φωτογραφία εξωφύλλου"));
        allRecipes.setValue(recipes);
    }
}
```

Εικόνα 20– Ενημέρωση

Παραπάνω απεικονίζεται ένα παράδειγμα για το πως μπορούμε να ενημερώσουμε ένα αντικείμενο MutableLiveData. Καλώντας αυτή τη μέθοδο δημιουργούμε μια νέα λίστα, προσθέτουμε ακόμα μία νέα συνταγή και με τη βοήθεια της setValue(T) ενημερώνουμε το MutableLiveData αντικείμενο. Αυτό έχει ως αποτέλεσμα να κληθεί η onChanged() μέθοδος (εικόνα 19) με την ενημερωμένη λίστα.

4.3. Εφαρμογή της βιβλιοθήκης Room

Για την κατανόηση της λειτουργίας της βιβλιοθήκης Room θα δοθεί ένα παράδειγμα με οντότητα(entity) δεδομένων και ένα αντικείμενο πρόσβασης δεδομένων(DAO).

Οντότητα δεδομένων(Data entity)

Παρακάτω αναφέρεται μία οντότητα δεδομένων, η Recipe. Η κλάση Recipe είναι ένας πίνακας στη βάση δεδομένων μας. Κάθε αναφορά της Recipe αντιπροσωπεύει μια γραμμή στο πίνακα recipe της βάσης δεδομένων.

```
@Entity
public class Recipe {
    @PrimaryKey
    private int id;

    public String title;
    public String category;
}
```

Εικόνα 21 - Entity

Αντικείμενο πρόσβασης δεδομένων(DAO)

Το αντικείμενο πρόσβασης δεδομένων(DAO) μας παρέχει τις κατάλληλες μεθόδους που χρειαζόμαστε για να επικοινωνεί η εφαρμογή μας με τα διατηρήσιμα δεδομένα που υπάρχουν στους πίνακες της βάσης δεδομένων.

Στο παρακάτω κώδικα θα δούμε ένα interface DAO. Το RecipeDAO μας παρέχει τις μεθόδους που χρειαζόμαστε για να επικοινωνεί η εφαρμογή μας με τον πίνακα recipe.

```java
@Dao
public interface RecipeDAO {
    @Query("SELECT * FROM recipe")
    List<Recipe> getAllRecipes();

    @Query("SELECT * FROM recipe WHERE title LIKE (:title)")
    Recipe getRecipeWithTitle(String title);

    @Insert
    void insert(Recipe recipe);

    @Delete
    void delete(Recipe recipe);

    @Update
    void update(Recipe recipe);
}
```

Εικόνα 22- RecipeDAO

Όπως διακρίνεται υπάρχουν μέθοδοι χρήσιμοι για την εφαρμογή μας. Για λόγους ευχρηστίας η βιβλιοθήκη παρέχει τις συνήθεις μεθόδους Insert, Delete και Update. Επίσης μπορούμε να δημιουργήσουμε και πιο πολύπλοκα ερωτήματα για να ανταποκρίνονται στις ανάγκες της εφαρμογής μας με το annotation Query. Στα ερωτήματα έχουμε τη δυνατότητα να περνάμε και μεταβλητές όπως φαίνεται στη μέθοδο getRecipeWithTitle(String title).

Για να μην μπλοκάρει η διεπαφή χρήστη, για γρηγορότερη ανταπόκριση και να αποφύγουμε τη δημιουργία αρνητικών εντυπώσεων στο χρήστη, θα πρέπει τα ερωτήματα να γίνουν ασύγχρονα. Αυτό επιτυγχάνεται με τη βοήθεια κάποιου framework (RxJava, Guava) ή μέσω του ExecutorService interface.

Καθώς η εφαρμογή που αναπτύχθηκε είναι γραμμένη σε γλώσσα Java θα γίνει αναφορά μόνο σε αυτή τη γλώσσα σε συνδυασμό με τη LiveData βιβλιοθήκη και το πλαίσιο RxJava που χρησιμοποιήθηκε σε κάποιες περιπτώσεις.

Ασύγχρονα ερωτήματα

1) Ασύγχρονα ερωτήματα μιας λήψης(One-shot queries)

Τα ερωτήματα μιας λήψης τρέχουν μόνο μία φορά και επιστρέφουν ένα στιγμιότυπο της συγκεκριμένης στιγμής. Παρακάτω απεικονίζονται μερικά παραδείγματα:

```java
@Dao
public interface RecipeDAO {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public Completable insertRecipes(List<Recipe> recipes);

    @Update
    public Completable updateRecipes(List<Recipe> recipes);

    @Delete
    public Completable deleteRecipes(List<Recipe> recipes);

    @Query("SELECT * FROM recipe WHERE id = :id")
    public Single<User> getRecipeById(int id);
}
```

Εικόνα 23 - One-shot RxJava

Στην RxJava για τη χρήση μιας λήψης ασύγχρονων ερωτημάτων χρησιμοποιούνται οι εξής τύποι: Completable, Single και Maybe. Ενώ το JetPack δεν υποστηρίζει τη χρήση ασύγχρονων ερωτημάτων μιας λήψης.

2) Ασύγχρονα παρατηρίσιμα ερωτήματα(Observable queries)

Τα παρατηρήσιμα ερωτήματα είναι λειτουργίες ανάγνωσης που επιστρέφουν νέες τιμές κάθε φορά που γίνεται οποιαδήποτε αλλαγή στους πίνακες που εμπλέκονται. Είναι πολύ χρήσιμα όταν τα δεδομένα που εμφανίζονται στο χρήστη θέλουμε να είναι πάντα ενημερωμένα κατά τη χρήση λειτουργιών όπως εισαγωγή, ενημέρωση ή διαγραφή. Ακολουθούν μερικά παραδείγματα παρατηρήσιμων ερωτημάτων:

```java
@Dao
public interface RecipeDAO {

    @Query("SELECT * FROM recipe WHERE id = :id")
    public LiveData<Recipe> getRecipeById(int id);

    @Query("SELECT * FROM recipe WHERE title IN (:titles)")
    public LiveData<List<Recipe>> getRecipesByTitle(List<String> titles);
}
```

Εικόνα 24 - Observable LiveData examples

```java
@Dao
public interface RecipeDAO {

    @Query("SELECT * FROM recipe WHERE id = :id")
    public Flowable<Recipe> getRecipeById(int id);

    @Query("SELECT * from recipe WHERE title IN (:titles)")
    public Flowable<List<Recipe>> getRecipeByTitle(List<String> titles);
}
```

Εικόνα 25 - Observable RxJava examples

Η εικόνα 24 δείχνει τη χρήση της LiveData βιβλιοθήκης για τη δημιουργία ασύγχρονων παρατηρήσιμων ερωτημάτων, ενώ στην εικόνα 25 βλέπουμε τη χρήση του πλαισίου(framework) RxJava για τα αντίστοιχα ερωτήματα.

Βάση δεδομένων(Database)

Το επόμενο συστατικό είναι η βάση δεδομένων. Η κλάση AppDatabase είναι η υλοποίηση της βάσης δεδομένων. Είναι το κύριο σημείο πρόσβασης της εφαρμογής στα δεδομένα.

```
@Database(entities = {Recipe.class, User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract RecipeDAO recipeDAO();
    private static volatile AppDatabase INSTANCE;
}
```

Εικόνα 26- AppDatabase

Η κλάση που περιέχει τη βάση δεδομένων, πρέπει να πληρεί τις ακόλουθες προϋποθέσεις:

- Πρέπει να έχει το σχολιασμό(annotation) *@Database* και να περιλαμβάνει μια λίστα από τις οντότητες που σχετίζονται με τη βάση δεδομένων.
- Η κλάση πρέπει να δηλωθεί ως αφηρημένη(abstract) και να επεκτείνει(extends) τη RoomDatabase.
- Για κάθε interface που σχετίζεται με τη βάση δεδομένων, θα πρέπει να δηλωθεί ως μια αφηρημένη μέθοδος η οποία επιστρέφει μια αναφορά του DAO interface.

Η δήλωση της μεταβλητής INSTANCE ως volatile μας δίνει το προνόμιο η βάση μας να χρησιμοποιηθεί ταυτόχρονα από πολλά νήματα(threads) χωρίς να υπάρχει κάποιο πρόβλημα.

Έχοντας ορίσει την οντότητα δεδομένων(entity), το αντικείμενο πρόσβασης δεδομένων(DAO) και το αντικείμενο της βάσης δεδομένων(database), δημιουργούμε μια στατική μέθοδο με την οποία επιστρέφουμε μια αναφορά στη βάση δεδομένων μας με τον ακόλουθο κώδικα:

```
static AppDatabase getDatabase(final Context context) {
    if (INSTANCE == null) {
        synchronized (AppDatabase.class) {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(context.getApplicationContext()
                        , AppDatabase.class,  name: "CookAppDatabase")
                        .build();
            }
        }
    }
    return INSTANCE;
}
```

Εικόνα 27 - Database instance

Αφού δημιουργήσουμε τη σύνδεση μπορούμε να χρησιμοποιήσουμε τις αφηρημένες μεθόδους της AppDatabase και στη συνέχεια τις μεθόδους από το DAO για να επικοινωνούμε με τη βάση δεδομένων μας.

```
RecipeDAO recipeDAO = db.recipeDAO();
List<Recipe> recipes = recipeDAO.getAllRecipes();
```

Εικόνα 28 - DAO instance

ExecutorService Interface

Επιλέγουμε να χρησιμοποιήσουμε το ExecutorService Interface για χρήση ανάθεσης εργασιών ασύγχρονα σε ένα νήμα, χωρίς να μπλοκάρει το κύριο νήμα(main thread) και τη διεπαφή χρήστη. Άλλοι εναλλακτικοί τρόποι είναι η χρήση πλαισίου(framework) RxJava ή Guava.

```
private static final int NUMBER_OF_THREADS = 6;
static final ExecutorService databaseWriterExecutor = Executors.newFixedThreadPool(NUMBER_OF_THREADS);
```

Εικόνα 29 - ExecutorService setup

Οι παραπάνω γραμμές κώδικα θα δημιουργήσουν μια ομάδα από νήματα (συγκεκριμένα 6), τα οποία είναι έτοιμα να εκτελέσουν οποιαδήποτε εργασία τους ανατεθεί.

4.4. Η αποθήκη δεδομένων – Repository

Όπως ειπώθηκε στο προηγούμενο κεφάλαιο, η αποθήκη δεδομένων (Repository) είναι ο μεσολαβητής ανάμεσα στις διάφορες πηγές δεδομένων μιας εφαρμογής.

Καθώς η εφαρμογή χρησιμοποιεί μόνο μια τοπική βάση για τις ανάγκες της, η αποθήκη δεδομένων περιέχει μόνο τη σύνδεση με αυτή τη τοπική βάση.

Παρακάτω γίνεται αναπαράσταση του σχεδιαστικού προτύπου της αποθήκης δεδομένων.

```
public class Repository{
    private final RecipeDao recipeDao;

    public Repository(Application application) {
        Database database = Database.getDatabase(application);
        recipeDao = database.recipeDao();
    }

    public LiveData<List<Recipe>> getAllRecipes(){
        return recipeDao.getAllRecipes();
    }
}
```

Εικόνα 30 - Αποθήκη δεδομένων (Repository)

Η κλάση αυτή περιέχει ένα κατασκευαστή και μεθόδους που είναι χρήσιμες για την επικοινωνία της βάσης δεδομένων με την εφαρμογή.

Στον κατασκευαστή της κλάσης δημιουργούμε τη σύνδεση με τη τοπική βάση δεδομένων μας καλώντας τη στατική μέθοδο getDatabase(application) και στη συνέχεια υλοποιούμε τη σύνδεση με το interface(RecipeDAO), που περιλαμβάνει χρήσιμες μεθόδους για την επικοινωνία της εφαρμογής με τη τοπική βάση δεδομένων.

Με τη μέθοδο getAllRecipes() μέσω του αντικειμένου recipeDao επιστρέφουμε σε LiveData<T> μορφή όλες τις συνταγές της τοπικής βάσης δεδομένων.

4.5. Σύνδεση της αποθήκης δεδομένων με τη ViewModel

Καθώς έχουμε υλοποιήσει τη λειτουργικότητα της αποθήκης δεδομένων, το μόνο που απομένει να επιτύχουμε είναι η σύνδεση με τη ViewModel κλάση.

Τροποποιούμε την κλάση ViewModel ως εξής:

- Δημιουργούμε μια αναφορά ως προς την αποθήκη δεδομένων (Repository) και την αρχικοποιούμε μέσα στο κατασκευαστή με τη παράμετρο application.
- Δημιουργούμε ένα αντικείμενο LiveData<T> λίστα από συνταγές και το αρχικοποιούμε επίσης κι αυτό στον κατασκευαστή.

```
public class MainActivityViewModel extends AndroidViewModel {
    private Repository repository;
    private LiveData<List<Recipe>> recipes;

    public MainActivityViewModel(@NonNull Application application) {
        super(application);
        repository = new Repository(application);
        recipes = repository.getAllRecipes();
    }

    public LiveData<List<Recipe>> getAllRecipes(){
        return recipes;
    }

}
```

Εικόνα 31 - Η νέα ViewModel κλάση

Παραπάνω απεικονίζεται ένα στιγμιότυπο της νέας ViewModel κλάσης. Δεν αντιπροσωπεύει τη τελική μορφή της, καθώς έχουν αναπτυχθεί κι άλλες γραμμές κώδικα.

5. ΣΥΜΠΕΡΑΣΜΑΤΑ

Τα αρχιτεκτονικά και σχεδιαστικά πρότυπα μας δίνουν έναν οδηγό το πως θα πρέπει να χτιστεί η εφαρμογή ώστε να υπάρχει μια οργάνωση. Η ανάπτυξη μιας εφαρμογής χωρίς να έχει σαν πρότυπο μια αρχιτεκτονική μπορεί να οδηγήσει σε προβλήματα στο μέλλον όπως:

- Μεγάλες κλάσεις όπου δυσκολεύεσαι ακόμα και ο ίδιος να τις διαβάσεις και να βρεις τη λογική μέσα από αυτές.
- Είναι εξαιρετικά χρονοβόρος και δύσκολος ο εντοπισμός σφαλμάτων μέσα σε μια μεγάλη κλάση.
- Δυσκολία στην συντήρηση και σε τυχόν νέες λειτουργίες που ίσως προστεθούν αργότερα.

Αυτά είναι μερικά από τα προβλήματα που μπορεί κάποιος να βρεθεί αντιμέτωπος αν σκέφτεται να δημιουργήσει μια εφαρμογή μεγάλης έκτασης και υψηλών προδιαγραφών χωρίς τη χρήση προτύπων.

Υπάρχουν διάφορα αρχιτεκτονικά πρότυπα στο διαδίκτυο που μπορεί κάποιος να ακολουθήσει για την ανάπτυξη της εφαρμογής του. Τα σημαντικότερα που υπάρχουν μέχρι στιγμής είναι:

- MVVM(Model-View-ViewModel)
- MVC(Model-View-Controller)
- MVP(Model-View-Presenter)

Έχοντας χρησιμοποιήσει το αρχιτεκτονικό μοντέλο MVVM θα έλεγα πως στην αρχή ήταν λίγο δύσκολο να το κατανοήσω αλλά καθώς το μελετούσα και προχωρούσε το έργο, αντιλαμβανόμουνα καλύτερα τις πρακτικές του. Αυτό με βοήθησε κυρίως ώστε να γράψω ευανάγνωστο κώδικα διαχωρίζοντας τη λογική από τη διεπαφή χρήστη.

Υπάρχουν πολλά δομικά συστατικά(components) που μπορούν να ενταχθούν στο MVVM όπου θα σας γλιτώσουν χρόνο και θα δώσουν λύσεις στα προβλήματα σας. Επίσης υπάρχει τεράστιο υλικό στο διαδίκτυο καθώς χρησιμοποιείται από πολλούς προγραμματιστές και συστήνεται από την Google.

Έχοντας όλα αυτά υπόψιν προτρέπω τον καθένα να αρχίσει τη χρήση αρχιτεκτονικών προτύπων και συγκεκριμένα να δοκιμάσει το MVVM.

6. ΑΝΑΦΟΡΕΣ – ΒΙΒΛΙΟΓΡΑΦΙΕΣ

- Cay S. Horstmann "Core Java, Volume I - Fundamentals" Tenth Edition, December 2015.
- Daniel Y. Liang "Introduction to JAVA programming and data structures" Twelfth Edition, 2020.
- Tony Bevis "Java Design Pattern Essential", Ability First Limited, 2nd ed., 2012.
- Erich Gamma, Richard Helmes, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.
- Martin, Robert Cecil, "*UML for Java Programmers*", (2003).
- Introtorx.com, Introduction to Rx.
- Docs.microsoft.com, The MVVM Pattern, URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)?redirectedfrom=MSDN
- Developer.android.com, Guide to app architecture, URL: https://developer.android.com/jetpack/guide
- Medium.com, Understanding MVVM architecture in android, URL: https://medium.com/swlh/understanding-mvvm-architecture-in-android-aa66f7e1a70b

- Medium.com, Deep dive into MVVM architecture components, URL: https://medium.com/swlh/deep-dive-into-mvvm-architecture-components-bd1e3dfdb930
- Proandroiddev.com, Exploring RxJava in Android, URL: https://proandroiddev.com/exploring-rxjava-in-android-e52ed7ef32e2