



ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

Διεθνές Πανεπιστήμιο Ελλάδος
Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεδιασμός και Υλοποίηση Εφαρμογής Android για Διαχείριση Ανακοινώσεων Οργανισμού



Φοιτητής:

Κώστας Παπασταθόπουλος
Αριθμός Μητρώου: 185255

Επιβλέπων:

Αντώνης Σιδηρόπουλος

26 Μαΐου 2026

Title of Dissertation Design and Implementation of an Android Application for Managing
Organization Announcements

Code of Dissertation 25353

Student's full name Kostas Papastathopoulos

Supervisor's full name Antonis Sidiropoulos

Date of undertaking 04-11-2025

Date of completion 26-05-2026

I hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance I received in its composition. I have, furthermore, noted the various sources from which I extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, I affirm the exclusive composition of this paper by myself only, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

This paper constitutes the intellectual property of Kostas Papastathopoulos the student that composed it. According to the open-access policy, the author/composer offers the International Hellenic University authorisation to use the right to reproduce, borrow, publicly present and digitally distribute the paper globally, in electronic form and media of all kinds, for teaching or research purposes, voluntarily. Open access to the full text, by no means grants the right to trespass the intellectual property of the author/composer, nor does it authorise the reproduction, republication, duplication, selling, commercial use, distribution, publication, downloading, uploading, translation, modification of any kind, in part or summary of the paper, without the explicit written consent of the author.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University, does not necessarily entail the adoption of the author's views, on behalf of the Department.

Πρόλογος

Η ιδέα για την εκπόνηση της παρούσας διπλωματικής εργασίας γεννήθηκε μέσα από μια καθημερινή, πρακτική ανάγκη. Παρατηρούσα συχνά τη δυσκολία που αντιμετωπίζαμε εγώ και οι συμφοιτητές μου στο να παρακολουθούμε έγκαιρα τις πανεπιστημιακές ανακοινώσεις, καθώς οι υπάρχουσες πλατφόρμες απαιτούσαν διαρκή χειροκίνητο έλεγχο και δεν ήταν βελτιστοποιημένες για κινητές συσκευές. Αποφάσισα, λοιπόν, να δημιουργήσω ένα εργαλείο που θα γεφύρωνε αυτό το κενό, προσφέροντας άμεση και προσωποποιημένη ενημέρωση.

Πέρα από την επίλυση ενός πρακτικού προβλήματος της ακαδημαϊκής μας κοινότητας, το έργο αυτό αποτέλεσε και ένα προσωπικό στοίχημα εξέλιξης. Έχοντας προηγουμένως ασχοληθεί με την ανάπτυξη Cross-Platform εφαρμογών, συνειδητοποίησα την επιθυμία μου να εστιάσω και να εμβαθύνω αποκλειστικά στη native ανάπτυξη για το οικοσύστημα του Android. Ήθελα να αξιοποιήσω την υπάρχουσα εμπειρία μου για να κατασκευάσω κάτι που δεν θα ήταν απλώς λειτουργικό, αλλά θα ακολουθούσε αυστηρά τα πιο σύγχρονα βιομηχανικά πρότυπα.

Ο πρωταρχικός μου στόχος ήταν να δημιουργήσω μια αρχιτεκτονικά άρτια εφαρμογή η οποία θα μπορούσε να σταθεί ως ένα έργο παραγωγικών προδιαγραφών. Ελπίζω το τελικό αποτέλεσμα να αποτελέσει ένα πρακτικό, καθημερινό βοήθημα για τους φοιτητές του τμήματος, και η παρούσα τεκμηρίωση να φανεί χρήσιμη σε όσους αναζητούν έναν ρεαλιστικό οδηγό στις σύγχρονες τεχνολογίες του Android.

Abstract

The widespread adoption of mobile devices has transformed the way students access academic information. In university settings, announcements represent a primary communication channel between faculty and students however, existing platforms are often not optimised for mobile use and lack real-time notification support.

This thesis presents the design and development of an Android application, *IEE IHU*, for aggregating and presenting university announcements to students of the Department of Information and Electronic Engineering at the International Hellenic University. The application provides a mobile-first interface with announcement browsing, full-text search, tag-based filtering, personalised subscriptions, attachment downloads, and JWT-based authentication, providing access to both public and private announcements.

The implementation follows the **Modern Android Development (MAD)** paradigm. The user interface is built entirely with **Jetpack Compose**, following the Unidirectional Data Flow (UDF) pattern and Material Design 3 guidelines. The architecture adheres to **Clean Architecture** with the **MVVM** pattern, organized in a multi-module structure with custom Gradle convention plugins. Dependency injection is handled by **Hilt**, ensuring low coupling and high testability.

Data management follows an **offline-first** approach: **Room** serves as the Single Source of Truth, while the **Paging 3** `RemoteMediator` transparently synchronizes the local database with the REST API. Scheduled background synchronization for notifications is implemented via **WorkManager**, since the Backend does not provide push notifications.

Performance was evaluated using the **Macrobenchmark** framework. **Baseline Profile** and **Startup Profile** reduced median Cold Start time by **8.24%** (from ~ 520 ms to ~ 477 ms). A comprehensive suite of unit tests and manual verification of **seven** core use cases confirmed the correctness of the implementation in real-world conditions.

The application has been published on the Google Play Store and is available as an open-source project on GitHub. The results demonstrate that the MAD framework enables the development of performant, maintainable, scalable, and reliable mobile applications suitable for academic communication environments.

Περίληψη

Η συνεχής αύξηση της χρήσης κινητών συσκευών έχει αλλάξει τον τρόπο που οι φοιτητές προσπελάνουν ακαδημαϊκές πληροφορίες. Στο πλαίσιο των πανεπιστημιακών ιδρυμάτων, οι ανακοινώσεις αποτελούν βασικό μέσο επικοινωνίας μεταξύ καθηγητών και φοιτητών ωστόσο, οι υπάρχουσες πλατφόρμες συχνά δεν είναι βελτιστοποιημένες για κινητές συσκευές και δεν υποστηρίζουν άμεσες ειδοποιήσεις.

Η παρούσα εργασία παρουσιάζει τη σχεδίαση και ανάπτυξη της εφαρμογής *IEE IHU* για το λειτουργικό σύστημα Android, η οποία συγκεντρώνει πανεπιστημιακές ανακοινώσεις του Τμήματος Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙΠΑΕ. Η εφαρμογή παρέχει προβολή ανακοινώσεων, πλήρη αναζήτηση, φιλτράρισμα ανά ετικέτα, προσωποποιημένες συνδρομές και ειδοποιήσεις, λήψη συνημμένων και αυθεντικοποίηση μέσω JWT, παρέχοντας πρόσβαση τόσο σε δημόσιες όσο και σε ιδιωτικές ανακοινώσεις.

Η υλοποίηση ακολουθεί το πρότυπο **Modern Android Development (MAD)**. Η διεπαφή χρήστη κατασκευάστηκε αποκλειστικά με **Jetpack Compose**, ακολουθώντας το μοτίβο UDF και τις αρχές Material Design 3. Η αρχιτεκτονική βασίζεται στο **Clean Architecture** με **MVVM** και multi-module δομή με προσαρμοσμένα Gradle convention plugins. Το **Hilt** διαχειρίζεται τη ροή εξαρτήσεων, εξασφαλίζοντας χαμηλή σύζευξη και υψηλή δυνατότητα ελέγχου.

Η διαχείριση δεδομένων ακολουθεί **offline-first** προσέγγιση: Το **Room** λειτουργεί ως Μοναδική Πηγή Αλήθειας, το **Paging 3** διαχειρίζεται τη σελιδοποιημένη φόρτωση μέσω **RemoteMediator** συγχρονίζοντας με διαφάνεια την τοπική βάση με το REST API, και το **WorkManager** εξασφαλίζει αξιόπιστο polling για ειδοποιήσεις, καθώς το Backend δεν παρέχει push notifications.

Η αξιολόγηση μέσω **Macrobenchmark** έδειξε βελτίωση του χρόνου Cold Start κατά **8.24%** (από ~520 ms σε ~477 ms) μέσω Baseline και Startup Profile. Μια πλήρης σουίτα unit tests και η χειροκίνητη επαλήθευση επτά βασικών σεναρίων χρήσης επιβεβαίωσαν την ορθή λειτουργία του συστήματος σε πραγματικές συνθήκες.

Η εφαρμογή είναι διαθέσιμη στο Google Play Store και στο GitHub. Τα αποτελέσματα αποδεικνύουν ότι το πλαίσιο MAD επιτρέπει την ανάπτυξη αποδοτικών, συντηρήσιμων, επεκτάσιμων και αξιόπιστων εφαρμογών κατάλληλων για εκπαιδευτικά περιβάλλοντα.

Περιεχόμενα

1	Εισαγωγή	1
1.1	Υπόβαθρο	1
1.2	Κίνητρο της Εργασίας	2
1.3	Στόχοι της Εργασίας	2
1.4	Ανάλυση Απαιτήσεων Συστήματος	3
1.4.1	Λειτουργικές Απαιτήσεις	3
1.4.2	Μη Λειτουργικές Απαιτήσεις	3
1.4.3	Περιορισμοί Συστήματος και Backend	4
1.5	Μεθοδολογία Υλοποίησης	4
1.6	Προαπαιτούμενες Γνώσεις	5
1.7	Δομή της Εργασίας	5
2	Τεχνολογικό Υπόβαθρο	6
2.1	Εισαγωγή	6
2.2	Επιλογή Πλατφόρμας Ανάπτυξης	6
2.3	Γλώσσα και Ασύγχρονος Προγραμματισμός	7
2.3.1	Kotlin: Η Επίσημη Γλώσσα Android	7
2.3.2	Από Threads σε Coroutines: Εξέλιξη του Ασύγχρονου Μοντέλου	7
2.3.3	Flow και StateFlow: Αντιδραστική Ροή Δεδομένων	8
2.4	Αρχιτεκτονική Εφαρμογής	9
2.4.1	Διαχείριση Κύκλου Ζωής και ViewModels	9
2.4.2	MVVM	9
2.4.3	Clean Architecture	10
2.4.4	Πολυαρθρωτή Δομή	11
2.4.5	Dependency Injection	11
2.5	Jetpack Compose	12
2.5.1	Composable Functions και Recomposition	13
2.5.2	State-Hoisting και Unidirectional Data Flow (UDF)	14
2.5.3	Type-Safe Navigation με Navigation Compose	15
2.6	Τεχνολογίες Αποθήκευσης και Δικτύου	16
2.6.1	Επικοινωνία με το Backend: Retrofit και REST	16
2.6.2	Τοπική Αποθήκευση: Room και SQLite	17
2.6.3	Offline-First: Room ως Μοναδική Πηγή Αλήθειας	18
2.6.4	Paging 3 και RemoteMediator: Σελιδοποίηση με Offline Υποστήριξη	18
2.6.5	DataStore: Αποθήκευση Προτιμήσεων	19

2.7	Σύνοψη	21
3	Αρχιτεκτονική και Δομή Εφαρμογής	22
3.1	Εισαγωγή	22
3.2	Συνολική Αρχιτεκτονική του Συστήματος	22
3.2.1	Τοπολογία Συστήματος	22
3.2.2	Αντιστοίχιση Επιπέδων Clean Architecture	23
3.3	Δομή του Έργου και Πολυαρθρωτή Δομή	23
3.3.1	App Module	24
3.3.2	Core Modules	24
3.3.3	Feature Modules	24
3.3.4	Build-Logic Module	25
3.4	Dependency Injection με Hilt	25
3.5	Σύνοψη	27
4	Υλοποίηση Data Layer	29
4.1	Εισαγωγή	29
4.2	Μοντέλα Δεδομένων	30
4.2.1	Αιτιολόγηση της Τριεπίπεδης Δομής	30
4.2.2	DTOs: Η Αναπαράσταση του API Response	30
4.2.3	Entities: Η Αναπαράσταση στη Βάση Δεδομένων	31
4.2.4	Mapper Functions: Μετατροπή μεταξύ Επιπέδων	32
4.3	Επικοινωνία Δικτύου	32
4.3.1	Επιλογή Βιβλιοθήκης Δικτύου	32
4.3.2	API Interface: AboardApiClient	33
4.3.3	Διαχείριση Αυθεντικοποίησης	33
4.3.4	Remote Data Source: Ενθυλάκωση της API Κλήσης	33
4.4	Τοπική Βάση Δεδομένων: Room	34
4.4.1	Επιλογή Βιβλιοθήκης Βάσης Δεδομένων	34
4.4.2	AppDatabase: Ορισμός Βάσης	34
4.4.3	DAOs: Δηλωτική Πρόσβαση στη Βάση	34
4.4.4	Room Relations: Σύνθετες Αναπαραστάσεις	34
4.5	Paging 3 και RemoteMediator	35
4.5.1	Αρχιτεκτονική Offline-First με Paging 3	35
4.5.2	Υλοποίηση και Κανονικοποιημένη Αποθήκευση	35
4.5.3	Ρύθμιση του Pager	36
4.6	Αποθήκευση Προτιμήσεων: DataStore	36
4.6.1	Επιλογή Μηχανισμού Αποθήκευσης Προτιμήσεων	36
4.6.2	Υλοποίηση και Θωράκιση	36
4.7	Διαχείριση και Χαρτογράφηση Σφαλμάτων	37
4.8	Επεξεργασία Εικόνων: Base64ImageExtractor	37
4.9	Συνημμένα Αρχεία: DownloadManager	38
4.10	Σύνοψη	38
5	Υλοποίηση Domain Layer	39

5.1	Εισαγωγή	39
5.2	Domain Models	40
5.3	Repository Interfaces	40
5.4	Use Cases	41
5.5	Σύνοψη	42
6	Υλοποίηση UI Layer	43
6.1	Εισαγωγή	43
6.2	ViewModels	44
6.2.1	Δομή State και Αρχικοποίηση	44
6.3	Συναρτήσεις Composable	45
6.3.1	State Collection & Hoisting	45
6.4	Πλοήγηση	46
6.4.1	Type-Safe Navigation	46
6.4.2	Deep Linking	47
6.5	Design System	47
6.5.1	Material Design 3	47
6.5.2	Επιπλέον Χαρακτηριστικά	47
6.6	Παρουσίαση Διεπαφής Χρήστη	48
6.6.1	Αρχική Οθόνη Ανακοινώσεων	49
6.6.2	Οθόνη Αναζήτησης	50
6.6.3	Οθόνη Λεπτομερειών Ανακοίνωσης	51
6.6.4	Οθόνη Προφίλ και Διαχείρισης Συνδρομών	52
6.6.5	Οθόνη Ρυθμίσεων	53
6.7	Σύνοψη	54
7	Οριζόντιες Λειτουργίες	55
7.1	Εισαγωγή	55
7.2	Σύστημα Ειδοποιήσεων και WorkManager	55
7.2.1	Επιχειρησιακή Λογική Αναζήτησης	55
7.2.2	Εκτέλεση στο Παρασκήνιο	56
7.2.3	Χρονοπρογραμματισμός και Διαχείριση	57
7.3	Βελτιστοποίηση Απόδοσης	58
7.3.1	Code Shrinking και Βελτιστοποίηση με R8	58
7.3.2	Baseline Profile	59
7.3.3	Startup Profile	60
7.3.4	Macrobenchmark Tests	61
7.4	Στρατηγική Δοκιμών	61
7.5	Σύνοψη	62
8	Αποτελέσματα	63
8.1	Εισαγωγή	63
8.2	Λειτουργική Επαλήθευση	63
8.2.1	Στρατηγική Δοκιμών και Κάλυψη	63
8.2.2	Αποτελέσματα Unit Tests	63

8.2.3	Σενάρια Χρήσης και Λειτουργική Ορθότητα	64
8.2.4	Επαλήθευση Offline-First Συμπεριφοράς	65
8.3	Απόδοση: Μετρήσεις Macrobenchmark	66
8.3.1	Μεθοδολογία Μέτρησης	66
8.3.2	Χρόνοι Εκκίνησης	66
8.4	Συζήτηση και Περιορισμοί	67
8.5	Σύνοψη	67
9	Συμπεράσματα και Μελλοντική Εργασία	69
9.1	Συμπεράσματα	69
9.2	Συνεισφορά της Εργασίας	70
9.3	Μελλοντική Εργασία	70
	Βιβλιογραφία	72
A	Εγχειρίδιο χρήσης	75
A.1	Εκκίνηση Εφαρμογής και Ταυτοποίηση	75
A.1.1	Περιήγηση ως Επισκέπτης	76
A.1.2	Σύνδεση στον Λογαριασμό	76
A.1.3	Ανανέωση Περιεχομένου	77
A.2	Προφίλ Χρήστη και Διαχείριση Ενδιαφερόντων	78
A.3	Πλοήγηση και Αναζήτηση Ανακοινώσεων	80
A.3.1	Προβολή Αρχικής Σελίδας	80
A.3.2	Δυναμική Φόρτωση Αποτελεσμάτων	81
A.3.3	Αναζήτηση	82
A.3.4	Εφαρμογή Φίλτρων	83
A.3.5	Προβολή Αποτελεσμάτων	84
A.4	Ρυθμίσεις Εφαρμογής	88
A.4.1	Επιλογές Ροών	89
A.4.2	Επιλογές Αναζήτησης	91
A.4.3	Επιλογές Εμφάνισης	92
A.4.4	Επιλογές Ειδοποιήσεων	93
A.4.5	Σχετικά με την Εφαρμογή	94
A.4.6	Περιορισμοί Λειτουργίας Επισκέπτη	95

Κατάλογος σχημάτων

2.1	Τα τρία επίπεδα του Clean Architecture και οι κατευθύνσεις εξάρτησης [13]. . .	10
2.2	Ενδεικτικό γράφημα εξαρτήσεων modules [11].	11
2.3	Το μοτίβο UDF: η κατάσταση κατεβαίνει προς το UI, τα events ανεβαίνουν προς το ViewModel [22].	15
2.4	Η στρωματωμένη αρχιτεκτονική Paging 3: Repository, ViewModel και UI συνεργάζονται για αποδοτική φόρτωση δεδομένων [28].	19
4.1	Αρχιτεκτονική Επισκόπηση: Data Layer [24]	30
4.2	Κύκλος ζωής Paging 3 με RemoteMediator	35
5.1	Αρχιτεκτονική Επισκόπηση: Επίπεδο Domain [23]	39
6.1	Αρχιτεκτονική Επισκόπηση: Επίπεδο UI [22]	43
6.2	Το UDF μοτίβο σε δράση: StateFlow ρέει προς το Composable, ενώ τα events χρήστη ανεβαίνουν αντίθετα [22]	44
6.3	Αρχική οθόνη ροή ανακοινώσεων με καρτέλες «Αρχική» / «Εγγεγραμμένα» . .	49
6.4	Οθόνη αναζήτησης με ενεργοποιημένα φίλτρα ετικετών	50
6.5	Οθόνη λεπτομερειών ανακοίνωσης με πλήρες κείμενο και συνημμένα	51
6.6	Οθόνη προφίλ χρήστη	52
6.7	Οθόνη ρυθμίσεων θέμα, συχνότητα ειδοποιήσεων	53
A.1	Αναδυόμενο παράθυρο προτροπής για σύνδεση	75
A.2	Αρχική οθόνη μη συνδεδεμένου χρήστη	76
A.3	Οθόνη πιστοποίησης	77
A.4	Οθόνη φόρτωσης	77
A.5	Οθόνη «Εγγεγραμμένα»	77
A.6	Ανανέωση περιεχομένου με κίνηση pull-to-refresh	78
A.7	Πρόσβαση στο Προφίλ	79
A.8	Προφίλ χρήστη	79
A.9	Μενού διαχείρισης	79
A.10	Αποσύνδεση χρήστη	80
A.11	Αρχική οθόνη επισκέπτη	80
A.12	Συμπτυγμένο FAB	81
A.13	Αναπτυγμένο FAB	81
A.14	Φόρτωση επιπλέον ανακοινώσεων	82
A.15	Μήνυμα τέλους	82

A.16 Μήνυμα αποτυχίας	82
A.17 Αναπτυγμένη γραμμή αναζήτησης με προτεινόμενα αποτελέσματα	83
A.18 Φιλτράρισμα αποτελεσμάτων κατά την πληκτρολόγηση	83
A.19 Κουμπιά πρόσβασης στα μενού φίλτρων μέσω της γραμμής αναζήτησης	84
A.20 Κουμπιά πρόσβασης στα φίλτρα δίπλα από το κουμπί FAB	84
A.21 Ενεργά φίλτρα ετικετών	85
A.22 Ενεργή αναζήτηση κειμένου	85
A.23 Οθόνη φόρτωσης αρχικών αποτελεσμάτων αναζήτησης	86
A.24 Ενημέρωση αποθηκευμένων αποτελεσμάτων αναζήτησης	87
A.25 Μήνυμα απουσίας αποτελεσμάτων αναζήτησης	88
A.26 Επισήμανση του εικονιδίου ρυθμίσεων στην αρχική οθόνη	89
A.27 Επιλογή ταξινόμησης των ροών	90
A.28 Ενεργοποίηση της ροής «Εγγεγραμμένα»	90
A.29 Ρύθμιση εμφάνισης φίλτρων δίπλα στο FAB	91
A.30 Αρχική οθόνη με απενεργοποιημένα τα φίλτρα του FAB	91
A.31 Επιλογή εμβέλειας αναζήτησης	92
A.32 Επιλογή θέματος εμφάνισης	93
A.33 Ρύθμιση δυναμικού χρώματος	93
A.34 Μετάβαση στις ρυθμίσεις ειδοποιήσεων	94
A.35 Ρύθμιση συχνότητας ελέγχου για νέες ειδοποιήσεις	94
A.36 Προβολή έκδοσης και σύνδεσμος GitHub	95
A.37 Προβολή των αδειών ανοιχτού κώδικα	95
A.38 Σύνδεσμος Discord	95
A.39 Απενεργοποιημένη επιλογή «Εγγεγραμμένα» και ρύθμιση ειδοποιήσεων για μη συνδεδεμένους χρήστες	96

Κατάλογος πινάκων

2.1	Σύγκριση native και Cross-Platform ανάπτυξης	7
3.1	Αντιστοίχιση επιπέδων Clean Architecture σε modules και κατεύθυνση εξάρτησης	23
8.1	Συγκεντρωτικά αποτελέσματα αυτοματοποιημένων δοκιμών ανά module	64
8.2	Αποτελέσματα χειροκίνητων δοκιμών σεναρίων χρήσης σε φυσικό hardware . .	65
8.3	Αξιολόγηση συμπεριφοράς offline-first αρχιτεκτονικής	66
8.4	Χρόνοι εκκίνησης ανά διαμόρφωση	66

Κεφάλαιο 1

Εισαγωγή

1.1 Υπόβαθρο

Η ευρεία διάδοση των κινητών συσκευών τα τελευταία χρόνια έχει μεταμορφώσει σημαντικά τον τρόπο με τον οποίο οι άνθρωποι αποκτούν πρόσβαση σε πληροφορίες και ψηφιακές υπηρεσίες. Τα smartphones αποτελούν πλέον βασικό εργαλείο καθημερινής επικοινωνίας, ενημέρωσης και παραγωγικότητας, επιτρέποντας στους χρήστες να αλληλεπιδρούν με διάφορα πληροφοριακά συστήματα ανεξάρτητα από τον χρόνο και τον τόπο.

Ειδικότερα, στο πλαίσιο της τριτοβάθμιας εκπαίδευσης, η άμεση ενημέρωση των φοιτητών αποτελεί κρίσιμη λειτουργική απαίτηση. Στο Διεθνές Πανεπιστήμιο της Ελλάδος (ΔΠΙΑΕ), οι ανακοινώσεις καλύπτουν ένα ευρύ φάσμα θεμάτων από το ωρολόγιο πρόγραμμα και τις εξετάσεις, έως διοικητικές ενημερώσεις και ακαδημαϊκές εκδηλώσεις. Παρόλο που οι πληροφορίες αυτές είναι διαθέσιμες μέσω διαδικτυακών πυλών, οι πλατφόρμες αυτές δεν είναι πλήρως βελτιστοποιημένες για χρήση σε κινητές συσκευές, ούτε υποστηρίζουν άμεσες ειδοποιήσεις (push notifications).

Ως εκ τούτου, αν και όλες οι ανακοινώσεις συγκεντρώνονται σε μία ενιαία ιστοσελίδα του τμήματος, οι φοιτητές αναγκάζονται να την επισκέπτονται χειροκίνητα και σε τακτά χρονικά διαστήματα για να ενημερωθούν. Η διαδικασία αυτή αποδεικνύεται συχνά δύσχρηστη στην καθημερινότητα και είναι ιδιαίτερα επιρρεπής σε παραλείψεις.

Για να κατανοηθεί η κλίμακα του προβλήματος, το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΠΙΑΕ εξυπηρετεί μεγάλο αριθμό φοιτητών και δημοσιεύει πλήθος ανακοινώσεων, κατανεμημένων σε δεκάδες μαθήματα και διοικητικές κατηγορίες. Δεδομένου ότι κάθε φοιτητής παρακολουθεί τουλάχιστον πέντε μαθήματα ανά εξάμηνο, ο εντοπισμός των πληροφοριών που τον αφορούν μέσα σε μια ενιαία, συνεχώς ανανεούμενη λίστα καθίσταται χρονοβόρος μέσω ενός απλού περιηγητή (browser) σε κινητή συσκευή. Ελλείπει αυτοματοποιημένων ειδοποιήσεων, κρίσιμες πληροφορίες όπως αλλαγές στον χώρο εξέτασης ή η ακύρωση ενός μαθήματος ενδέχεται να μην εντοπιστούν εγκαίρως.

Αξίζει να διευκρινιστεί ότι το πανεπιστήμιο διαθέτει ήδη ένα σύστημα υποστήριξης (Backend), δηλαδή έναν διακομιστή που αποθηκεύει τις ανακοινώσεις και παρέχει πρόσβαση σε αυτές μέσω διαδικτύου. Η παρούσα εργασία δεν επιχειρεί να αντικαταστήσει αυτό το σύστημα, αλλά να αναπτύξει μια εφαρμογή Android, η οποία θα επικοινωνεί μαζί του και θα προσφέρει στους φοιτητές ένα βελτιστοποιημένο και φιλικό περιβάλλον χρήσης.

1.2 Κίνητρο της Εργασίας

Το βασικό κίνητρο για την εκπόνηση της παρούσας εργασίας είναι η ανάγκη δημιουργίας ενός εργαλείου που θα αντιμετωπίσει τους προαναφερθέντες περιορισμούς, παρέχοντας στους φοιτητές έναν εύχρηστο και αποδοτικό τρόπο πρόσβασης στις πανεπιστημιακές ανακοινώσεις. Η υλοποίηση μιας εφαρμογής που συγκεντρώνει τα δεδομένα από το υπάρχον Backend σύστημα σε ένα ενιαίο περιβάλλον αναμένεται να μειώσει σημαντικά τον χρόνο αναζήτησης πληροφοριών και να αναβαθμίσει την εμπειρία των φοιτητών.

Παράλληλα, η ανάπτυξη της εφαρμογής αποτελεί ιδανική ευκαιρία για την πρακτική εφαρμογή σύγχρονων τεχνολογιών και βέλτιστων πρακτικών. Τα τελευταία χρόνια, η πλατφόρμα Android έχει εξελιχθεί ραγδαία, εισάγοντας νέες βιβλιοθήκες και εργαλεία που διευκολύνουν τη δημιουργία αποδοτικών και επεκτάσιμων εφαρμογών [1].

Ειδικότερα, η χρήση τεχνολογιών όπως το Jetpack Compose για δηλωτικές (declarative) διεπαφές χρήστη [2], η αρχιτεκτονική MVVM για τον διαχωρισμό της επιχειρησιακής λογικής από την παρουσίαση [3], και βιβλιοθήκες όπως το Room [4], το Paging 3 [5] και το WorkManager [6], επιτρέπουν την ανάπτυξη λογισμικού που συνδυάζει υψηλή απόδοση, ευκολία συντήρησης και αξιοπιστία. Οι τεχνολογίες αυτές αναλύονται διεξοδικά στο Κεφάλαιο 2.

Επιπλέον, η αξιολόγηση της απόδοσης της εφαρμογής αποτελεί ξεχωριστό αντικείμενο μελέτης. Η χρήση εργαλείων όπως το Macrobenchmark επιτρέπει την ανάλυση κρίσιμων λειτουργιών ιδίως του χρόνου εκκίνησης σε πραγματικές συνθήκες χρήσης, καθώς και τη βελτιστοποίηση τους μέσω τεχνικών όπως τα Baseline Profile και Startup Profile [7].

1.3 Στόχοι της Εργασίας

Στόχος της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και η ανάπτυξη μιας εφαρμογής για το λειτουργικό σύστημα Android, η οποία θα αντλεί πανεπιστημιακές ανακοινώσεις από το υπάρχον σύστημα Backend του ΔΠΙΑΕ και θα τις παρουσιάζει στους φοιτητές μέσω μιας σύγχρονης και εργονομικής διεπαφής.

Για την επίτευξη αυτού του σκοπού, η εργασία δομείται γύρω από τρεις βασικούς άξονες, οι οποίοι γεφυρώνουν την πρακτική επίλυση του προβλήματος με τη μελέτη σύγχρονων προτύπων μηχανικής λογισμικού:

- **Εφαρμογή Σύγχρονων Πρακτικών Ανάπτυξης (Modern Android Development):** Η υλοποίηση στοχεύει στην πλήρη υιοθέτηση των τρεχουσών βέλτιστων πρακτικών, εστιάζοντας στη δημιουργία μιας *offline-first* εμπειρίας. Αυτό προϋποθέτει τον σχεδιασμό ενός ισχυρού μηχανισμού τοπικής αποθήκευσης και συγχρονισμού, ώστε η εφαρμογή να παραμένει λειτουργική και γρήγορη ανεξαρτήτως της ποιότητας σύνδεσης στο διαδίκτυο.
- **Αρχιτεκτονική Ελεγκτασιμότητα και Αποσύνδεση:** Δεύτερος στόχος είναι η οργάνωση του κώδικα σύμφωνα με τις αρχές της Clean Architecture μέσα σε μια πολυαρθρωτή (multi-module) δομή. Ο διαχωρισμός των επιπέδων (Παρουσίαση, Επιχειρησιακή Λογική, Δεδομένα) αποσκοπεί στη δημιουργία ενός συστήματος εύκολα συντηρήσιμου, επεκτάσιμου και επιδεκτικού σε αυτοματοποιημένους ελέγχους (testing).
- **Μετρήσιμη Βελτιστοποίηση Απόδοσης:** Ο τρίτος άξονας αφορά την αυστηρή αξιολόγηση και βελτίωση της απόδοσης της εφαρμογής. Μέσω εργαλείων όπως το

Macrobenchmark και τεχνικών όπως τα Baseline Profile, στόχος είναι η ποσοτική μέτρηση και ελαχιστοποίηση του χρόνου εκκίνησης (startup time) και η διασφάλιση της ομαλής λειτουργίας της διεπαφής (smooth scrolling).

1.4 Ανάλυση Απαιτήσεων Συστήματος

Για τη μετάβαση από τον εντοπισμό του προβλήματος στον τεχνικό σχεδιασμό της λύσης, κρίνεται απαραίτητος ο αυστηρός καθορισμός των απαιτήσεων και των περιορισμών της εφαρμογής. Η ανάλυση αυτή οργανώνεται σε τρεις κατηγορίες: λειτουργικές απαιτήσεις, μη λειτουργικές απαιτήσεις και περιορισμούς υποδομής.

1.4.1 Λειτουργικές Απαιτήσεις

Οι λειτουργικές απαιτήσεις περιγράφουν τις άμεσες δυνατότητες που πρέπει να παρέχει η εφαρμογή στον τελικό χρήστη:

- **Προβολή και Αναζήτηση:** Η εφαρμογή πρέπει να προβάλλει τις πανεπιστημιακές ανακοινώσεις σε χρονολογική σειρά και να υποστηρίζει την άμεση αναζήτηση συγκεκριμένου περιεχομένου ή τίτλων.
- **Φιλτράρισμα και Κατηγοριοποίηση:** Πρέπει να παρέχεται η δυνατότητα φιλτραρίσματος των ανακοινώσεων βάσει θεματικών κατηγοριών ή ανά συγγραφέα, ώστε ο φοιτητής να απομονώνει την πληροφορία που τον αφορά.
- **Ειδοποιήσεις (Push Notifications):** Ο χρήστης πρέπει να μπορεί να εγγραφεται σε συγκεκριμένες κατηγορίες, λαμβάνοντας στοχευμένες ειδοποιήσεις στο παρασκήνιο όταν δημοσιεύεται νέο υλικό.
- **Διαχείριση Συνημμένων Αρχείων:** Απαιτείται η δυνατότητα προβολής και λήψης εγγράφων, όπως αρχεία PDF, που επισυνάπτονται στις ανακοινώσεις.
- **Ταυτοποίηση Χρηστών (Authentication):** Οι φοιτητές πρέπει να μπορούν να συνδεθούν για να αποκτούν πρόσβαση σε ιδιωτικές ανακοινώσεις που δεν είναι δημόσια ορατές.

1.4.2 Μη Λειτουργικές Απαιτήσεις

Οι μη λειτουργικές απαιτήσεις καθορίζουν τα ποιοτικά χαρακτηριστικά του συστήματος και την εμπειρία χρήσης:

- **Offline-First:** Η εφαρμογή οφείλει να είναι λειτουργική ακόμη και σε συνθήκες απουσίας ή αστάθειας δικτύου.
- **Υψηλή Απόδοση και Διαχείριση Πόρων:** Η φόρτωση μεγάλου όγκου δεδομένων πρέπει να γίνεται σταδιακά. Παράλληλα, ο χρόνος εκκίνησης (startup time) της εφαρμογής πρέπει να είναι ελαχιστοποιημένος.

- **Εργονομική Διεπαφή:** Το περιβάλλον χρήστη πρέπει να είναι σύγχρονο, πλήρως προσαρμοσμένο για οθόνες αφής και να προσφέρει ομαλή πλοήγηση χωρίς διακοπές.
- **Επεκτασιμότητα και Συντηρησιμότητα:** Η εσωτερική δομή του κώδικα πρέπει να διευκολύνει τη μελλοντική προσθήκη νέων λειτουργιών, τη διαχείριση σφαλμάτων και τη διεξαγωγή αυτοματοποιημένων ελέγχων.

1.4.3 Περιορισμοί Συστήματος και Backend

Οι τεχνικοί περιορισμοί που υπαγορεύονται από την υπάρχουσα υποδομή του ιδρύματος περιλαμβάνουν:

- **Συμβατότητα με το Υφιστάμενο Σύστημα:** Η εφαρμογή δεν θα αντικαταστήσει το υπάρχον σύστημα διαχείρισης ανακοινώσεων του πανεπιστημίου, αλλά θα λειτουργεί ως πελάτης που προσαρμόζεται σε αυτό.
- **Επικοινωνία μέσω REST API:** Η άντληση των δεδομένων, ο συγχρονισμός και οι λειτουργίες ταυτοποίησης πρέπει να πραγματοποιούνται αποκλειστικά μέσω κλήσεων πρωτοκόλλου HTTP καταναλώνοντας δεδομένα σε μορφή JSON.

1.5 Μεθοδολογία Υλοποίησης

Η ανάπτυξη της εφαρμογής πραγματοποιήθηκε με τη γλώσσα προγραμματισμού Kotlin, η οποία αποτελεί την επίσημη και προτεινόμενη επιλογή της Google για το οικοσύστημα Android [8]. Η διεπαφή χρήστη υλοποιήθηκε αποκλειστικά με το Jetpack Compose, το σύγχρονο εργαλείο (toolkit) της Google που επιτρέπει τον σχεδιασμό της εμφάνισης δηλωτικά μέσω κώδικα, καταργώντας την ανάγκη για ξεχωριστά αρχεία XML.

Η αρχιτεκτονική της εφαρμογής ακολουθεί το πρότυπο MVVM σε συνδυασμό με τις αρχές του Clean Architecture [9]. Αυτές οι σχεδιαστικές προσεγγίσεις εξασφαλίζουν τον σαφή διαχωρισμό ανάμεσα στο επίπεδο παρουσίασης (UI Layer), το επίπεδο επιχειρησιακής λογικής (Domain Layer) και το επίπεδο δεδομένων (Data Layer).

Για τη διαχείριση των δεδομένων αξιοποιήθηκαν οι βιβλιοθήκες Room (για την τοπική βάση δεδομένων) και Paging 3 (για τη σταδιακή φόρτωση μεγάλου όγκου δεδομένων). Παράλληλα, η βιβλιοθήκη WorkManager ανέλαβε την εκτέλεση εργασιών στο παρασκήνιο (π.χ. περιοδικός έλεγχος για νέες ανακοινώσεις), εγγυώμενη την ολοκλήρωσή τους ακόμη και μετά από επανεκκίνηση της συσκευής. Η διαχείριση των εξαρτήσεων (Dependency Injection) υλοποιήθηκε αυτοματοποιημένα μέσω του Hilt [10].

Ο κώδικας οργανώθηκε σε πολυαρθρωτή (multi-module) δομή, διαχωρίζοντας την εφαρμογή σε ανεξάρτητα υποσυστήματα ανάλογα με τη λειτουργικότητά τους. Η προσέγγιση αυτή επιτρέπει ταχύτερη μεταγλώττιση, σαφή διαχείριση εξαρτήσεων και ευκολότερη μελλοντική επέκταση [11].

Τέλος, εφαρμόστηκαν τεχνικές βελτιστοποίησης της απόδοσης μέσω Baseline Profiles και Startup Profiles ειδικών αρχείων τα οποία «καθοδηγούν» τον μεταγλωττιστή για το πως να βελτιστοποιήσει το τελικό εκτελέσιμο αρχείο και το λειτουργικό σύστημα ώστε να μεταγλωττίσει εκ των προτέρων (AOT compilation) τα πιο κρίσιμα τμήματα του κώδικα. Με τον τρόπο αυτό

μειώνεται αισθητά ο χρόνος εκκίνησης της εφαρμογής, γεγονός που επιβεβαιώθηκε μέσω του Macrobenchmark framework [12].

1.6 Προαπαιτούμενες Γνώσεις

Η παρούσα εργασία απευθύνεται σε αναγνώστες που διαθέτουν βασική εξοικείωση με τις αρχές του αντικειμενοστραφούς προγραμματισμού και τα θεμέλια της μηχανικής λογισμικού. Δεν απαιτείται προηγούμενη εμπειρία στην πλατφόρμα Android ή στη γλώσσα Kotlin. Το Κεφάλαιο 2 παρουσιάζει αναλυτικά κάθε τεχνολογία που χρησιμοποιήθηκε, επεξηγώντας τόσο τον τρόπο λειτουργίας της όσο και τη σκοπιμότητα της επιλογής της. Η γνώση βασικών εννοιών (όπως βάσεις δεδομένων, πρωτόκολλο HTTP και αρχιτεκτονικά πρότυπα) θα διευκολύνει την κατανόηση, χωρίς όμως να είναι απολύτως απαραίτητη. Για περαιτέρω εμβάθυνση στην πλατφόρμα Android, ο αναγνώστης παραπέμπεται στην επίσημη τεκμηρίωση της Google [13].

1.7 Δομή της Εργασίας

Η παρούσα διπλωματική εργασία οργανώνεται σε εννέα κεφάλαια.

Στο **πρώτο κεφάλαιο** (το παρόν) περιγράφεται το πρόβλημα, το κίνητρο, οι στόχοι, οι απαιτήσεις και οι προαπαιτούμενες γνώσεις, απαντώντας επί της ουσίας στο «γιατί» της εργασίας.

Στο **δεύτερο κεφάλαιο** παρουσιάζονται αναλυτικά οι τεχνολογίες που χρησιμοποιήθηκαν και τεκμηριώνονται οι λόγοι επιλογής τους.

Στο **τρίτο κεφάλαιο** αναλύεται η γενική αρχιτεκτονική του συστήματος: η τοπολογία, οι αρχές του Clean Architecture, η multi-module δομή και η διαχείριση εξαρτήσεων (Dependency Injection).

Στο **τέταρτο κεφάλαιο** εμβαθύνουμε στο *Επίπεδο Δεδομένων (Data Layer)*: εξετάζονται τα μοντέλα δεδομένων, ο client δικτύου, το Room, το DataStore καθώς και ο μηχανισμός λειτουργίας εκτός σύνδεσης (offline-first) με τη χρήση Paging 3 και RemoteMediator.

Στο **πέμπτο κεφάλαιο** αναλύεται το *Επίπεδο Επιχειρησιακής Λογικής (Domain Layer)*: τα Repository interfaces, τα Use Cases και τα domain models που συνθέτουν τον πυρήνα της εφαρμογής.

Στο **έκτο κεφάλαιο** παρουσιάζεται το *Επίπεδο Παρουσίασης (UI Layer)*: τα ViewModels, η ροή δεδομένων StateFlow/UDF, το Jetpack Compose, η πλοήγηση (Navigation) και το σχεδιαστικό πρότυπο Material 3.

Στο **έβδομο κεφάλαιο** αναλύονται οι οριζόντιες λειτουργίες που διαπερνούν τα επιμέρους επίπεδα: ο εναλλακτικός μηχανισμός ειδοποιήσεων μέσω WorkManager, οι στρατηγικές βελτιστοποίησης και η στρατηγική δοκιμών (unit testing).

Στο **όγδοο κεφάλαιο** πραγματοποιείται η αξιολόγηση της εφαρμογής, μέσα από λειτουργική επαλήθευση, δοκιμές απόδοσης (benchmarks) και ανάλυση της κατανάλωσης πόρων.

Τέλος, στο **ένατο κεφάλαιο** συνοψίζονται τα τελικά συμπεράσματα, αναδεικνύεται η συνεισφορά του έργου, αναγνωρίζονται οι περιορισμοί του και προτείνονται κατευθύνσεις για μελλοντική επέκταση.

Κεφάλαιο 2

Τεχνολογικό Υπόβαθρο

2.1 Εισαγωγή

Το παρόν κεφάλαιο παρουσιάζει το τεχνολογικό υπόβαθρο που διαμορφώνει την αρχιτεκτονική και τις υλοποιητικές επιλογές της εφαρμογής. Ξεκινά με τη θεμελιακή στρατηγική επιλογή μεταξύ native και Cross-Platform ανάπτυξης (Ενότητα 2.2), στη συνέχεια εξετάζει τη γλώσσα Kotlin και το ασύγχρονο μοντέλο των Coroutines (Ενότητα 2.3), παρουσιάζει την αρχιτεκτονική δομή MVVM και Clean Architecture (Ενότητα 2.4), εισάγει το Jetpack Compose ως σύγχρονο πλαίσιο διεπαφής (Ενότητα 2.5), και τέλος περιγράφει τις τεχνολογίες αποθήκευσης και δικτύου που συνθέτουν το Data Layer (Ενότητα 2.6).

2.2 Επιλογή Πλατφόρμας Ανάπτυξης

Κατά τη σχεδίαση κινητής εφαρμογής, το πρώτο στρατηγικό ερώτημα είναι: μία κοινή βάση κώδικα για Android και iOS, ή ξεχωριστές native εφαρμογές;

Οι Cross-Platform λύσεις, όπως η Flutter [14] και η React Native [15], επιτρέπουν συγγραφή ενιαίου κώδικα για αμφότερες τις πλατφόρμες. Το κύριο πλεονέκτημά τους είναι η οικονομία ανάπτυξης. Το μειονέκτημα έγκειται στην παρεμβολή ενός επιπλέον επιπέδου λογισμικού (bridge) μεταξύ εφαρμογής και λειτουργικού συστήματος, που εισάγει καθυστέρηση και δυσκολίες πρόσβασης σε εξειδικευμένα APIs.

Για την παρούσα εφαρμογή επιλέχθηκε η native ανάπτυξη. Ο λόγος δεν ήταν μόνο η απόδοση: ο στόχος ήταν η εις βάθος επίδειξη βέλτιστων πρακτικών του Android (π.χ. WorkManager [6], Paging 3 [5] με RemoteMediator [16]), λειτουργίες για τις οποίες τα Cross-Platform εργαλεία απαιτούν wrappers ή απλώς δεν τις υποστηρίζουν άμεσα. Ο Πίνακας 2.1 συνοψίζει τη σύγκριση.

Κριτήριο	Native	Cross-Platform
Απόδοση	Μέγιστη ταχύτητα, με άμεση επικοινωνία με το σύστημα.	Overhead λόγω bridge ή της μη-χανής γραφικών.
Εμπειρία Χρήστη	Εναρμονίζεται με την εμφάνιση της συσκευής.	Απόκλιση από τα πρότυπα κάθε πλατφόρμας.
Πρόσβαση σε APIs	Άμεση υποστήριξη νέων δυνατοτήτων.	Εξαρτάται από εξωτερικά plugins ή wrappers.
Οικοσύστημα	Επίσημα, εξειδικευμένα εργαλεία από την Google.	Εργαλεία χαμηλότερης ποιότητας ή ελλιπούς συντήρησης.
Βάση Κώδικα	Γράφεται αποκλειστικά και μόνο για το Android.	Μία κοινή βάση κώδικα για Android και iOS.
Χρόνος Ανάπτυξης	Πιο χρονοβόρο.	Ταχύτερη κυκλοφορία.

Πίνακας 2.1: Σύγκριση native και Cross-Platform ανάπτυξης

Έχοντας επιλέξει native ανάπτυξη, η επόμενη ερώτηση είναι ποια γλώσσα και ποιο ασύγχρονο μοντέλο θα χρησιμοποιηθεί ερωτήματα αλληλένδετα, όπως εξηγεί η επόμενη ενότητα.

2.3 Γλώσσα και Ασύγχρονος Προγραμματισμός

2.3.1 Kotlin: Η Επίσημη Γλώσσα Android

Η Kotlin είναι στατικά τυποποιημένη γλώσσα της JetBrains που έγινε η επίσημη γλώσσα Android από την Google το 2017 [8]. Εκτελείται στο JVM, εξασφαλίζοντας πλήρη δια-λειτουργικότητα με υπάρχοντα Java APIs.

Το πιο σημαντικό χαρακτηριστικό της Kotlin για αξιοπιστία εφαρμογής είναι το Null Safety [17]. Στη Java, οποιαδήποτε μεταβλητή μπορεί να γίνει `null` και τα `NullPointerException` εντοπίζονται μόνο κατά την εκτέλεση, προκαλώντας crash της εφαρμογής. Η Kotlin το επιλύει αυτό σε επίπεδο γλώσσας: ο μεταγλωττιστής διακρίνει ρητά μεταξύ nullable (`String?`) και μη-nullable (`String`) τύπων. Εάν χρησιμοποιηθεί μεταβλητή τύπου nullable χωρίς τον απαραίτητο έλεγχο, ο κώδικας δε μεταγλωττίζεται. Έτσι, ολόκληρη η συγκεκριμένη κατηγορία σφαλμάτων εξαλείφεται πριν η εφαρμογή φτάσει στον χρήστη.

Η επιλογή γλώσσας δεν είναι ανεξάρτητη από το ασύγχρονο μοντέλο: η Kotlin προσφέρει στην ίδια τη γλώσσα τον μηχανισμό που κάνει τον ασύγχρονο κώδικα αναγνώσιμο, όπως εξηγείται αμέσως παρακάτω.

2.3.2 Από Threads σε Coroutines: Εξέλιξη του Ασύγχρονου Μοντέλου

Κάθε κινητή εφαρμογή εκτελεί χρονοβόρες εργασίες: κλήσεις δικτύου, εγγραφές σε βάσεις δεδομένων, επεξεργασία αρχείων. Αν αυτές εκτελούνταν στο κύριο thread εκτέλεσης (αυτό που χειρίζεται τη διεπαφή χρήστη) η εφαρμογή θα «πάγωνε» και το Android θα την τερμάτιζε με το μήνυμα «*Application Not Responding*» (ANR). Η αντιμετώπιση αυτού του προβλήματος εξελίχθηκε σε τρία στάδια:

- **Threads.** Ξεχωριστά νήματα εκτέλεσης που τρέχουν παράλληλα με το κύριο thread. Η διαχείρισή τους είναι όμως επικίνδυνη: δύο threads δεν μπορούν να γράψουν στα ίδια

δεδομένα ταυτόχρονα χωρίς συγχρονισμό, με κίνδυνο **deadlocks** αδιέξοδα όπου δύο threads αναμένουν αμοιβαία την αποδέσμευση πόρων και **race conditions** ταυτόχρονες εγγραφές που οδηγούν σε απρόβλεπτα αποτελέσματα.

- **Callbacks.** Ο προγραμματιστής καλεί μια ασύγχρονη λειτουργία και ορίζει μια συνάρτηση (callback) η οποία θα εκτελεστεί με την ολοκλήρωση της ασύγχρονης διεργασίας. Αυτό λύνει το πρόβλημα blocking, αλλά όταν πολλές ασύγχρονες λειτουργίες εξαρτώνται η μία από την άλλη, ο κώδικας καταλήγει σε **callback hell**: βαθιά φωλιασμένο, δυσανάγνωστο σύνολο συναρτήσεων.
- **Coroutines.** Τα Coroutines [18] λύνουν και τα δύο παραπάνω προβλήματα. Μια coroutine είναι «ελαφριά» μονάδα εκτέλεσης που μπορεί να *αναστέλλεται* (suspend) όταν αναμένει μια εξωτερική απόκριση, και να *επαναλαμβάνεται* (resume) αυτόματα όταν αυτή φτάσει χωρίς να δεσμεύει thread στο ενδιαμέσο. Σημαντικό είναι ότι οι coroutines *δεν αντικαθιστούν* τα threads: εκτελούνται πάνω σε μια διαχειριζόμενη δεξαμενή threads (thread pool) μέσω των Dispatchers, αλλά χιλιάδες coroutines μπορούν να μοιράζονται ελάχιστα threads, καθώς δεν τα καταλαμβάνουν κατά τη διάρκεια αναστολής.

Το κρίσιμο πλεονέκτημα από πλευράς αναγνωσιμότητας είναι ότι ο κώδικας γράφεται σε *διαδοχικό* ύφος, σαν να ήταν σύγχρονος, ενώ εκτελείται ασύγχρονα. Το παράδειγμα που ακολουθεί εκτελεί τις ίδιες τρεις ασύγχρονες λειτουργίες και με τις δύο προσεγγίσεις:

```
// Callbacks  δυσανάγνωστο "callback hell"
fetchUser { user ->
    fetchAnnouncements(user.id) { announcements ->
        saveToDatabase(announcements) { updateUi() }
    }
}

// Coroutines ευανάγνωστο, ίδιο αποτέλεσμα
val user          = fetchUser()
val announcements = fetchAnnouncements(user.id)
saveToDatabase(announcements)
updateUi()
```

Η κατανομή εργασιών σε threads γίνεται μέσω Dispatchers: **Dispatchers.IO** για λειτουργίες δικτύου και δίσκου, **Dispatchers.Main** για ενημερώσεις του UI, και **Dispatchers.Default** για υπολογιστικά έντονες εργασίες.

2.3.3 Flow και StateFlow: Αντιδραστική Ροή Δεδομένων

Οι κορουτίνες (coroutines) επιλύουν το ζήτημα της εφάπαξ ασύγχρονης εκτέλεσης. Ωστόσο, προκύπτει μια συμπληρωματική ανάγκη: η συνεχής ενημέρωση της διεπαφής χρήστη (UI) σε κάθε αλλαγή των δεδομένων. Εκεί μπορούμε να χρησιμοποιήσουμε τα **Flows** [19].

Ένα Flow είναι ασύγχρονη ροή τιμών που εκπέμπει νέες τιμές με την πάροδο του χρόνου σαν ζωντανή σύνδεση σε πηγή δεδομένων. Αντί για την περιοδική άντληση της τρέχουσας

τιμής (polling), η εφαρμογή εγγράφεται (subscribes) ως παρατηρητής για να λαμβάνει κάθε νέα ενημέρωση. Αυτό το μοτίβο ονομάζεται αντιδραστικός προγραμματισμός (reactive programming). Η εξειδικευμένη παραλλαγή **StateFlow** [20] κρατάει πάντα μία «τρέχουσα τιμή» και την εκπέμπει αμέσως σε κάθε νέο παρατηρητή. Στην εφαρμογή, το **ViewModel** διατηρεί το **StateFlow** ενώ το **UI** «εγγράφεται» σε αυτό: κάθε φορά που αλλάζει η κατάσταση, το **UI** ενημερώνεται αυτόματα χωρίς κανένα ρητό κώδικα «ανανέωσε την οθόνη». Η πρακτική σημασία αυτού και ο τρόπος που τα **Flows** συνδέουν τη βάση δεδομένων με το **UI** γίνεται πλήρως εμφανής στην Ενότητα 2.6, όπου το **Room** επιστρέφει δεδομένα απευθείας ως **Flow**. Με **Flow** και **Coroutines** ως θεμέλιο, η επόμενη ερώτηση είναι πώς δομείται ο κώδικας γύρω τους αυτό το ερώτημα απαντά η αρχιτεκτονική.

2.4 Αρχιτεκτονική Εφαρμογής

Τα ζητήματα κύκλου ζωής, δοκιμασιμότητας και κλιμακωσιμότητας δεν επιλύονται από μία και μόνη τεχνολογία, αλλά από μια σειρά συμπληρωματικών αρχιτεκτονικών επιλογών που παρουσιάζονται εδώ ως ενιαίο σύνολο.

2.4.1 Διαχείριση Κύκλου Ζωής και ViewModels

Στα πρώτα χρόνια του **Android**, βασικό δομικό στοιχείο ήταν η **Activity**: μία κλάση ανά οθόνη, που διαχειριζόταν τα πάντα διάταξη, κλήσεις δικτύου, αποθήκευση, αλληλεπίδραση χρήστη. Αυτό το αντιπρότυπο έγινε γνωστό ως «**God Activity**». Ο κώδικας ήταν δυσανάγνωστος και δυσκολότερα ελεγχόμενος, και υπέφερε από ένα συγκεκριμένο πρόβλημα: κάθε φορά που ο χρήστης περιέστρεφε τη συσκευή, το **Android** κατέστρεφε και ξανάδημιουργούσε την **Activity**, με αποτέλεσμα την απώλεια της τρέχουσας κατάστασης.

Τα **ViewModels** [21] λύνουν αυτό το πρόβλημα. Κάθε **ViewModel** είναι ένα αντικείμενο σχεδιασμένο να επιβιώνει από αλλαγές διαμόρφωσης (configuration changes): όταν το **Activity** καταστρέφεται και ξαναδημιουργείται, τα **ViewModels** παραμένουν ζωντανά με τα δεδομένα άθικτα. Καταστρέφεται μόνο όταν ο χρήστης αποχωρήσει οριστικά από την οθόνη.

2.4.2 MVVM

Το **MVVM** [3] οργανώνει τον κώδικα σε τρεις ρόλους γύρω από το **ViewModel**:

- **Model.** Τα δεδομένα και η λογική ανάκτησής τους.
- **View.** Η διεπαφή χρήστη. Δεν περιέχει επιχειρησιακή λογική, παρά μόνο προβάλλει τα δεδομένα που του παρέχει το **ViewModel**.
- **ViewModel.** Μετασχηματίζει δεδομένα σε μορφή κατάλληλη για εμφάνιση από το **View** και διαχειρίζεται τα events του χρήστη.

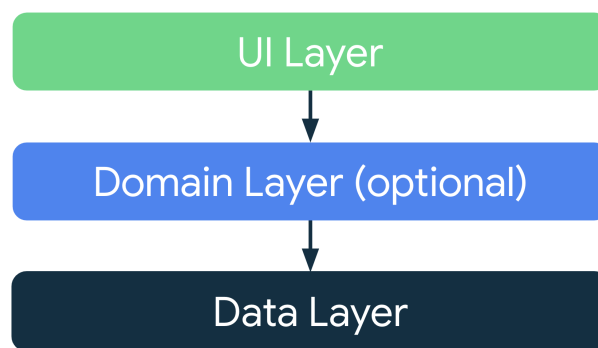
Αυτός ο διαχωρισμός αποφέρει τέσσερα πρακτικά οφέλη. Πρώτον, το **ViewModel** **επιβιώνει από configuration changes**: παραμένει ζωντανό κατά την περιστροφή συσκευής ή οποιαδήποτε άλλη αλλαγή διαμόρφωσης, διατηρώντας την κατάσταση χωρίς επανάληψη κλήσεων δικτύου.

Δεύτερον, ευνοεί τη **δοκιμασιμότητα**: ένα ViewModel δοκιμάζεται αυτόνομα χωρίς να τρέχει εφαρμογή, καθώς δεν εξαρτάται από το Android SDK αφού αρκεί η παροχή εικονικών δεδομένων (mocks) για να ελεγχθεί η παραγωγή του επιθυμητού `UiState`. Τρίτον, απλοποιεί την **προεπισκόπηση Composables**: επειδή το View δεν περιέχει λογική, αρκεί να του περαστεί ένα στατικό `UiState` για να αποδοθεί μέσω του `@Preview` του Android Studio, χωρίς πραγματικά δεδομένα ή σύνδεση δικτύου. Τέλος, βελτιώνει την **αναγνωσιμότητα και συντηρησιμότητα**: κάθε αρχείο έχει έναν και μόνο σαφή ρόλο, οπότε ένας νέος συνεργάτης μπορεί να κατανοήσει τη λογική μιας οθόνης διαβάζοντας μόνο το ViewModel, χωρίς να πλοηγηθεί σε κώδικα διεπαφής.

2.4.3 Clean Architecture

Το MVVM ορίζει τη σχέση View–ViewModel–Model, αλλά δεν προδιαγράφει πώς οργανώνεται το Model. Εδώ εφαρμόζονται οι αρχές του **Clean Architecture** [9], ο κώδικας οργανώνεται σε ομόκεντρα επίπεδα, και κάθε επίπεδο εξαρτάται αποκλειστικά από τα κατώτερα του, ως εξής:

- **UI Layer**: Τα Composables και τα ViewModels [22]. Εκθέτει `UiState` και ανταποκρίνεται σε events χρήστη.
- **Domain Layer**: Τα Use Cases κλάσεις που εκτελούν μία επιχειρησιακή λειτουργία (π.χ. `GetAnnouncementsUseCase`). Δε διαθέτει καμία εξάρτηση από το Android SDK ή το δίκτυο αντίθετα, χρησιμοποιεί και δηλώνει αποκλειστικά αφηρημένες διεπαφές (interfaces) [23]. Αυτό το επίπεδο είναι κρίσιμο όταν η επιχειρησιακή λογική επαναχρησιμοποιείται από πολλά ViewModels.
- **Data Layer**: Τα Repositories, οι βάσεις δεδομένων, οι κλήσεις δικτύου [24]. Καθορίζει τον τρόπο ανάκτησης των δεδομένων, λειτουργεί όμως ανεξάρτητα από τον τελικό σκοπό χρήσης τους.



Σχήμα 2.1: Τα τρία επίπεδα του Clean Architecture και οι κατευθύνσεις εξάρτησης [13].

Η πρακτική συνέπεια αυτής της δομής γίνεται εμφανής όταν αλλάξει κάτι στο Data Layer π.χ. αντικατάσταση βάσης δεδομένων ή αλλαγή API. Επειδή το Domain και το UI Layer εξαρτώνται από αφηρημένες διεπαφές και όχι από συγκεκριμένες υλοποιήσεις, παραμένουν αμετάβλητα. Αυτή είναι η αρχή **Dependency Inversion (DIP)** από το σύνολο αρχών SOLID. Συμπληρωματικά,

η αρχή **Single Responsibility (SRP)** εξασφαλίζει ότι κάθε Use Case έχει μία και μόνη ευθύνη για παράδειγμα, το `GetAnnouncementsUseCase` αντλεί ανακοινώσεις και τίποτε άλλο κάτι που απλοποιεί τόσο τη συντήρηση όσο και τον έλεγχο του κώδικα.

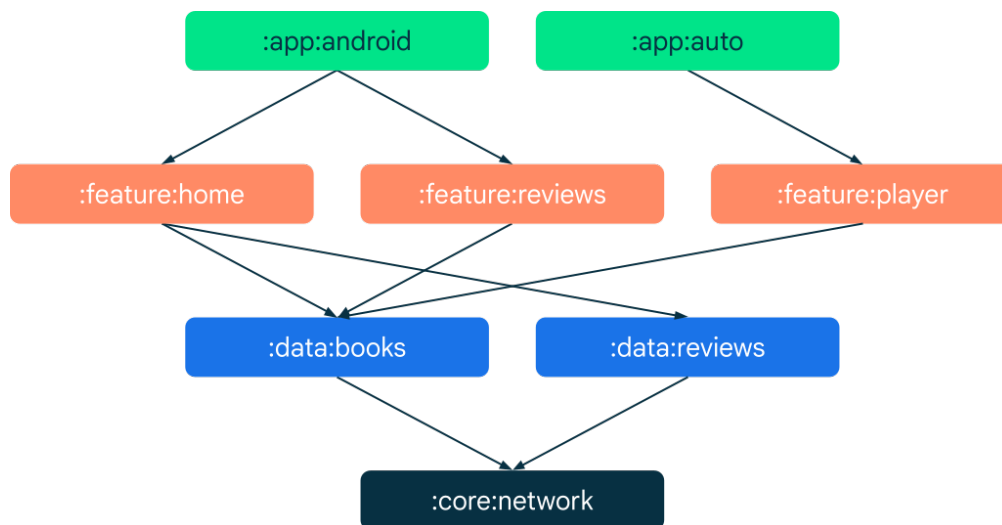
2.4.4 Πολυαρθρωτή Δομή

Εφόσον το Clean Architecture ορίζει τα επίπεδα λογικά, η πολυαρθρωτή δομή [11] (multi-module) τα εφαρμόζει δομικά: ο κώδικας χωρίζεται σε ανεξάρτητα modules αντί να βρίσκεται σε ένα ενιαίο μονόλιθο.

Το κύριο πρακτικό όφελος είναι η ταχύτητα ανάπτυξης. Κάθε αλλαγή σε ένα module απαιτεί μεταγλώττιση μόνο αυτού και των modules που εξαρτώνται από αυτό, όχι ολόκληρης της εφαρμογής. Σε μεγάλα projects, η δυνατότητα αυτή γνωστή ως *incremental compilation* μειώνει δραστικά τον χρόνο αναμονής κατά την ανάπτυξη.

Εξίσου σημαντική είναι η *επιβολή* σωστού διαχωρισμού ευθυνών σε επίπεδο filesystem: ένα feature module απλά δεν μπορεί να έχει πρόσβαση στον κώδικα άλλου feature module, καθώς δεν το εισάγει ως εξάρτηση. Αντί για ανεπίσημες συμβάσεις, τα όρια επιβάλλονται από το ίδιο το build system. Επιπλέον, τα core modules όπως `core:network` ή `core:database` μπορούν να επαναχρησιμοποιηθούν από πολλά feature modules χωρίς αντιγραφή κώδικα.

Τέλος, για να αποφευχθεί ο πολλαπλασιασμός ίδιων ρυθμίσεων Gradle σε κάθε module, χρησιμοποιούνται **convention plugins**: επαναχρησιμοποιήσιμα τεμάχια διαμόρφωσης που εφαρμόζονται με μία γραμμή σε οποιοδήποτε module. Η πρακτική εφαρμογή τους αναλύεται στο Κεφάλαιο 3.



Σχήμα 2.2: Ενδεικτικό γράφημα εξαρτήσεων modules [11].

2.4.5 Dependency Injection

Το Clean Architecture απαιτεί τα επίπεδα να εξαρτώνται από interfaces, όμως κατά την εκτέλεση (runtime) πρέπει να παρέχονται οι πραγματικές υλοποιήσεις. Αυτό το κενό καλύπτει το **Dependency Injection (DI)**: αντί ένα αντικείμενο να δημιουργεί τις εξαρτήσεις του, του παρέχονται εξωτερικά, συνήθως μέσω **constructor injection**. Έτσι, διαχωρίζεται η δημιουργία των αντικειμένων από τη χρήση τους. Το βασικότερο πρακτικό όφελος είναι η άμεση διευκόλυνση

του testing, καθώς κάθε εξάρτηση μπορεί εύκολα να αντικατασταθεί με ένα mock αντικείμενο στα unit tests, χωρίς να απαιτείται καμία αλλαγή στον κώδικα.

```
// Χωρίς DI το αντικείμενο δημιουργεί μόνο του τις εξαρτήσεις του
class AnnouncementsViewModel(
    private val repository: AnnouncementRepository =
        ↪ AnnouncementRepositoryImpl(
            localDataSource = AnnouncementLocalDataSourceImpl(
                database = AnnouncementDatabaseImpl()
            ),
            remoteDataSource = AnnouncementRemoteDataSourceImpl(
                client = AnnouncementRetrofitClientImpl()
            ),
        )
): ViewModel()

// Με DI οι εξαρτήσεις παρέχονται εξωτερικά μέσω constructor
@HiltViewModel
class AnnouncementsViewModel @Inject constructor(
    private val repository: AnnouncementRepository
): ViewModel()
```

Το Hilt είναι το επίσημα προτεινόμενο DI framework για Android [10]. Ο προγραμματιστής ορίζει σε *modules* ποια υλοποίηση αντιστοιχεί σε κάθε interface για παράδειγμα, ότι το `AnnouncementRepository` παρέχεται από το `AnnouncementRepositoryImpl` και το Hilt αναλαμβάνει αυτόματα τη δημιουργία και παροχή των σωστών αντικειμένων σε κάθε σημείο της εφαρμογής που τα χρειάζεται. Το Κεφάλαιο 3 παρουσιάζει την εφαρμογή αυτής της αρχής στη δομή του έργου, με συγκεκριμένα παραδείγματα από τα `NetworkModule` και `DatabaseModule`.

Με αρχιτεκτονική θεμελιωμένη και εξαρτήσεις διαχειρισμένες, το επόμενο ερώτημα είναι πώς χτίζεται η διεπαφή που «βλέπει» αυτή την αρχιτεκτονική.

2.5 Jetpack Compose

Πριν το Jetpack Compose, η διεπαφή Android οριζόταν σε αρχεία **XML**: ο κώδικας Kotlin φόρτωνε ένα XML layout, εντόπιζε στοιχεία με `findViewById` και τα ενημέρωνε χειροκίνητα μέσω callbacks. Αυτό είναι το **επιτακτικό (imperative)** μοντέλο ο προγραμματιστής περιγράφει πώς να αλλάξει το UI. Το μοντέλο αυτό ήταν επιρρεπές σε ασυνέπειες μεταξύ της κατάστασης στο `ViewModel` και της κατάστασης που εμφάνιζε το `View`, καθώς κάθε αλλαγή έπρεπε να μεταφραστεί χειροκίνητα σε κλήσεις ενημέρωσης στοιχείων.

Η παρούσα εφαρμογή χρησιμοποιεί **Jetpack Compose** [2], το οποίο εισάγει το **δηλωτικό (declarative)** μοντέλο: ο προγραμματιστής περιγράφει τι πρέπει να δείχνει η οθόνη δεδομένης μιας κατάστασης, και το framework ενημερώνει αποδοτικά μόνο ό,τι χρειαστεί. Η αντιστοίχιση είναι μαθηματική:

$$UI = f(state) \quad (2.1)$$

όπου *state* είναι το `UiState` που εκθέτει το `ViewModel`, και *f* είναι το σύνολο των `Composable` συναρτήσεων που το μετατρέπουν σε οπτική αναπαράσταση. Κάθε φορά που το *state* αλλάξει, το `Compose` επανεκτελεί αυτόματα μόνο τις συναρτήσεις που εξαρτώνται από αυτό μια διαδικασία γνωστή ως *recomposition*. Το παράδειγμα που ακολουθεί αντιπαραβάλλει τις δύο προσεγγίσεις για την εμφάνιση ενός μηνύματος φόρτωσης:

```
// Imperative χειροκίνητη ενημέρωση στοιχείου
loadingView.visibility = if (state.isLoading) View.VISIBLE else
    → View.GONE

// Declarative το UI περιγράφεται ως συνάρτηση της κατάστασης
@Composable
fun AnnouncementsScreen(state: AnnouncementsUiState) {
    if (state.isLoading) {
        CircularProgressIndicator()
    } else {
        AnnouncementsList(state.announcements)
    }
}
```

2.5.1 Composable Functions και Recomposition

Στο `Compose`, η διεπαφή χτίζεται από **Composable functions** Kotlin συναρτήσεις με το annotation `@Composable` που καλούν η μία την άλλη για να δημιουργήσουν σύνθετα layouts. Δεν υπάρχει ξεχωριστό αρχείο layout: δομή, εμφάνιση και λογική αλληλεπίδρασης βρίσκονται στον ίδιο κώδικα Kotlin, όπως φαίνεται στο παράδειγμα που ακολουθεί:

```
@Composable
fun AnnouncementsScreen(viewModel: AnnouncementsViewModel =
    → hiltViewModel()) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()

    if (state.isLoading) {
        CircularProgressIndicator()
    } else {
        AnnouncementsList(announcements = state.announcements)
    }
}

@Composable
fun AnnouncementsList(announcements: List<Announcement>) {
    LazyColumn {
        items(announcements) { announcement ->
```

```

        AnnouncementCard(announcement)
    }
}

```

Όταν αλλάζει η κατάσταση που παρατηρεί ένα Composable, το Compose εκτελεί **recomposition**: επανακαλεί αυτόματα τη συνάρτηση με τη νέα κατάσταση. Το Compose παρακολουθεί συγκεκριμένα αντικείμενα τύπου State στην περίπτωση αυτή το uiState που εκτίθεται από το ViewModel ως StateFlow και συλλέγεται μέσω collectAsStateWithLifecycle(). Σημαντικό είναι ότι το Compose δεν ξαναχτίζει ολόκληρη την οθόνη: εντοπίζει ακριβώς ποια Composables εξαρτώνται από την τιμή που άλλαξε και επανεκτελεί μόνο αυτά, διατηρώντας αμετάβλητα τα υπόλοιπα.

2.5.2 State-Hoisting και Unidirectional Data Flow (UDF)

Στο Compose, κάθε Composable function λαμβάνει δεδομένα ως παραμέτρους και επιστρέφει callbacks για αλλαγές δε διατηρεί δική της κατάσταση (stateless). Αυτή η αρχή ονομάζεται **State-Hoisting** και εφαρμόζεται άμεσα στον κώδικα:

```

// Stateless Composable λαμβάνει state και εκπέμπει events
@Composable
fun AnnouncementsContent(
    announcements: List<Announcement>,
    isLoading: Boolean,
    onRefresh: () -> Unit
) { ... }

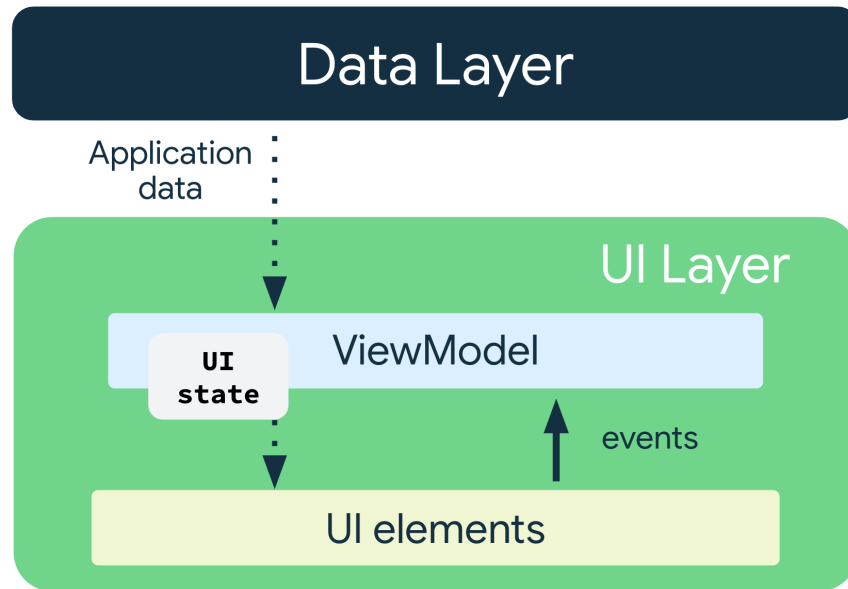
// To ViewModel κρατάει την κατάσταση και χειρίζεται τα events
@Composable
fun AnnouncementsScreen(viewModel: AnnouncementsViewModel =
    ↪ hiltViewModel()) {
    val state by viewModel.uiState.collectAsStateWithLifecycle()
    AnnouncementsContent(
        announcements = state.announcements,
        isLoading      = state.isLoading,
        onRefresh      = viewModel::refresh
    )
}

```

Το State-Hoisting σε συνδυασμό με τα ViewModels δημιουργεί το μοτίβο **Unidirectional Data Flow (UDF)** [25], που απεικονίζεται στο Σχήμα 2.3:

- **Η κατάσταση (state) ρέει προς τα κάτω:** Το ViewModel εκθέτει StateFlow τα Composables εγγράφονται και εμφανίζουν ό,τι λαμβάνουν.

- **Τα γεγονότα (events) ρέουν προς τα πάνω:** Ο χρήστης αλληλεπιδρά με ένα Composable αυτό ειδοποιεί το ViewModel μέσω callback εκείνο αποφασίζει την αλλαγή κατάστασης.



Σχήμα 2.3: Το μοτίβο UDF: η κατάσταση κατεβαίνει προς το UI, τα events ανεβαίνουν προς το ViewModel [22].

Η μονόδρομη ροή εξαλείφει bugs όπου δύο μέρη κρατούν ασύγχρονες «εκδοχές» της ίδιας κατάστασης. Επιπλέον, επειδή κάθε Composable είναι stateless, η δοκιμή του απλοποιείται σημαντικά: αρκεί να περαστούν διαφορετικές τιμές παραμέτρων για να επαληθευτεί η συμπεριφορά του.

2.5.3 Type-Safe Navigation με Navigation Compose

Η πλοήγηση μεταξύ των οθονών της εφαρμογής διαχειρίζεται μέσω του **Navigation Compose** [26] με τη χρήση **type-safe navigation**. Αντί για απλές συμβολοσειρές (strings), χρησιμοποιούνται κλάσεις της Kotlin ως προορισμοί (destinations).

Με αυτόν τον τρόπο, ο μεταγλωττιστής ελέγχει την ορθότητα κάθε μετάβασης κατά τη μεταγλώττιση τυχόν λανθασμένα ή ελλιπή ορίσματα αποτελούν compile error και όχι runtime crash κατά την εκτέλεση.

```
@Serializable
object AnnouncementsDestination
```

```
@Serializable
data class AnnouncementDetailDestination(val id: String)
```

```
// Πλοήγηση με type-safety
navController.navigate(AnnouncementDetailDestination(announcement.id))
```

2.6 Τεχνολογίες Αποθήκευσης και Δικτύου

Ενώ η γενική αρχιτεκτονική ορίζει το ποιες οντότητες διαβάζουν τα δεδομένα, το επίπεδο δεδομένων καθορίζει την πηγή («από πού») και τον τρόπο ανάκτησης τους («πώς»). Οι επιλογές εδώ διαμορφώνουν απευθείας την offline εμπειρία και την κατανάλωση μνήμης.

2.6.1 Επικοινωνία με το Backend: Retrofit και REST

Η εφαρμογή επικοινωνεί με το Backend μέσω **REST API**: αιτήματα HTTP (GET, POST κ.λπ.) με απαντήσεις σε μορφή **JSON**. Ένα τυπικό παράδειγμα απόκρισης για μια ανακοίνωση έχει την ακόλουθη μορφή:

```
{
  "data": {
    "id": 64826,
    "title": "Αλλαγή ώρας εξέτασης",
    "body": "Η εξέταση μεταφέρεται στις 14:00.",
    "created_at": "2025-08-26 10:34",
    "author": {
      "name": "Παπαδόπουλος .",
      "id": 156
    }
  }
}
```

Η βιβλιοθήκη **Retrofit** [27] αναλαμβάνει την επικοινωνία με το δίκτυο, ενώ το **kotlinx.serialization** μετατρέπει αυτόματα τις JSON απαντήσεις σε Kotlin αντικείμενα (deserialization). Ο προγραμματιστής ορίζει ένα interface με τα διαθέσιμα αιτήματα και το Retrofit δημιουργεί αυτόματα την υλοποίηση:

```
interface AnnouncementApiService {
    @GET("announcements")
    suspend fun getAnnouncements(
        @Query("page") page: Int,
        @Query("size") size: Int
    ): PagedResponse<AnnouncementDto>

    @GET("announcements/{id}")
    suspend fun getAnnouncementById(@Path("id") id: Int):
        ↳ AnnouncementDto

    @GET("tags")
    suspend fun getTags(): List<TagDto>

    @POST("subscriptions/{tagId}")
```

```

suspend fun subscribeToTag(@Path("tagId") tagId: Int)
}

```

Τα annotations (@GET, @POST, @Query, @Path) ορίζουν τον τύπο αιτήματος και τις παραμέτρους URL. Οι τύποι επιστροφής χρησιμοποιούν το επίθεμα Dto (Data Transfer Object) για να διαχωριστούν από τα εσωτερικά μοντέλα της εφαρμογής. Η λέξη-κλειδί suspend δηλώνει εκτέλεση μέσω Coroutines (βλ. Ενότητα 2.3.2): τα δεδομένα φτάνουν στο Repository χωρίς ποτέ να δεσμεύουν το κύριο thread.

2.6.2 Τοπική Αποθήκευση: Room και SQLite

Τα δεδομένα από το δίκτυο δεν παρουσιάζονται απευθείας στο UI αποθηκεύονται πρώτα τοπικά μέσω Room [4], ενός επιπέδου αφαίρεσης πάνω από την ενσωματωμένη **SQLite**. Χρησιμοποιώντας SQLite απευθείας, ο προγραμματιστής γράφει SQL ερωτήματα ως strings και διαχειρίζεται μόνος του τη μετατροπή μεταξύ Kotlin αντικειμένων και γραμμών πίνακα μια προσέγγιση επιρρεπής σε σφάλματα που εντοπίζονται μόνο κατά την εκτέλεση. Το Room αντιμετωπίζει και τα δύο: ελέγχει τα SQL ερωτήματα κατά τη μεταγλώττιση και αναλαμβάνει αυτόματα τη μετατροπή αντικειμένων.

Η δομή του Room αποτελείται από τρία στοιχεία, όπως φαίνεται στο παράδειγμα:

```

// 1. Entity αντιστοιχεί σε πίνακα της βάσης
@Entity(tableName = "announcements")
data class AnnouncementEntity(
    @PrimaryKey val id: Int,
    val title: String,
    val date: String,
)

// 2. Dao ορίζει τα διαθέσιμα ερωτήματα
@Dao
interface AnnouncementDao {
    @Query("SELECT * FROM announcements ORDER BY date DESC")
    fun getAll(): Flow<List<AnnouncementEntity>>

    @Upsert
    suspend fun upsertAll(announcements: List<AnnouncementEntity>)
}

// 3. Database συνδέει entities και DAOs
@Database(entities = [AnnouncementEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun announcementDao(): AnnouncementDao
}

```

Το κρίσιμο πλεονέκτημα της επιστροφής Flow από το DAO είναι η άμεση ενσωμάτωση με το UDF μοτίβο (βλ. Ενότητα 2.5.2): κάθε φορά που αλλάζουν δεδομένα στη βάση, το Flow

εκπέμπει νέα τιμή, το ViewModel ενημερώνει το `UiState`, και το UI ανασυντάσσεται αυτόματα χωρίς καμία χειροκίνητη ενημέρωση.

2.6.3 Offline-First: Room ως Μοναδική Πηγή Αλήθειας

Με Retrofit και Room διαθέσιμα, προκύπτουν δύο διαφορετικές στρατηγικές ανάκτησης δεδομένων. Στην παραδοσιακή προσέγγιση, το UI ζητά δεδομένα από το δίκτυο και χωρίς σύνδεση εμφανίζεται κενή οθόνη ή μήνυμα σφάλματος. Η **offline-first** προσέγγιση αντιστρέφει αυτή τη λογική: η βάση δεδομένων Room ορίζεται ως **Μοναδική Πηγή Αλήθειας (Single Source of Truth SSoT)**, και το δίκτυο χρησιμεύει αποκλειστικά για να την ενημερώνει στο παρασκήνιο. Η ροή δεδομένων είναι πάντα μονόδρομη: Δίκτυο → Room → UI. Το Repository υλοποιεί αυτή τη λογική:

```

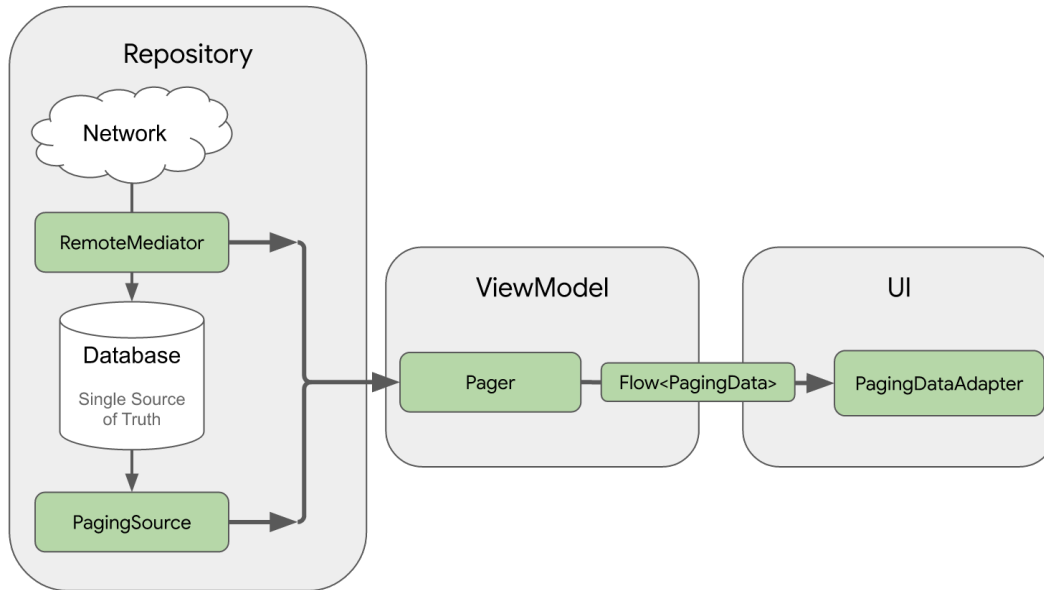
override fun getAnnouncementWithId(id: Int): Flow<Announcement?> {
    return announcementLocalDataSource.getAnnouncementWithId(id)
        .map { it?.toAnnouncement() }
}
override suspend fun refreshAnnouncementWithId(id: Int) {
    database.withTransaction {
        val announcement =
            ↪ announcementRemoteDataSource.fetchAnnouncementWithId(id)
        announcementDao.upsertAnnouncement(announcement)
    }
}

```

Αποτέλεσμα: η εφαρμογή φορτώνει αμέσως τα αποθηκευμένα δεδομένα ακόμα και χωρίς σύνδεση, και ανανεώνεται αθόρυβα όταν φτάσουν νέα δεδομένα από το δίκτυο χωρίς καμία επιπλέον λογική στο ViewModel ή το UI.

2.6.4 Paging 3 και RemoteMediator: Σελιδοποίηση με Offline Υποστήριξη

Αν η εφαρμογή φόρτωνε ταυτόχρονα όλες τις ανακοινώσεις ενδεχομένως εκατοντάδες εγγραφές η μνήμη της συσκευής θα εξαντλούνταν γρήγορα και η εφαρμογή θα τερματιζόταν από το λειτουργικό σύστημα. Η βιβλιοθήκη Paging 3 [5] αντιμετωπίζει αυτό εφαρμόζοντας **σελιδοποίηση**: φορτώνει μόνο την πρώτη «σελίδα» (π.χ. 20 ανακοινώσεις) και καθώς ο χρήστης κυλά, φορτώνει αυτόματα την επόμενη, όπως αποτυπώνεται στη στρωματωμένη αρχιτεκτονική του Σχήματος 2.4. Η μνήμη παραμένει σταθερή ανεξάρτητα από τον συνολικό αριθμό ανακοινώσεων.



Σχήμα 2.4: Η στρωματωμένη αρχιτεκτονική Paging 3: Repository, ViewModel και UI συνεργάζονται για αποδοτική φόρτωση δεδομένων [28].

Το Paging 3 από μόνο του δεν εγγυάται offline υποστήριξη. Το **RemoteMediator** [16] είναι το στοιχείο που συνδυάζει τη σελιδοποίηση με την offline-first στρατηγική της Ενότητας 2.6.3: όταν ο χρήστης φτάνει κοντά στο τέλος της τρέχουσας σελίδας, το Paging 3 καλεί το RemoteMediator, το οποίο αποστέλλει αίτημα δικτύου, αποθηκεύει τα αποτελέσματα στο Room, και επιστρέφει. Το αποτέλεσμα είναι ότι το UI δεν αλληλεπιδρά ποτέ απευθείας με το δίκτυο παρατηρεί αποκλειστικά το Room, το οποίο ενημερώνεται διαφανώς από τον RemoteMediator. Η σελιδοποίηση, η ανανέωση και η offline υποστήριξη αντιμετωπίζονται ως ένα ενιαίο σύστημα.

2.6.5 DataStore: Αποθήκευση Προτιμήσεων

Για μικρές ποσότητες δεδομένων προτιμήσεις χρήστη, token ταυτοποίησης ή βιβλιοθήκη **DataStore** [29] αντικαθιστά το παλιό SharedPreferences. Το βασικό πρόβλημα του SharedPreferences ήταν ότι εκτελούσε αναγνώσεις και εγγραφές στο κύριο thread, με κίνδυνο ANR. Το DataStore το επιλύει αυτό χρησιμοποιώντας Coroutines και Flow: κάθε ανάγνωση επιστρέφει Flow και κάθε εγγραφή εκτελείται σε suspend συνάρτηση, εξασφαλίζοντας ότι το κύριο thread δε δεσμεύεται ποτέ.

Η εφαρμογή χρησιμοποιεί και τις δύο εκδόσεις του DataStore για διαφορετικούς σκοπούς. Το **Preferences DataStore** αποθηκεύει απλά ζεύγη κλειδιού-τιμής στην περίπτωση αυτή, το token ταυτοποίησης. Για πιο δομημένα δεδομένα προτιμήσεις χρήστη, ετικέτες, προφίλ χρησιμοποιείται το **Proto DataStore**, το οποίο ορίζει ένα **Protobuf** σχήμα με στατικούς τύπους, ανάλογο με το πώς το Room ορίζει @Entity για τη βάση δεδομένων:

```

message UserPreferences {
    Theme theme = 4;
    ...
}

```

```

}
message SubscribableTagsProto {
    repeated SubscribableTagProto tags = 1;
    ...
}
message ProfileProto{
    int32 id = 1;
    string name = 2;
    ...
}

```

```

// Preferences DataStore απλό ζεύγος κλειδιού-τιμής για το token

```

```

override fun getAboardAccessToken() = datastore.data.map {
    ↪ it[ABOARD_ACCESS_TOKEN_KEY] }

override suspend fun setAboardAccessToken(accessToken: String) {
    datastore.edit {
        it[ABOARD_ACCESS_TOKEN_KEY] = accessToken
    }
}

```

```

// Proto DataStore δομημένη πρόσβαση με στατικούς τύπους

```

```

override fun getUserTheme(): Flow<Theme> = datastore.data.map {
    ↪ it.theme }

override suspend fun setUserTheme(theme: Theme) {
    datastore.updateData { prefs ->
        prefs.toBuilder()
            .setTheme(theme)
            .build()
    }
}

```

Η επιλογή μεταξύ των δύο ακολουθεί έναν σαφή κανόνα: απλά, ανεξάρτητα δεδομένα αποθηκεύονται ως ζεύγη κλειδιού-τιμής, ενώ δομημένα δεδομένα με συσχετίσεις μεταξύ τους αποθηκεύονται σε Protobuf σχήμα, όπου ένα λάθος τύπου αποτελεί compile error και όχι σφάλμα κατά την εκτέλεση.

2.7 Σύνοψη

Το παρόν κεφάλαιο παρουσίασε το τεχνολογικό πλαίσιο της εφαρμογής ως αλληλένδετο σύνολο επιλογών, όχι ως ανεξάρτητη λίστα εργαλείων. Κάθε τεχνολογία απαντά σε συγκεκριμένο πρόβλημα που απορρέει από τους περιορισμούς της πλατφόρμας Android, και κάθε επιλογή συνδέεται με τις υπόλοιπες. Το Clean Architecture απαιτεί DI για να παρέχονται οι υλοποιήσεις των interfaces κατά την εκτέλεση. Το DI διευκολύνει τις αυτοματοποιημένες δοκιμές, επιτρέποντας εύκολη αντικατάσταση εξαρτήσεων με mock αντικείμενα. Το StateFlow ενώνει το ViewModel με το Compose, εξασφαλίζοντας αυτόματη ενημέρωση του UI κάθε φορά που αλλάζει η κατάσταση. Τέλος, ο RemoteMediator ενώνει το Paging 3 με το Room, επιτυγχάνοντας σελιδοποίηση με ταυτόχρονη offline υποστήριξη.

Έχοντας κατανοήσει τι κάνει κάθε εργαλείο και γιατί επιλέχθηκε, το Κεφάλαιο 3 παρουσιάζει τη δομή της εφαρμογής και τις αρχιτεκτονικές αποφάσεις που συνδέουν αυτές τις τεχνολογίες σε ένα ενιαίο σύστημα. Τα Κεφάλαια 4--6 δείχνουν πώς κάθε επίπεδο Data, Domain και UI υλοποιούνται συγκεκριμένα, ακολουθώντας τη ροή του Clean Architecture.

Κεφάλαιο 3

Αρχιτεκτονική και Δομή Εφαρμογής

3.1 Εισαγωγή

Το Κεφάλαιο 2 παρουσίασε τις τεχνολογίες και τα αρχιτεκτονικά πρότυπα που συγκροτούν το **Modern Android Development** [1]. Το παρόν κεφάλαιο παρουσιάζει τον τρόπο με τον οποίο τα πρότυπα αυτά εφαρμόστηκαν δομικά στην εφαρμογή IEE IHU: την τοπολογία του συστήματος, την αντιστοίχιση των επιπέδων Clean Architecture [9] στη δομή του έργου, την αρχιτεκτονική multi-module [11] που διατηρεί χαμηλή σύζευξη μεταξύ οθονών, και τον τρόπο που το Hilt [10] συνδέει αυτόματα ολόκληρο το γράφημα εξαρτήσεων. Η υλοποίηση κάθε επιπέδου ξεχωριστά (**Data** [24], **Domain** [23] και **UI** [22]) αναλύεται στα Κεφάλαια 4--6.

3.2 Συνολική Αρχιτεκτονική του Συστήματος

3.2.1 Τοπολογία Συστήματος

Η εφαρμογή λειτουργεί ως Android client ενός υπάρχοντος Backend συστήματος. Σε υψηλό επίπεδο, το σύστημα αποτελείται από τρία βασικά μέρη που επικοινωνούν αποκλειστικά μέσω HTTP/JSON:

1. **Android Client**: παρουσίαση δεδομένων, διαχείριση κατάστασης, offline αποθήκευση.
2. **Backend REST API**: επιχειρησιακή λογική, αυθεντικοποίηση, παραγωγή δεδομένων.
3. **Τοπική βάση Room**: Single Source of Truth (SSoT) για το UI, ενδιάμεση κρυφή μνήμη (cache), υποστήριξη offline λειτουργίας.

Η τοπική βάση δεδομένων δεν είναι απλώς cache σύμφωνα με την **offline-first** αρχή που τέθηκε στο Κεφάλαιο 2, το UI αντλεί **δεδομένα αποκλειστικά** από τη **βάση Room** και ποτέ απευθείας από το δίκτυο. Ο μηχανισμός **RemoteMediator** [16], ο οποίος αναλύεται διεξοδικά στο Κεφάλαιο 4, αναλαμβάνει στο παρασκήνιο τον συγχρονισμό της Room με το Backend.

3.2.2 Αντιστοίχιση Επιπέδων Clean Architecture

Ακολουθώντας το θεωρητικό πλαίσιο του Κεφαλαίου 2, η εφαρμογή οργανώνεται σε τρία επίπεδα με αυστηρή κατεύθυνση εξάρτησης: κάθε επίπεδο γνωρίζει αποκλειστικά τα υποκείμενα επίπεδα και ποτέ τα υπερκείμενα [13].

Η κατεύθυνση εξάρτησης εφαρμόζει πιστά την αρχή **DIP** (Dependency Inversion Principle) του Κεφαλαίου 2: τα ViewModels [21] εξαρτώνται από Use Case, και τα repositories εφαρμόζουν interfaces που ορίζονται στο domain layer. Η αντικατάσταση της υποδομής δεδομένων (π.χ. αλλαγή API ή βάσης δεδομένων) δεν επηρεάζει τη λογική του UI.

Επίπεδο	Modules	Εξαρτάται από
UI Layer	feature:*	Domain Layer
Domain Layer	core:domain core:model	Από κανένα
Data Layer	core:data	Domain Layer

Πίνακας 3.1: Αντιστοίχιση επιπέδων Clean Architecture σε modules και κατεύθυνση εξάρτησης

3.3 Δομή του Έργου και Πολυαρθρωτή Δομή

Το Κεφάλαιο 2 εξήγησε τα οφέλη της πολυαρθρωτής δομής (multi-module) [11] ως πρότυπο, όπως η σταδιακή μεταγλώττιση (incremental compilation), η επιβολή ορίων ευθυνών και η επαναχρησιμοποίηση κώδικα. Εδώ παρουσιάζεται η εφαρμογή αυτού του προτύπου στην εφαρμογή IEE IHU.

Σε πρακτικό επίπεδο, κάθε module είναι ένας ανεξάρτητος φάκελος κώδικα με το δικό του αρχείο `build.gradle.kts` (όπου ορίζονται οι εξαρτήσεις και οι ρυθμίσεις του module). Το Gradle γνωρίζει ποια modules εξαρτώνται από ποια και, κατά κάθε αλλαγή κώδικα, μεταγλωττίζει μόνο τα επηρεαζόμενα όχι το σύνολο του project. Η εφαρμογή IEE IHU οργανώνει τον κώδικα σε τέσσερις κύριες κατηγορίες modules:

app: Το κεντρικό module της εφαρμογής. Ορίζει το **Application** και το κεντρικό **NavHost** [26], και εξαρτάται από όλα τα feature modules. Είναι το μοναδικό module που «γνωρίζει» ολόκληρη την εφαρμογή.

feature:*: Ένα module ανά λειτουργικότητα που βλέπει ο χρήστης π.χ. **feature:announcements**, **feature:auth**, **feature:settings**. Κάθε feature module περιέχει τα Composables [2], και τα ViewModels [21] της συγκεκριμένης οθόνης.

core:*: Κοινόχρηστος κώδικας που επαναχρησιμοποιείται, άμεσα ή έμμεσα, από πολλά feature modules π.χ. **core:network** (Retrofit [27]), **core:database** (Room [4]), **core:ui** (κοινά Composables), **core:data** (repositories), **core:domain** (Use Case κλάσεις και interfaces).

build-logic: Convention plugins που κεντριοκοπούν τη διαμόρφωση του Gradle, όπως εξηγείται στην Ενότητα 3.3.4.

3.3.1 App Module

Αποτελεί το μοναδικό εκτελέσιμο module, ορίζει:

- Την κλάση **IEEApplication** που αρχικοποιεί το Hilt [10].
- Το root **NavHost** που συνδέει όλα τα feature modules.
- Το **AppModule** του Hilt με τις scopes εφαρμογής.

Σε αντίθεση με τις προγενέστερες «God Activities» (κλάσεις Activity που συγκέντρωναν όλη τη λογική της εφαρμογής σε ένα αρχείο), το app module δεν περιέχει επιχειρησιακή λογική και λειτουργεί αποκλειστικά ως σημείο σύνθεσης (composition root), δηλαδή ως το σημείο στο οποίο αρχικοποιούνται και συνδέονται όλα τα υπόλοιπα modules.

3.3.2 Core Modules

Τα core modules παρέχουν κοινή υποδομή σε όλα τα feature modules. Ένα feature module εισάγει *μόνο* ό,τι χρειάζεται, αποφεύγοντας περιττές εξαρτήσεις:

- **core:model**: Περιέχει μόνο μοντέλα που χρειάζονται στο domain layer.
- **core:domain**: Use Cases και Repository interfaces.
- **core:data**: Repository implementations, mapper functions.
- **core:database**: Βάση δεδομένων Room, DAOs, Entity κλάσεις.
- **core:network**: Retrofit interfaces, Interceptors, DTO κλάσεις.
- **core:datastore**: Proto DataStore [29] για τις προτιμήσεις χρήστη.
- **core:designsystem**: Τυπογραφία, Χρωματική παλέτα, atomic composables [30].
- **core:ui**: Επαναχρησιμοποιήσιμα composables.
- **core:work**: WorkManager [6] tasks για εργασίες παρασκηνίου.
- **core:notifications**: Κανάλια και αποστολή ειδοποιήσεων [31].
- **core:analytics** / **core:crashlytics**: Αποτυπώσεις χρήσης και σφαλμάτων.

3.3.3 Feature Modules

Κάθε feature module αντιστοιχεί σε μία ολοκληρωμένη οθόνη και εξαρτάται *μόνο* από core modules ποτέ από άλλα feature modules. Αυτό διατηρεί τη σύζευξη χαμηλή και επιτρέπει ανεξάρτητη ανάπτυξη:

- **feature:home**: Κύρια ροή ανακοινώσεων.
- **feature:announcement**: Λεπτομέρειες ανακοίνωσης, λήψη συνημμένων.

- **feature:search:** Αναζήτηση ανακοινώσεων.
- **feature:profile:** Διαχείριση συνδρομών, προβολή λεπτομερειών χρήστη.
- **feature:settings:** Ρύθμιση θέματος, συχνότητα ειδοποιήσεων κ.α.
- **feature:auth:** Οθόνη σύνδεσης.
- **feature:licenses:** Άδειες ανοιχτού λογισμικού.

3.3.4 Build-Logic Module

Χωρίς convention plugins, κάθε feature module θα έπρεπε να επαναλαμβάνει τις ίδιες δεκάδες γραμμές ρυθμίσεων Gradle (ποια έκδοση Android SDK να χρησιμοποιεί, ποιες βιβλιοθήκες να εισάγει αυτόματα κ.λπ.). Ένα **convention plugin** είναι ένα επαναχρησιμοποιήσιμο κομμάτι ρύθμισης Gradle που εφαρμόζεται με μία γραμμή σε οποιοδήποτε module, εξασφαλίζοντας DRY (Don't Repeat Yourself) κώδικα και κεντρικό έλεγχο ρυθμίσεων.

```
class FeatureConventionPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        with(project) {
            apply(plugin = "com.kastik.library.compose")
            apply(plugin = "com.kastik.hilt")
            dependencies {
                add("implementation", project(":core:domain"))
                add("implementation", project(":core:ui"))
                add("implementation", project(":core:designsystem"))
            }
        }
    }
}
```

Με αυτόν τον τρόπο, ένα νέο feature module χρειάζεται να αναφέρει μόνο id του convention plugin στο `build.gradle.kts` του, χωρίς να επαναλαμβάνει δεκάδες γραμμές ρυθμίσεων.

3.4 Dependency Injection με Hilt

Όπως εξηγήθηκε στο Κεφάλαιο 2, το Clean Architecture απαιτεί τα επίπεδα να εξαρτώνται από interfaces, αλλά είναι απαραίτητος ένας μηχανισμός που θα παρέχει τις συγκεκριμένες υλοποιήσεις κατά την εκτέλεση. Το **Hilt** [10] αναλαμβάνει αυτόν τον ρόλο στην εφαρμογή IEE IHU. Κάθε module ορίζει τα δικά του Hilt modules που υποδεικνύουν στο framework ποια υλοποίηση αντιστοιχεί σε κάθε interface. Παρακάτω παρουσιάζονται τα τρία βασικά σημεία σύνδεσης της εφαρμογής.

Το `NetworkModule` στο `core:network` παρέχει το `OkHttpClient`, το `Retrofit` [27] και τα API services. Το annotation `@InstallIn(SingletonComponent::class)` ορίζει

ότι τα παρεχόμενα αντικείμενα είναι singleton σε επίπεδο εφαρμογής, δηλαδή δημιουργούνται μία φορά και διατηρούνται στη μνήμη όσο η εφαρμογή εκτελείται:

```
@Module
@InstallIn(SingletonComponent::class)
object NetworkModule {
    @Provides
    @Singleton
    fun provideOkHttpClient(
        tokenInterceptor: TokenInterceptor,
        aboardAuthenticator: AboardAuthenticator,
    ): OkHttpClient =
        OkHttpClient.Builder()
            .connectTimeout(15, TimeUnit.SECONDS)
            .readTimeout(30, TimeUnit.SECONDS)
            .addInterceptor(tokenInterceptor)
            .authenticator(aboardAuthenticator)
            .build()

    @Provides
    @Singleton
    fun provideRetrofit(
        client: OkHttpClient,
        json: Json
    ): Retrofit = Retrofit.Builder()
        .baseUrl(API_BASE_URL)
        .addConverterFactory(SortTypeQueryConverterFactory())
        .addConverterFactory(json.asConverterFactory(contentType))
        .client(client)
        .build()

    @Provides
    @Singleton
    fun provideAboardService(
        retrofit: Retrofit
    ): AnnouncementApiService =
        retrofit.create(AnnouncementApiService::class.java)
}
```

Αντίστοιχα, το DatabaseModule στο core:database παρέχει τη Room βάση ως singleton. Το annotation @ApplicationContext ζητά από το Hilt να εγχύσει αυτόματα το Context της εφαρμογής απαραίτητο για τη δημιουργία της βάσης δεδομένων:

```
@Module
```

```
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(
        @ApplicationContext context: Context
    ): AppDatabase =
        Room.databaseBuilder(
            context,
            AppDatabase::class.java,
            "announcement_cache.db"
        ).build()

    @Provides
    fun provideAnnouncementDao(db: AppDatabase): AnnouncementDao =
        db.announcementDao()

    @Provides
    fun provideRemoteKeyDao(db: AppDatabase): RemoteKeyDao =
        db.remoteKeyDao()
}
```

Στα feature modules, τα ViewModels λαμβάνουν τις εξαρτήσεις τους αυτόματα μέσω @HiltViewModel και @Inject constructor δεν απαιτείται χειροκίνητη δημιουργία αντικειμένων:

```
@HiltViewModel
class HomeViewModel @Inject constructor(
    private val getAnnouncementsUseCase: GetAnnouncementsUseCase,
    private val getUserSubscriptionsUseCase:
        ⇨ GetUserSubscriptionsUseCase
) : ViewModel() { /* ... */ }
```

Το αποτέλεσμα είναι ότι κάθε επίπεδο του Clean Architecture γνωρίζει μόνο τα interfaces που χρειάζεται ποτέ τις συγκεκριμένες υλοποιήσεις ενώ το Hilt διαχειρίζεται διαφανώς ολόκληρο το γράφημα εξαρτήσεων της εφαρμογής.

3.5 Σύνοψη

Το παρόν κεφάλαιο έθεσε τα δομικά θεμέλια της εφαρμογής IEE IHU: την τοπολογία του συστήματος με τη Room ως SSoT, την αντιστοίχιση των επιπέδων Clean Architecture σε modules, τη multi-module οργάνωση που επιβάλλει χαμηλή σύζευξη μέσω του Gradle, και το Hilt που συνδέει αυτόματα ολόκληρο το γράφημα εξαρτήσεων.

Τα επόμενα τρία κεφάλαια αναλύουν την υλοποίηση κάθε επιπέδου ξεχωριστά: το Κεφάλαιο 4 καλύπτει το Data Layer (Retrofit, Room, Paging 3, DataStore), το Κεφάλαιο 5 το Domain Layer (Use Cases, Repository interfaces), και το Κεφάλαιο 6 το UI Layer (Composables, ViewModels, Navigation).

Κεφάλαιο 4

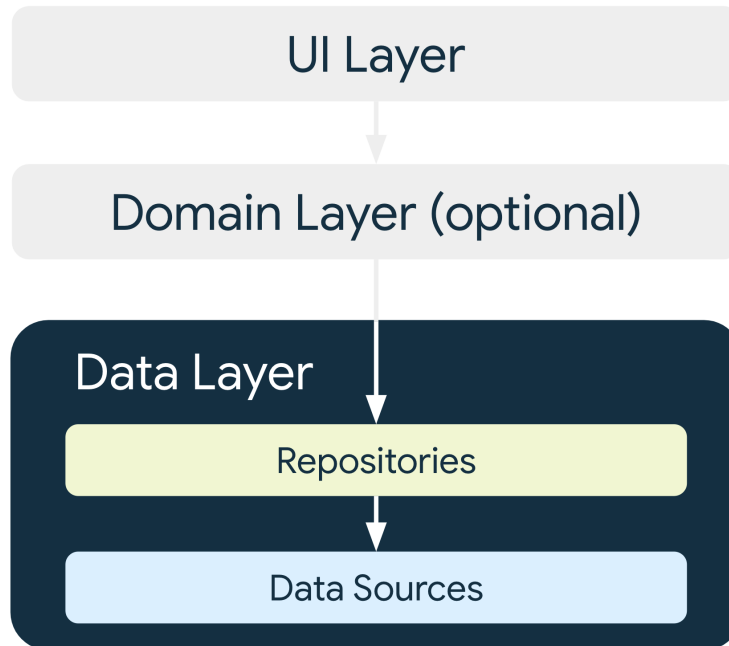
Υλοποίηση Data Layer

4.1 Εισαγωγή

Στο Κεφάλαιο 3 ορίστηκαν τα αρχιτεκτονικά θεμέλια της εφαρμογής: η προσέγγιση Clean Architecture με τα τρία ομόκεντρα επίπεδα, η multi-module δομή και η αυτόματη διαχείριση εξαρτήσεων μέσω Hilt. Το παρόν κεφάλαιο εστιάζει στο θεμελιώδες επίπεδο υλοποίησης και αναλύει το **Data Layer**, το εξωτερικότερο επίπεδο της αρχιτεκτονικής, το οποίο είναι υπεύθυνο για την απόκτηση, αποθήκευση και παροχή δεδομένων στα ανώτερα επίπεδα.

Βάσει της αρχής Αντιστροφής Εξαρτήσεων (Dependency Inversion Principle), το Data Layer εξαρτάται από αφαιρέσεις που ορίζει το Domain Layer, και όχι το αντίστροφο: τα interfaces των Repositories δηλώνονται στο Domain και υλοποιούνται στο Data. Παρ' όλα αυτά, στην παρούσα ανάλυση το Data Layer παρουσιάζεται πρώτο, ακολουθώντας μια «από κάτω προς τα πάνω» προσέγγιση. Η επιλογή αυτή υπαγορεύεται από παιδαγωγικούς λόγους: η κατανόηση της υποδομής δεδομένων, του πώς ρέουν τα δεδομένα από το δίκτυο στη βάση και από τη βάση στο UI, αποτελεί απαραίτητο εφόδιο για να εκτιμηθεί στη συνέχεια η αφαιρεκτική λογική του Domain Layer (Κεφάλαιο 5). Η αρχιτεκτονική ορθότητα της κατεύθυνσης των εξαρτήσεων παραμένει ανεπηρέαστη από αυτή τη σειρά παρουσίασης.

Το Data Layer της εφαρμογής IEE IHU αποτελείται από τέσσερα υποσυστήματα: τα **μοντέλα δεδομένων** (DTOs, Entities και Domain Models), τον **client δικτύου** (Retrofit + OkHttp), την **τοπική αποθήκευση δεδομένων** (Room + DataStore) και τον μηχανισμό **offline-first συγχρονισμού** (Paging 3 + RemoteMediator). Συνδυαστικά, υλοποιούν την αρχή της **Μοναδικής Πηγής Αλήθειας (Single Source of Truth)** [13]: το UI αντλεί δεδομένα αποκλειστικά από το Room, ενώ το δίκτυο χρησιμεύει μόνο ως μηχανισμός ενημέρωσης αυτής της βάσης. Με αυτόν τον τρόπο, η εφαρμογή παραμένει λειτουργική ακόμα και χωρίς σύνδεση στο διαδίκτυο, εμφανίζοντας τα τελευταία γνωστά δεδομένα. Η ιδιότητα αυτή αποκτά κρίσιμη σημασία σε περιβάλλοντα με ασταθή συνδεσιμότητα.



Σχήμα 4.1: Αρχιτεκτονική Επισκόπηση: Data Layer [24]

4.2 Μοντέλα Δεδομένων

4.2.1 Αιτιολόγηση της Τριεπίπεδης Δομής

Ένα συνηθισμένο αρχιτεκτονικό σφάλμα σε εφαρμογές Android είναι η χρήση ενός μοναδικού μοντέλου δεδομένων σε όλα τα επίπεδα. Η προσέγγιση αυτή, παρόλο που μειώνει αρχικά τον αριθμό των κλάσεων, δημιουργεί ισχυρή σύζευξη (tight coupling): η δομή του JSON που επιστρέφει το API, το σχήμα της τοπικής βάσης και η αναπαράσταση που απαιτεί το UI σπάνια συμπίπτουν πλήρως. Μια αλλαγή στο API επιβάλλει τότε την ταυτόχρονη τροποποίηση της βάσης δεδομένων και του επιπέδου διεπαφής (UI layer), παραβιάζοντας την αρχή της Ενιαίας Ευθύνης (Single Responsibility).

Το Data Layer της εφαρμογής χρησιμοποιεί τρία ξεχωριστά μοντέλα, ένα για κάθε επίπεδο:

1. **DTOs:** Αντικατοπτρίζουν πιστά τη δομή της απόκρισης JSON του API.
2. **Entities:** Αντιπροσωπεύουν τις εγγραφές στο Room.
3. **Domain Models:** Χρησιμοποιούνται από τα Use Cases, τα ViewModels και τα Composables.

Η διαχωρισμένη αυτή δομή διασφαλίζει ότι μια ενδεχόμενη αλλαγή στο σχήμα του API δε θα επηρεάσει τους αντίστοιχους mappers, τη βάση δεδομένων ή το UI.

4.2.2 DTOs: Η Αναπαράσταση του API Response

Τα **DTOs** (Data Transfer Objects) μοντελοποιούν επακριβώς τη δομή του JSON που επιστρέφει το REST API. Για τη σειριοποίηση επιλέχθηκε η βιβλιοθήκη `kotlinx.serialization` έναντι

εναλλακτικών όπως τα Gson και Moshi. Η επιλογή αυτή αιτιολογείται από τρεις παράγοντες: πρώτον, η `kotlinx.serialization` αποτελεί την επίσημη βιβλιοθήκη σειριοποίησης της JetBrains και ευθυγραμμίζεται πλήρως με το υπόλοιπο Kotlin-first οικοσύστημα της εφαρμογής [8]. Δεύτερον, επεξεργάζεται τα δεδομένα κατά τον χρόνο μεταγλώττισης (compile-time), αποφεύγοντας τη χρήση reflection, η οποία θα επιβάρυνε την απόδοση της εφαρμογής κατά τον χρόνο εκτέλεσης (runtime). Τέλος, υποστηρίζει εγγενώς (natively) τους nullable τύπους της Kotlin [17], ελαχιστοποιώντας τον κίνδυνο εμφάνισης `NullPointerException` λόγω μη αναμενόμενων null πεδίων στην απόκριση.

Επιπλέον, στις ρυθμίσεις του αναλυτή (parser) JSON, έχει ενεργοποιηθεί η παράβλεψη άγνωστων πεδίων (`ignoreUnknownKeys = true`). Η ρύθμιση αυτή προσφέρει ανθεκτικότητα θωρακίζοντας την εφαρμογή έναντι εξαιρέσεων σειριοποίησης σε περίπτωση που το Backend προσθέσει νέα, μη αναμενόμενα πεδία στην απόκριση μελλοντικά.

Η δομή των DTOs υλοποιείται με τη χρήση κατάλληλων annotations για την αντιστοίχιση των πεδίων του JSON, όπως φαίνεται στο ακόλουθο ενδεικτικό απόσπασμα:

```
@Serializable
data class AnnouncementDto (
    @SerializedName("id") val id: Int,
    @SerializedName("title") val title: String,
    @SerializedName("body") val body: String,
    @SerializedName("createdAt") val createdAt: String,
    @SerializedName("author") val author: AuthorDto,
    @SerializedName("tags") val tags: List<TagDto>
)
```

4.2.3 Entities: Η Αναπαράσταση στη Βάση Δεδομένων

Ένα **Entity** αντιπροσωπεύει μια εγγραφή πίνακα στο Room [4]. Το annotation `@Entity` ορίζει τον πίνακα, το `@PrimaryKey` καθορίζει το πρωτεύον κλειδί και το `@ForeignKey` επιβάλλει την αναφορική ακεραιότητα μεταξύ των πινάκων.

Αξίζει να σημειωθεί ότι το `AnnouncementEntity` δεν αποθηκεύει ολόκληρο το σώμα της ανακοίνωσης. Αυτό αποθηκεύεται χωριστά στο `BodyEntity`. Η συγκεκριμένη σχεδιαστική απόφαση επιτρέπει την αποθήκευση των εκτενών πεδίων κειμένου σε ξεχωριστό πίνακα. Έτσι, αποφεύγεται η φόρτωσή τους κατά την ανάκτηση λιστών, μειώνοντας σημαντικά τη χρήση μνήμης και τον χρόνο απόκρισης των ερωτημάτων.

```
@Entity(
    foreignKeys = [
        ForeignKey(entity = AuthorEntity::class, parentColumns = ["id"],
            ↪ childColumns = ["authorId"], onDelete = CASCADE)
    ],
    indices = [Index("id"), Index("authorId")]
)
data class AnnouncementEntity (
```

```

@PrimaryKey val id: Int,
val authorId: Int,
val title: String,
val createdAt: String
)

```

Η χρήση της ιδιότητας `onDelete = CASCADE` εξασφαλίζει ότι αν διαγραφεί ένας συγγραφέας, διαγράφονται αυτόματα και οι αντίστοιχες ανακοινώσεις του, αποτρέποντας τη δημιουργία ορφανών εγγγραφών στη βάση.

4.2.4 Mapper Functions: Μετατροπή μεταξύ Επιπέδων

Η μετατροπή μεταξύ των μοντέλων πραγματοποιείται μέσω των **mapper functions**. Πρόκειται για extension functions της Kotlin, μια προγραμματιστική δυνατότητα που επεκτείνει υπάρχοντες τύπους χωρίς τη χρήση κληρονομικότητας, διατηρώντας τον κώδικα των μετασχηματισμών συνοπτικό και ευανάγνωστο. Οι συναρτήσεις αυτές είναι καθαρές χωρίς εξωτερικές εξαρτήσεις, γεγονός που απλοποιεί τη δοκιμή τους. Ενδεικτικά, η μετατροπή από το επίπεδο των DTOs στο επίπεδο των Entities και του Domain Model υλοποιείται ως εξής:

```

fun AnnouncementDto.toAnnouncementEntity() = AnnouncementEntity(
    id = id, authorId = author.id, title = title, createdAt = createdAt
)

fun AnnouncementPreviewRelation.toAnnouncement() = Announcement(
    id = announcement.id, title = announcement.title, body = "",
    tags = tags.map { it.toTag() }, author = author.name, date =
        ↪ announcement.createdAt
)

```

4.3 Επικοινωνία Δικτύου

Η επικοινωνία με το Backend διαχειρίζεται από το module `core:network`.

4.3.1 Επιλογή Βιβλιοθήκης Δικτύου

Για την υλοποίηση του HTTP client εξετάστηκαν οι κύριες εναλλακτικές. Το **Retrofit** [27] είναι ένα επίπεδο αφαίρεσης (abstraction layer) πάνω από το **OkHttp**, που επιτρέπει τη δήλωση REST endpoints ως Kotlin interfaces, με τον κώδικα επικοινωνίας να παράγεται αυτόματα. Αντίθετα, η εγγενής χρήση του **OkHttp** παρέχει μέγιστο έλεγχο, αλλά απαιτεί τη συγγραφή εκτενούς επαναλαμβανόμενου κώδικα (boilerplate).

Επιλέχθηκε το **Retrofit** για τους εξής λόγους: η δηλωτική (declarative) σύνταξη των interfaces μειώνει σημαντικά τον boilerplate κώδικα. Η ωριμότητα της βιβλιοθήκης και η εκτεταμένη τεκμηρίωσή της ελαχιστοποιούν τον κίνδυνο απρόβλεπτων ζητημάτων σε παραγωγικό περιβάλλον,

ενώ η εγγενής υποστήριξη των Kotlin coroutines μέσω των suspend functions [18] εξαλείφει την ανάγκη χρήσης callbacks.

4.3.2 API Interface: AboardApiClient

Το AboardApiClient ορίζει όλα τα REST endpoints ως suspend functions, εξασφαλίζοντας ότι κάθε κλήση δικτύου εκτελείται ασύγχρονα εντός ενός Coroutine [18]:

```
interface AboardApiClient {
    @GET("announcements")
    suspend fun getAnnouncements(
        @Query("page") page: Int,
        @Query("perPage") perPage: Int
    ): PagedAnnouncementsResponseDto
}
```

4.3.3 Διαχείριση Αυθεντικοποίησης

Η διαχείριση της αυθεντικοποίησης στην εφαρμογή απαιτεί ιδιαίτερη αρχιτεκτονική μέριμνα λόγω ενός κλασικού προβλήματος: εάν ο Authenticator που ανανεώνει το token σε περίπτωση απόκρισης 401 χρησιμοποιεί τον ίδιο Retrofit client με τα κανονικά αιτήματα, προκαλείται ένας ατέρμων βρόχος (infinite loop) δικτυακών αιτημάτων. Κάθε αίτημα ανανέωσης token που αποτυγχάνει, θα ενεργοποιούσε εκ νέου τον Authenticator.

Η λύση στο ζήτημα αυτό είναι η χρήση δύο διακριτών clients μέσω Hilt qualifiers: ο **BaseAboardClient**, ένας βασικός client χωρίς ενσωματωμένο Authenticator, χρησιμοποιείται αποκλειστικά για την κλήση του endpoint ανανέωσης. Αντίθετα, ο **AuthenticatorAboardClient** φέρει ολόκληρη τη στοίβα των interceptors, καθώς και τον Authenticator για τα υπόλοιπα αιτήματα.

Το OkHttp pipeline επεξεργάζεται τα αιτήματα μέσω δύο βασικών συνιστωσών:

1. **TokenInterceptor**: Επισυνάπτει αυτόματα το JWT token στην κεφαλίδα Authorization.
2. **AboardAuthenticator**: Ενεργοποιείται αποκλειστικά κατά τη λήψη αποκρίσεων με κωδικό 401 ώστε να ανανεώσει το token.

4.3.4 Remote Data Source: Ενθυλάκωση της API Κλήσης

Σύμφωνα με την αρχή της ενθυλάκωσης, παρεμβάλλεται το επίπεδο **Remote Data Source** μεταξύ του Retrofit interface και του Repository. Αυτή η αφαίρεση απομονώνει τις δικτυακές λεπτομέρειες, επιτρέποντας την προετοιμασία των παραμέτρων και διευκολύνοντας την αντικατάσταση του δικτυακού επιπέδου με fakes κατά τη διάρκεια των δοκιμών.

4.4 Τοπική Βάση Δεδομένων: Room

4.4.1 Επιλογή Βιβλιοθήκης Βάσης Δεδομένων

Για την τοπική αποθήκευση δομημένων δεδομένων εξετάστηκαν τρεις κύριες εναλλακτικές. Το **Room** [4] είναι το επίσημο abstraction layer της Google πάνω από το SQLite, πλήρως ολοκληρωμένο στο Android Jetpack. Το **SQLDelight** είναι ένα ORM που παράγει type-safe κώδικα από SQL ερωτήματα. Το **Realm** είναι μια αντικειμενοστρεφής βάση δεδομένων.

Επιλέχθηκε το **Room** για τους εξής λόγους: η βαθιά ενσωμάτωσή του με το οικοσύστημα του Android Jetpack (ιδιαίτερα με το Paging 3 [5] και τα Kotlin Flows [19]) αποτελεί κρίσιμο πλεονέκτημα για τη συγκεκριμένη αρχιτεκτονική. Επιπλέον, ο compile-time έλεγχος των SQL ερωτημάτων εντοπίζει σφάλματα πριν από την εκτέλεση, ενώ η ενσωματωμένη υποστήριξη για migrations διευκολύνει την εξέλιξη του σχήματος.

4.4.2 AppDatabase: Ορισμός Βάσης

Η κλάση AppDatabase αποτελεί το κεντρικό σημείο ορισμού της βάσης δεδομένων, καταγράφοντας τα entities, την έκδοση του σχήματος και τους απαραίτητους TypeConverters.

Η προετοιμασία της βάσης μέσω Hilt [10] ενσωματώνει δύο κρίσιμες ρυθμίσεις:

- **Write Ahead Logging (WAL):** Επιτρέπει την ταυτόχρονη εκτέλεση αναγνώσεων και εγγραφών. Αυτό αποτρέπει το μπλοκάρισμα του UI κατά τη διάρκεια της ενημέρωσης των δεδομένων από τον RemoteMediator [16].
- **Fallback to Destructive Migration:** Επαναφέρει τη βάση στην αρχική της κατάσταση σε περίπτωση αναντιστοιχίας εκδόσεων, εφόσον δεν έχει οριστεί διαδρομή μετάπτωσης (migration path). Η επιλογή αυτή είναι αποδεκτή καθώς η βάση λειτουργεί αποκλειστικά ως τοπική cache.

4.4.3 DAOs: Δηλωτική Πρόσβαση στη Βάση

Τα **DAOs (Data Access Objects)** ορίζουν τις λειτουργίες πρόσβασης στα δεδομένα με δηλωτικό τρόπο. Ο μεταγλωττιστής παράγει αυτόματα τον κώδικα εκτέλεσης και επαληθεύει τα SQL ερωτήματα κατά τη μεταγλώττιση. Οι μέθοδοι ανάκτησης σύνθετων σχέσεων επισημαίνονται με το annotation @Transaction ώστε η εκτέλεση των επιμέρους SELECT ερωτημάτων να πραγματοποιούνται αδιαίρετα (atomically), διασφαλίζοντας την ακεραιότητα των δεδομένων. Παράλληλα, η επιστροφή τύπων Flow [19] ή PagingSource [5] επιτρέπει την αυτόματη προώθηση των αλλαγών. Η τελική αντιδραστική χαρτογράφηση (reactive mapping) αυτών των ροών από Entities της βάσης σε Domain Models πραγματοποιείται στο επίπεδο του Repository. Έτσι, το Repository γεφυρώνει το DAO με τα Use Cases, προστατεύοντας τα ανώτερα επίπεδα από τεχνικές λεπτομέρειες αποθήκευσης.

4.4.4 Room Relations: Σύνθετες Αναπαραστάσεις

Το Room υποστηρίζει σύνθετες αναπαραστάσεις μέσω των **Relations** [32]. Η εφαρμογή ορίζει δύο διαφορετικές «σχέσεις» για την ίδια οντότητα, ανάλογα με το πλαίσιο χρήσης, αξιοποιώντας

με αυτόν τον τρόπο την εξοικονόμηση μνήμης που αναφέρθηκε προηγουμένως.

Το **AnnouncementPreviewRelation** χρησιμοποιείται για λίστες, φορτώνοντας μόνο τα απαραίτητα πεδία χωρίς το πλήρες σώμα κειμένου, ενώ το **AnnouncementDetailRelation** φορτώνεται μόνο κατά την προβολή μιας ανακοίνωσης και συμπεριλαμβάνει το ένθετο **BodyEntity**. Η δομή τους βασίζεται στα annotations **@Embedded** και **@Relation**, καθώς και στη χρήση της κλάσης **Junction** για την υλοποίηση της σχέσης πολλά-προς-πολλά μέσω του ενδιάμεσου πίνακα σύνδεσης **TagsCrossRefEntity**.

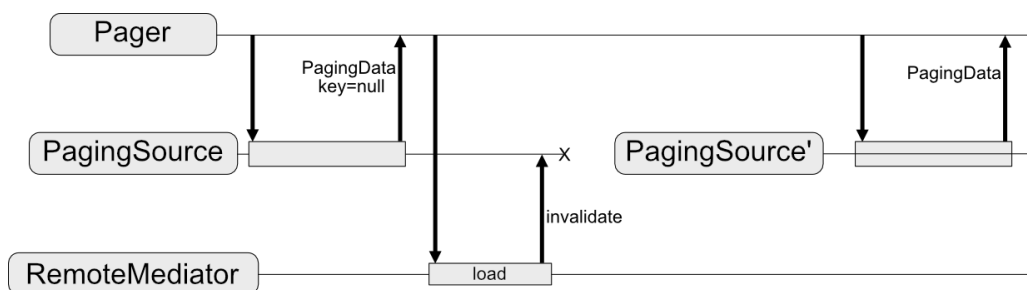
4.5 Paging 3 και RemoteMediator

4.5.1 Αρχιτεκτονική Offline-First με Paging 3

Η προσέγγιση offline-first διασφαλίζει ότι μια εφαρμογή λειτουργεί πλήρως χωρίς σύνδεση δικτύου, αξιοποιώντας τοπικά δεδομένα που συγχρονίζονται κατά την επανασύνδεση. Σε αντίθεση με την απλή προβολή cached δεδομένων, η τοπική βάση αποτελεί πάντα τη μοναδική πηγή αλήθειας (single source of truth), ακόμη και όταν υπάρχει ενεργή σύνδεση [13].

Για τη σταδιακή φόρτωση μεγάλων λιστών χρησιμοποιείται η βιβλιοθήκη Paging 3. Η βιβλιοθήκη αυτή αναλαμβάνει την τμηματική φόρτωση (pagination) των δεδομένων από το δίκτυο και την τοπική βάση, διατηρώντας τα συγχρονισμένα και προσφέροντας μια ομαλή εμπειρία χρήστη (UX). Η αρχιτεκτονική αυτού του μηχανισμού δομείται σε τρία βασικά στοιχεία:

- **PagingSource**: Φορτώνει δεδομένα από το Room, επιστρέφοντας **AnnouncementPreviewRelation**.
- **RemoteMediator**: Συντονίζει τη φόρτωση από το δίκτυο και την αποθήκευση στο Room, επιτρέποντας στο **PagingSource** να διαβάζει πάντα από το Room [16].
- **Pager**: Δημιουργεί το **PagingData** flow και καθορίζει τις παραμέτρους σελιδοποίησης.



Σχήμα 4.2: Κύκλος ζωής Paging 3 με RemoteMediator

4.5.2 Υλοποίηση και Κανονικοποιημένη Αποθήκευση

Ο μηχανισμός **AnnouncementRemoteMediator** αποτελεί τον εννοητικό της offline-first στρατηγικής [16]. Η βασική του λειτουργία εστιάζει στη μέθοδο **load()**, η οποία προσδιορίζει την τρέχουσα σελίδα φόρτωσης με βάση τον τύπο (**LoadType.REFRESH**, **LoadType.APPEND** ή **LoadType.PREPEND**).

Ένα κρίσιμο στοιχείο για τη διατήρηση της ακεραιότητας των δεδομένων είναι η διαχείριση της παλαιότερης προσωρινής μνήμης (cache invalidation). Όταν εκτελείται **REFRESH** (π.χ. χειροκίνητη ανανέωση από τον χρήστη), ο RemoteMediator διαγράφει τις παλιές ανακοινώσεις από τη βάση δεδομένων και ανακτά ένα φρέσκο αντίγραφο από τον διακομιστή. Αυτό εξασφαλίζει την αφαίρεση τυχόν στοιχείων που έχουν στο μεταξύ διαγραφεί στο Backend. Σε περίπτωση **APPEND**, ανακτώνται τα αντίστοιχα RemoteKeys από τη βάση δεδομένων για τον προσδιορισμό του επόμενου δείκτη (index) και τα νέα δεδομένα προστίθενται.

Η αποθήκευση των ληφθέντων DTOs πραγματοποιείται εντός μιας ενιαίας συναλλαγής (database transaction). Αυτό διασφαλίζει ότι, σε περίπτωση σφάλματος ακόμη και κατά τη διαδικασία ανανέωσης (refresh) τα προσωρινά αποθηκευμένα δεδομένα παραμένουν άθικτα, προστατεύοντας την ακεραιότητα της βάσης.

4.5.3 Ρύθμιση του Pager

Ο Pager συνδέει το PagingSource του Room με τον RemoteMediator [16]. Η διαμόρφωση της κλάσης **PagingConfig** ορίζει το μέγεθος της σελίδας (**pageSize = 20**) και το αρχικό μέγεθος φόρτωσης (**initialLoadSize = 60**), εξασφαλίζοντας επαρκή δεδομένα για την αρχική εμφάνιση. Παράλληλα, η ενεργοποίηση των placeholders (**enablePlaceholders = true**) επιτρέπει στο UI να προβάλλει skeleton components κατά τη διάρκεια ασύγχρονων διεργασιών.

4.6 Αποθήκευση Προτιμήσεων: DataStore

4.6.1 Επιλογή Μηχανισμού Αποθήκευσης Προτιμήσεων

Για δεδομένα μικρής κλίμακας (όπως προτιμήσεις χρήστη, authentication tokens, προφίλ), η χρήση του Room [4] θα ήταν υπερβολική: οι σχεσιακές δυνατότητες δεν απαιτούνται και το overhead του σχήματος είναι άσκοπο. Η κλασική λύση στο Android, το SharedPreferences, παρουσιάζει γνωστά προβλήματα: αποθηκεύει και διαβάζει δεδομένα σύγχρονα στο κύριο νήμα (main thread, με κίνδυνο εμφάνισης ANR), δεν παρέχει type safety και δεν υποστηρίζει δομημένα σχήματα.

Το **DataStore** [29], η επίσημη αντικατάσταση του SharedPreferences, αντιμετωπίζει όλα αυτά τα ζητήματα: λειτουργεί πλήρως ασύγχρονα, εκθέτει τα δεδομένα ως Flow [19] και παρέχει δύο παραλλαγές: το **Proto DataStore** (για δομημένα δεδομένα μέσω Protocol Buffers) και το **Preferences DataStore** (για απλά ζεύγη κλειδιού-τιμής).

4.6.2 Υλοποίηση και Θωράκιση

Η εφαρμογή αξιοποιεί το **Proto DataStore** [29] για τη διαχείριση δομημένων δεδομένων, συμπεριλαμβανομένων των προτιμήσεων χρήστη (User Preferences), του προφίλ (Profile) και των συνδρομών (Subscriptions). Μέσω της υλοποίησης των αντίστοιχων Serializers (π.χ. `Serializer<UserPreferences>`), καθορίζονται οι προεπιλεγμένες τιμές και η διαδικασία αποσειριοποίησης από το InputStream.

Σε όλες αυτές τις υλοποιήσεις ακολουθείται κοινή αρχιτεκτονική προστασίας και πρόσβασης. Η αρχικοποίηση του DataStore ενσωματώνει έναν ReplaceFileCorruptionHandler, ο οποίος κατα-

γράφει τυχόν σφάλματα καταστροφής αρχείων στο Crashlytics και επαναφέρει αυτόματα τις αρχικές τιμές, αποτρέποντας την κατάρρευση (crash) της εφαρμογής. Παράλληλα, το Migration API εξασφαλίζει τη συμβατότητα και μεταφορά των δεδομένων κατά τις αναβαθμίσεις. Τέλος, η πρόσβαση απομονώνεται πίσω από interfaces (όπως το PreferencesLocalDataSource), διασφαλίζοντας ότι τα ανώτερα επίπεδα επικοινωνούν αποκλειστικά μέσω Flow, χωρίς γνώση του υποκείμενου μηχανισμού αποθήκευσης

4.7 Διαχείριση και Χαρτογράφηση Σφαλμάτων

Ένας κρίσιμος ρόλος του Data Layer είναι η προστασία των ανώτερων επιπέδων από τεχνικές λεπτομέρειες εξαιρέσεων (όπως `HttpException`, `IOException`). Η εφαρμογή ακολουθεί το πρότυπο του Anti-Corruption Layer: τα σφάλματα αναχαιτίζονται στα Repositories και μετατρέπονται σε αντικείμενα του Domain Layer.

Η ιεραρχία των σφαλμάτων μοντελοποιείται μέσω κλειστών κλάσεων (sealed class) (π.χ. `NetworkError`), η οποία επιτρέπει στον μεταγλωττιστή να επαληθεύει την εξαντλητική κάλυψη όλων των περιπτώσεων.

Η μετατροπή των εξαιρέσεων πραγματοποιείται από συναρτήσεις επέκτασης (π.χ. `.toNetworkError()`), οι οποίες χαρτογραφούν σφάλματα (όπως `UnknownHostException` ή `SocketTimeoutException`) σε domain αντικείμενα (`NetworkError.Connection`, `NetworkError.Authentication` κ.λπ.). Στα Repositories, η διαχείριση αυτή υλοποιείται με try-catch. Για την απόλυτη προστασία της εφαρμογής, χρησιμοποιείται ένα τελικό μπλοκ else το οποίο αναχαιτίζει κάθε άλλη απρόβλεπτη εξαίρεση (π.χ. σφάλματα βάσης δεδομένων ή σειριοποίησης) και τη χαρτογραφεί σε ένα γενικό σφάλμα (π.χ. `NetworkError.Unknown`).

4.8 Επεξεργασία Εικόνων: Base64ImageExtractor

Μια ειδική πρόκληση αφορά τις εικόνες στο σώμα των ανακοινώσεων: το API επιστρέφει εικόνες κωδικοποιημένες ως Base64 απευθείας μέσα στο σώμα του κειμένου. Αυτό επιβαρύνει τη βάση δεδομένων και υποβαθμίζει την απόδοση του UI.

Η κλάση `Base64ImageExtractorImpl` επιλύει το πρόβλημα αυτό πριν την αποθήκευση των δεδομένων στη βάση, ακολουθώντας τα εξής βήματα:

1. Εντοπίζει τις Base64 εικόνες μέσω Regular Expression.
2. Αποκωδικοποιεί παράλληλα τις εικόνες.
3. Παράγει ένα μοναδικό αναγνωριστικό όνομα αρχείου εφαρμόζοντας τον αλγόριθμο SHA-256.
4. Αποθηκεύει τις εικόνες στον φάκελο προσωρινής μνήμης της εφαρμογής.
5. Αντικαθιστά τα Base64 URLs (`data:image/*;base64`) με τοπικά Uri (`file://`).

4.9 Συνημμένα Αρχεία: DownloadManager

Η λήψη συνημμένων αρχείων (όπως έγγραφα PDF) ανατίθεται στο module core:downloader, το οποίο αξιοποιεί το API του **DownloadManager** του λειτουργικού συστήματος Android. Η επιλογή αυτή, έναντι μιας χειροκίνητης υλοποίησης (π.χ. μέσω OkHttp streams), κρίνεται βέλτιστη καθώς το λειτουργικό σύστημα αναλαμβάνει αυτόματα τη διαχείριση των διακοπών δικτύου, την επανέναρξη λήψεων, την προβολή ειδοποιήσεων προόδου στο σύστημα και την ασφαλή εγγραφή στον δημόσιο κατάλογο λήψεων.

Η κλάση FileDownloaderImpl προετοιμάζει ένα αντικείμενο Request, εισάγοντας τις απαραίτητες κεφαλίδες αυθεντικοποίησης (Authorization: Bearer <token>) για την πρόσβαση σε προστατευμένα αρχεία.

Μόλις ολοκληρωθεί η λήψη, το Android εκπέμπει ένα σήμα (DOWNLOAD_COMPLETE). Αυτό συλλαμβάνεται από τον DownloadCompleteReceiver, ο οποίος ανακτά το Uri του αρχείου και εκκινεί μια πρόθεση (Intent) τύπου ACTION_VIEW ώστε να ανοίξει το αρχείο αυτόματα κάποια άλλη εφαρμογή του χρήστη. Για τη συμμόρφωση με το μοντέλο ασφαλείας του Android, στο Intent επισυνάπτεται ρητά το flag FLAG_GRANT_READ_URI_PERMISSION, επιτρέποντας σε εξωτερικές εφαρμογές προβολής (π.χ. PDF viewers) να αναγνώσουν το ληφθέν αρχείο και να το ανοίξουν.

4.10 Σύνοψη

Το παρόν κεφάλαιο ανέλυσε το Data Layer της εφαρμογής IEE IHU, εστιάζοντας σε τρία επίπεδα: τις αρχιτεκτονικές αποφάσεις και την αιτιολόγησή τους, τις υλοποιήσεις και τους περιορισμούς που αυτές συνεπάγονται.

Το Κεφάλαιο 5 μεταβαίνει στο αμέσως ανώτερο επίπεδο: αναλύει το Domain Layer, τους επιχειρησιακούς κανόνες και τις αφαιρέσεις που ορίζουν τις δυνατότητες της εφαρμογής.

Κεφάλαιο 5

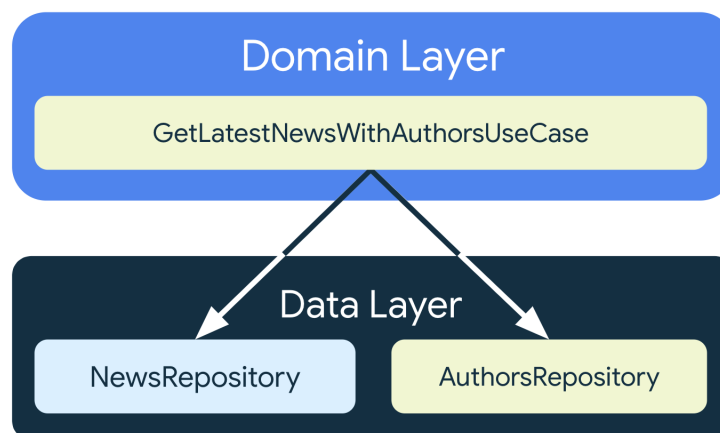
Υλοποίηση Domain Layer

5.1 Εισαγωγή

Το Domain Layer είναι το κεντρικό, ανεξάρτητο επίπεδο του Clean Architecture [9]. Είναι πλήρως αποσυζευγμένο (decoupled) από τεχνολογίες υποδομής, όπως βάσεις δεδομένων, δίκτυο ή UI frameworks (π.χ. Jetpack Compose). Περιέχει αποκλειστικά την **επιχειρησιακή λογική**: τι μπορεί να κάνει η εφαρμογή και κάτω από ποιες συνθήκες.

Σε πρακτικό επίπεδο αποτελείται από τρία στοιχεία: τα **domain models** (τα «καθαρά» αντικείμενα της επιχειρησιακής λογικής, ανεξάρτητα από API ή βάση δεδομένων), τα **Repository interfaces** (αφηρημένες συμβάσεις που ορίζουν τι δεδομένα χρειάζεται το layer) και τα **Use Cases** (κλάσεις αποκλειστικής ευθύνης που ενθυλακώνουν μια συγκεκριμένη επιχειρησιακή λειτουργία).

Ο λόγος ύπαρξης αυτού του layer είναι η **απομόνωση**: ο κώδικας επιχειρησιακής λογικής μπορεί να ελεγχθεί με καθαρά JVM unit tests, παραμένοντας ουσιαστικά αποσυζευγμένος από το framework του Android, τις βάσεις δεδομένων ή το δίκτυο.



Σχήμα 5.1: Αρχιτεκτονική Επισκόπηση: Επίπεδο Domain [23]

5.2 Domain Models

Τα domain models ορίζονται στο **core:model** module ως καθαρές Kotlin κλάσεις, χωρίς annotations αποθήκευσης (π.χ. Room) ή σειριοποίησης (π.χ. JSON) και χωρίς καμία άλλη εξάρτηση.

Αυτή η προσέγγιση εξασφαλίζει ότι τα δεδομένα παραμένουν ανεπηρέαστα από τον τρόπο αποθήκευσης ή μεταφοράς τους. Για παράδειγμα:

```
data class Announcement (  
    val id: Int,  
    val title: String,  
    val content: String,  
    val createdAt: String  
)
```

5.3 Repository Interfaces

Κάθε Repository interface ορίζεται στο module **core:domain** και υλοποιείται στο **core:data**. Αυτή η αντιστροφή εξάρτησης (Dependency Inversion Principle) σημαίνει ότι το Domain Layer δεν εξαρτάται ποτέ από την υλοποίηση, παρά μόνο από την αφαίρεση (abstraction).

Κάθε interface ακολουθεί σταθερές συμβάσεις:

- Οι μέθοδοι επιστρέφουν αποκλειστικά domain models, πρωταρχικούς τύπους ή ροές δεδομένων (Flows).
- Λειτουργίες που ενδέχεται να αποτύχουν και δεν επιστρέφουν ροές δεδομένων ενθυλακώνονται σε μια δομή Result, ώστε τα σφάλματα να διαχειρίζονται με ασφάλεια.
- Τα ίδια τα σφάλματα αναπαρίστανται αυστηρά ως domain-level models, αποκρύπτοντας την πηγή του προβλήματος (π.χ. δίκτυο ή τοπική βάση).

Η υλοποίηση αυτών των interfaces στο Data Layer (Κεφάλαιο 4) διαχειρίζεται την πηγή των δεδομένων με τρόπο εντελώς διαφανή για το Domain Layer. Αξίζει να σημειωθεί ότι, ως πραγματιστικός συμβιβασμός για την αποδοτική διαχείριση μεγάλου όγκου δεδομένων, επιτρέπεται κατ' εξαίρεση η χρήση του PagingData στο Domain Layer, παρότι αποτελεί τμήμα του ευρύτερου Android οικοσυστήματος.

Η χρήση ανασταλτικών συναρτήσεων (suspend) επιτρέπει την ασύγχρονη εκτέλεση λειτουργιών χωρίς τον αποκλεισμό του νήματος (thread blocking). Αντίστοιχα, οι μέθοδοι που επιστρέφουν Flow αντιπροσωπεύουν ασύγχρονες ροές δεδομένων, οι οποίες εκπέμπουν νέες τιμές κάθε φορά που η υποκείμενη πηγή μεταβάλλεται. Όταν μια λειτουργία ολοκληρώνεται, επιστρέφει μια γενική δομή Result<D, E>. Σε περίπτωση επιτυχίας, ο τύπος D περιέχει τα ζητούμενα δεδομένα ή Unit, ενώ σε αποτυχία επιστρέφεται ένα Error που ορίζει υποκλάσεις για συγκεκριμένα σφάλματα σε επίπεδο επιχειρησιακής λογικής.

```
interface AnnouncementRepository {
    fun getPagedAnnouncements(): Flow<PagingData<Announcement>>
    fun getAnnouncementById(id: Int): Flow<Announcement?>
    suspend fun syncAnnouncements(): Result<Unit, DataError>
}
```

5.4 Use Cases

Ένα Use Case είναι μια κλάση που εκτελεί μία και μόνο επιχειρησιακή λειτουργία. Η ειδική μέθοδος operator fun invoke() επιτρέπει να καλείται το Use Case σαν να ήταν απλή συνάρτηση (getAnnouncementsUseCase() αντί για getAnnouncementsUseCase.invoke()).

Αν και συχνά ένα Use Case απλώς λειτουργεί ως περίβλημα για μια μεμονωμένη συνάρτηση ενός Repository, η πραγματική του αξία αναδεικνύεται σε πιο σύνθετα σενάρια. Συγκεκριμένα, ένα Use Case μπορεί να:

- Συνδυάζει πληροφορίες καλώντας πολλαπλά Repositories.
- Συνδυάζει, μετασχηματίζει και φιλτράρει τα δεδομένα πριν τα παραδώσει στο επόμενο επίπεδο.
- Περιέχει καθαρή επιχειρησιακή λογική που είναι εντελώς ανεξάρτητη από οποιοδήποτε Repository.

Τα Use Cases εξαρτιούνται μόνο από *interfaces*. Αν το domain layer δει ποτέ τον τύπο Room ή Retrofit, είναι αρχιτεκτονικό λάθος:

```
// 1. Απλό Use Case
class GetAnnouncementsUseCase @Inject constructor(
    private val repository: AnnouncementRepository
) {
    operator fun invoke(): Flow<PagingData<Announcement>> =
        ↪ repository.getPagedAnnouncements()
}

// 2. Σύνθετο Use Case
class SignInUserUseCase @Inject constructor(
    private val profileRepository: ProfileRepository,
    private val authenticationRepository: AuthenticationRepository,
) {
    suspend operator fun invoke(code: String): Result<Unit,
        ↪ DataError> {
        val authResult = authenticationRepository
            .exchangeCodeForAboardToken(code)
        return when (authResult) {
            is Result.Success -> profileRepository.refreshProfile()
            is Result.Error -> authResult
        }
    }
}
```

```
    }  
  }  
}  
  
// 3. Use Case χωρίς εξάρτηση από Repository  
class ValidateSearchQueryUseCase @Inject constructor() {  
    operator fun invoke(query: String): Boolean {  
        return query.isNotBlank() && query.length >= 3  
    }  
}
```

5.5 Σύνοψη

Το Domain Layer είναι το πιο σταθερό τμήμα της εφαρμογής και αλλάζει σπάνια, καθώς οι επιχειρησιακοί κανόνες δεν εξαρτώνται από τεχνολογικές επιλογές. Τα **Repository interfaces** ορίζουν τη σύμβαση μεταξύ domain και data layer. Τα **Use Cases** ενθυλακώνουν κάθε επιχειρησιακή λειτουργία σε μια κλάση αποκλειστικής ευθύνης. Τα **domain models** αναπαριστούν τα δεδομένα χωρίς λεπτομέρειες υλοποίησης.

Το αποτέλεσμα είναι ότι κάθε Use Case μπορεί να ελεγχθεί σε καθαρό JVM περιβάλλον, χωρίς συσκευή και χωρίς mock του Android SDK μια ιδιότητα που επαληθεύεται ποσοτικά στο Κεφάλαιο 8.

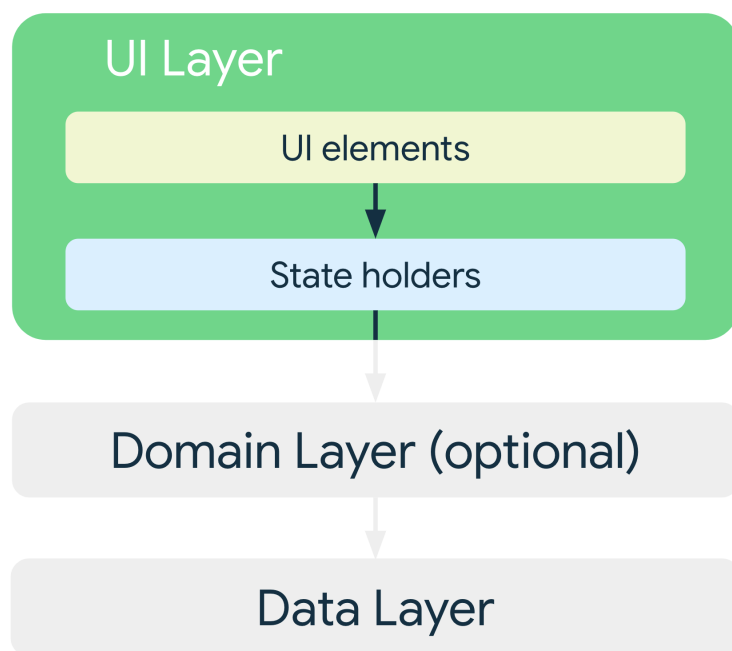
Το Κεφάλαιο 6 περνάει στο εξωτερικό επίπεδο της αρχιτεκτονικής: τη διεπαφή χρήστη. Ενώ το Domain Layer ορίζει τι κάνει η εφαρμογή, το UI Layer ορίζει πώς το βλέπει και αλληλεπιδρά ο χρήστης.

Κεφάλαιο 6

Υλοποίηση UI Layer

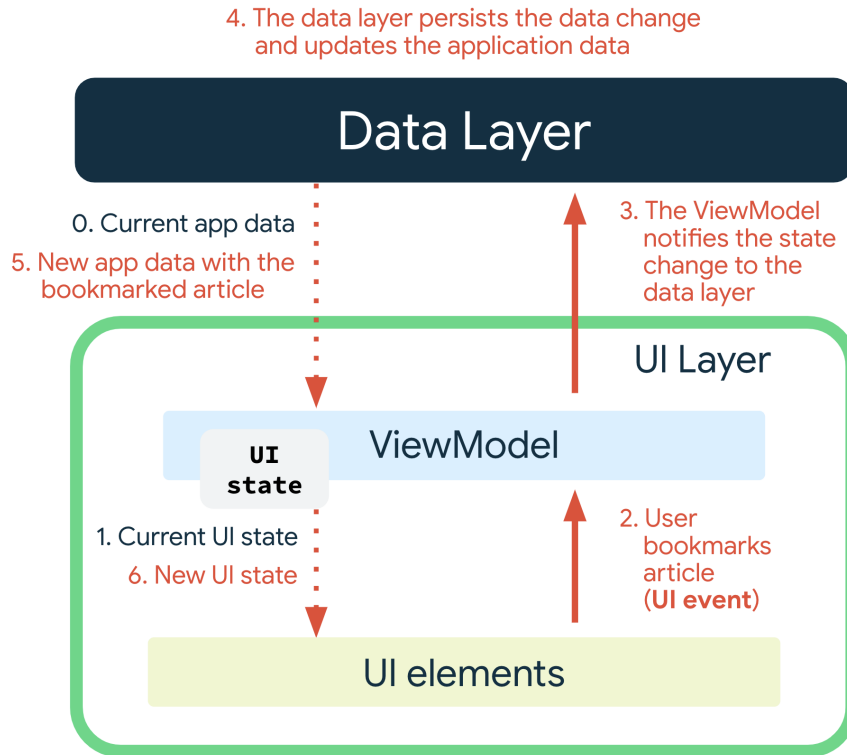
6.1 Εισαγωγή

Το UI Layer είναι το τελευταίο επίπεδο του Clean Architecture, το μόνο που «βλέπει» ο χρήστης.



Σχήμα 6.1: Αρχιτεκτονική Επισκόπηση: Επίπεδο UI [22]

Η βασική αρχή που διέπει ολόκληρο το επίπεδο είναι το **Unidirectional Data Flow (UDF)**. Σύμφωνα με το UDF, η κατάσταση (state) ρέει προς τα κάτω από το ViewModel προς το Composable ($UI = f(state)$), και τα events ρέουν προς τα πάνω.



Σχήμα 6.2: Το UDF μοτίβο σε δράση: StateFlow ρέει προς το Composable, ενώ τα events χρήστη ανεβαίνουν αντίθετα [22]

Εφαρμόζοντας αυτή τη λογική, το επίπεδο διεπαφής της εφαρμογής IEE IHU αποτελείται από τέσσερα βασικά στοιχεία που αναλύονται παρακάτω:

1. **ViewModels:** Διαχείριση των events του χρήστη και δημιουργία State.
2. **Συναρτήσεις Composable:** Δηλωτική κατασκευή UI που αντιδρά στο State.
3. **Navigation:** Type-Safe πλοήγηση μεταξύ οθονών.
4. **Design System:** Ενιαία αισθητική με Material 3.

6.2 ViewModels

Τα ViewModels αποτελούν τον πυρήνα της λογικής παρουσίασης, απομονώνοντας τα δεδομένα από το ίδιο το UI.

6.2.1 Δομή State και Αρχικοποίηση

Αντί να χρησιμοποιεί η οθόνη πολλές ξεχωριστές μεταβλητές κατάστασης, η εφαρμογή χρησιμοποιεί **sealed classes** που αντιπροσωπεύουν αμοιβαία αποκλειόμενες καταστάσεις (π.χ. Loading, Success, Error):

```
sealed class HomeUiState {
    data object Loading : HomeUiState
    data class Error(val message: String) : HomeUiState
    data class Success(
        val tags: List<Tag>,
        val authors: List<Author>
    ) : HomeUiState
}
```

Αντί για τη χρήση init blocks, τα δεδομένα αντλούνται δηλωτικά μέσω **Flows** και μετατρέπονται σε **StateFlow** μέσω του **stateIn**, συνδέοντας άμεσα τον κύκλο ζωής με το UI χωρίς side-effects:

```
@HiltViewModel
class HomeViewModel @Inject constructor(
    getTagsUseCase: GetTagsUseCase,
    getAuthorsUseCase: GetAuthorsUseCase,
) : ViewModel() {

    val uiState: StateFlow<HomeUiState> = combine(
        getTagsUseCase(),
        getAuthorsUseCase()
    ) { tags, authors ->
        HomeUiState.Success(tags = tags, authors = authors)
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5000L),
        initialValue = HomeUiState.Loading
    )
}
```

Έχοντας πλέον διαμορφώσει την κατάσταση με ασφάλεια στο ViewModel, το επόμενο βήμα είναι η οπτική της απόδοση.

6.3 Συναρτήσεις Composable

Τα Composables αναλαμβάνουν να παρατηρούν την κατάσταση του ViewModel και να προσαρμόζουν τη διεπαφή χρήστη δυναμικά.

6.3.1 State Collection & Hoisting

Το Composable χρησιμοποιεί την `collectAsStateWithLifecycle()` προκειμένου να παύει αυτόματα η συλλογή όταν η εφαρμογή μεταβαίνει στο παρασκήνιο. Το `when` εξαντλεί επιλεκτικά όλες τις δυνατές καταστάσεις:

```

@Composable
fun HomeScreenRoute(
    viewModel: HomeViewModel = hiltViewModel(),
) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    when (val state = uiState) {
        is HomeUiState.Loading -> LoadingWheel()
        is HomeUiState.Error -> ErrorScreen(
            message = state.message,
            onRetry = viewModel::onRetry
        )
        is HomeUiState.Success -> HomeScreen(
            tags = state.tags,
            authors = state.authors,
            onTagClick = viewModel::onTagClick,
            onAuthorClick = viewModel::onAuthorClick
        )
    }
}

```

Σύμφωνα με την αρχή **State Hoisting**, η HomeScreen παίρνει την κατάσταση εξωτερικά και εκτελεί callbacks, καθιστώντας τον κώδικα εύκολα δοκιμάσιμο.

Με την οπτική απεικόνιση των μεμονωμένων οθονών έτοιμη, ακολουθεί η διασύνδεσή τους για τη δημιουργία μιας πλήρους εμπειρίας.

6.4 Πλοήγηση

Το Navigation component οργανώνει τον «χάρτη» της εφαρμογής και το πώς ο χρήστης κινείται μεταξύ των Composables.

6.4.1 Type-Safe Navigation

Το Compose Type Safe Navigation [26] μεταφέρει τον έλεγχο των παραμέτρων πλοήγησης στον μεταγλωττιστή, αποτρέποντας σφάλματα της παλαιάς string-based προσέγγισης. Η κατεύθυνση πλοήγησης γίνεται μέσω callback από τα Composables προς τον γονέα (NavHost):

```

@Serializable object HomeRoute
@Serializable data class AnnouncementRoute(val announcementId: Int)

NavHost(navController = navController, startDestination = HomeRoute)
↳ {
    composable<HomeRoute> {
        HomeScreen(onAnnouncementClick = { id ->

```

```

        navController.navigate(AnnouncementRoute(id))
    })
}
composable<AnnouncementRoute> { backStackEntry ->
    val route = backStackEntry.toRoute<AnnouncementRoute>()
    AnnouncementDetailScreen(announcementId =
        ↪ route.announcementId)
}
}

```

6.4.2 Deep Linking

Ένα **deep link** ανοίγει απευθείας μια συγκεκριμένη οθόνη. Στην περίπτωση μας, διαχειρίζεται εσωτερικά link (`com.kastik.apps://`) αλλά και web URLs του Backend:

```

composable<AnnouncementRoute> {
    deepLinks = listOf(
        navDeepLink<AnnouncementRoute> {
            basePath = "com.kastik.apps://announcement"
        },
        navDeepLink<AnnouncementRoute> {
            basePath = "https://aboard.iee.ihu.gr/announcements"
        }
    )
}

```

Έχοντας πλέον έναν πλήρη, λειτουργικό χάρτη πλοήγησης, το τελευταίο στοιχείο είναι η εφαρμογή μιας ενιαίας οπτικής ταυτότητας.

6.5 Design System

Το Design System εξασφαλίζει μια συνεκτική αισθητική και χρηστική εμπειρία σε όλες τις οθόνες της εφαρμογής.

6.5.1 Material Design 3

Η υλοποίηση οργανώνεται βάσει του **Material Design 3** [30] και της αρχής του **Atomic Design**, συνθέτοντας μικρά components σε μεγαλύτερες οθόνες. Σε επίπεδο δομής, η εφαρμογή αποφεύγει το τυπικό `BottomNavigationBar`, οργανώνοντας τη βασική πλοήγηση μέσω **Top App Bar actions**, κάτι που αρμόζει στη δευτερεύουσα φύση των ρυθμίσεων και του προφίλ.

6.5.2 Επιπλέον Χαρακτηριστικά

Η εφαρμογή έχει σχεδιαστεί για να προσαρμόζεται πλήρως στις προτιμήσεις του χρήστη:

- **Dark Mode:** Πλήρης υποστήριξη σκοτεινής λειτουργίας.
- **Dynamic Color:** Η χρωματική παλέτα προσαρμόζεται δυναμικά στο wallpaper της συσκευής.

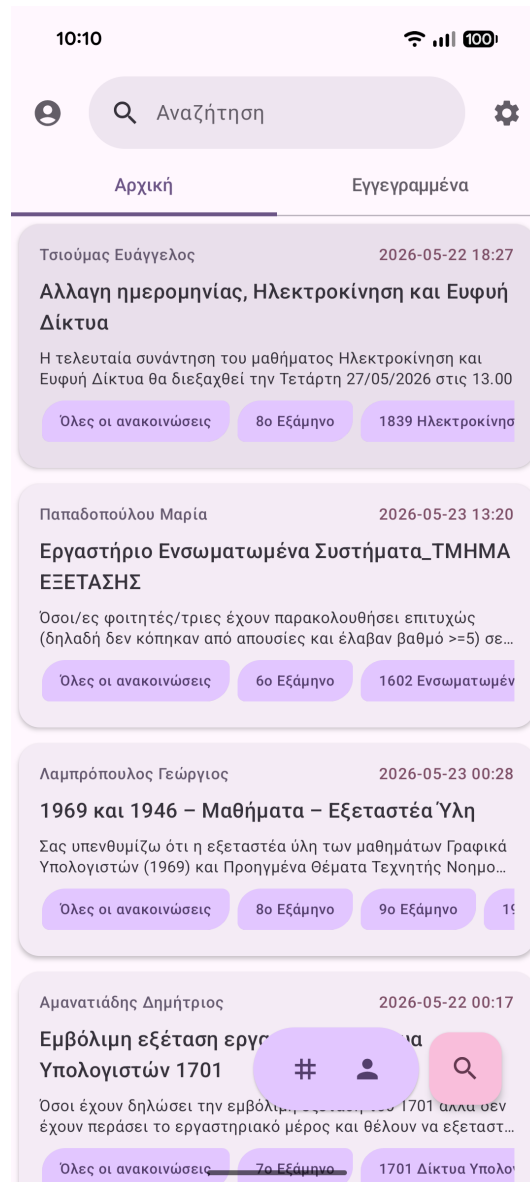
6.6 Παρουσίαση Διεπαφής Χρήστη

Η διεπαφή χρήστη υλοποιείται αποκλειστικά με **Jetpack Compose** [2], ακολουθώντας τις αρχές **Material 3** [30] και το **UDF μοτίβο** που αναλύθηκαν στα προηγούμενα τμήματα. Κάθε οθόνη είναι άμεση αντανάκλαση της κατάστασης που εκθέτει το αντίστοιχο ViewModel. Ακολουθεί η παρουσίαση των βασικών οθονών της εφαρμογής.

6.6.1 Αρχική Οθόνη Ανακοινώσεων

Η αρχική οθόνη (feature:home) αποτελεί το κύριο σημείο εισόδου. Παρουσιάζει δύο καρτέλες: **Αρχική** (γενική ροή ανακοινώσεων) και **Εγγεγραμμένα** (φιλτραρισμένη βάση εγγραφών).

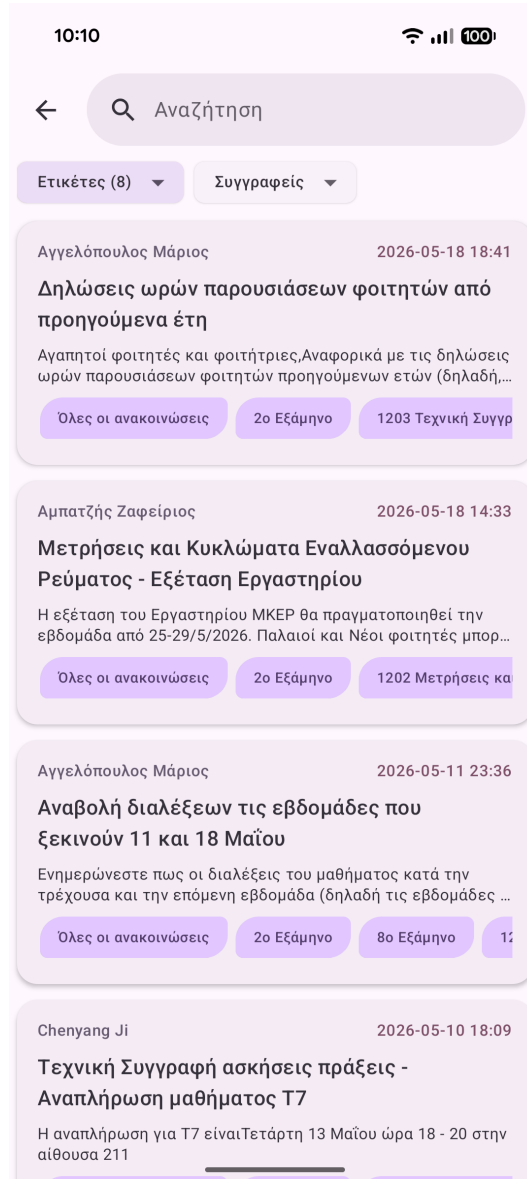
Η λίστα ανακοινώσεων φορτώνεται προοδευτικά με τη χρήση **LazyColumn**. Κάθε στοιχείο εμφανίζει: τίτλο, όνομα συγγραφέα, ημερομηνία δημοσίευσης, προεπισκόπηση κειμένου και τις ετικέτες της ανακοίνωσης. Στην κορυφή το top app bar παρέχει πρόσβαση στην αναζήτηση, τα φίλτρα, το προφίλ και τις ρυθμίσεις.



Σχήμα 6.3: Αρχική οθόνη ροή ανακοινώσεων με καρτέλες «Αρχική» / «Εγγεγραμμένα»

6.6.2 Οθόνη Αναζήτησης

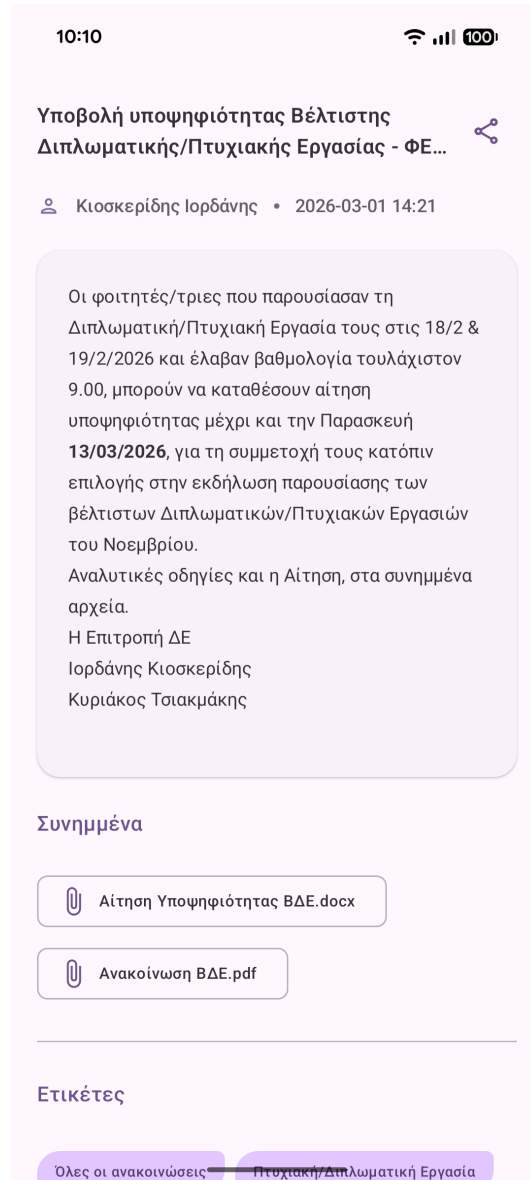
Η οθόνη αναζήτησης (feature:search) ενεργοποιείται όταν ο χρήστης εκτελέσει κάποια αναζήτηση κείμενου ή όταν εφαρμόσει κάποιο φίλτρο.



Σχήμα 6.4: Οθόνη αναζήτησης με ενεργοποιημένα φίλτρα ετικετών

6.6.3 Οθόνη Λεπτομερειών Ανακοίνωσης

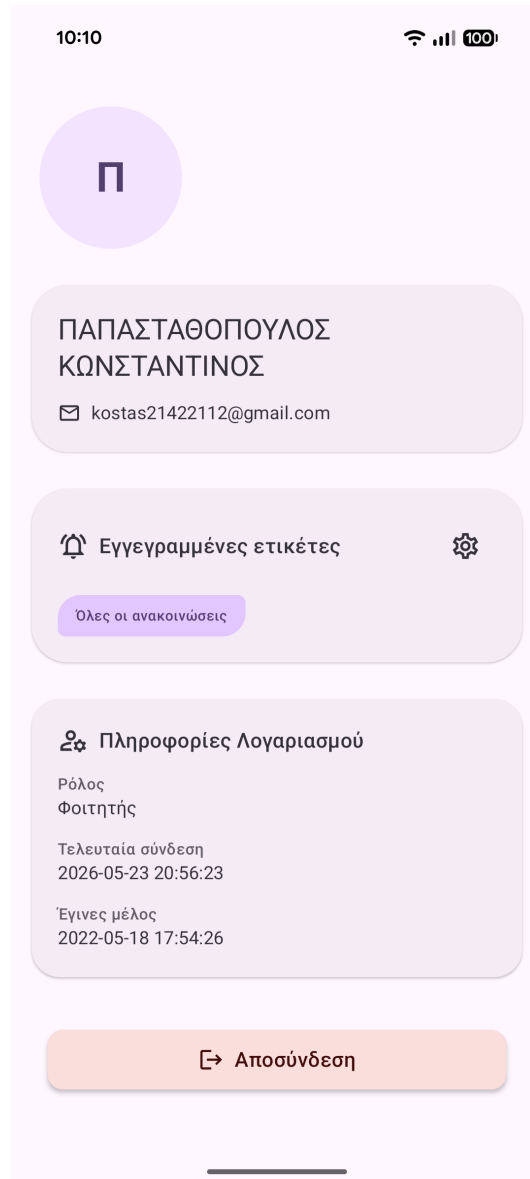
Η οθόνη λεπτομερειών (feature:announcement) παρουσιάζει τίτλο, συγγραφέα, ημερομηνία, πλήρες κείμενο, εικόνες, συνημμένα αρχεία, και ετικέτες. Επιπλέον δίνει τη δυνατότητα λήψης συνημμένων αρχείων πατώντας τα.



Σχήμα 6.5: Οθόνη λεπτομερειών ανακοίνωσης με πλήρες κείμενο και συνημμένα

6.6.4 Οθόνη Προφίλ και Διαχείρισης Συνδρομών

Η οθόνη προφίλ (feature:profile) παρέχει πληροφορίες του χρήστη και επιτρέπει τη διαχείριση ετικετών.

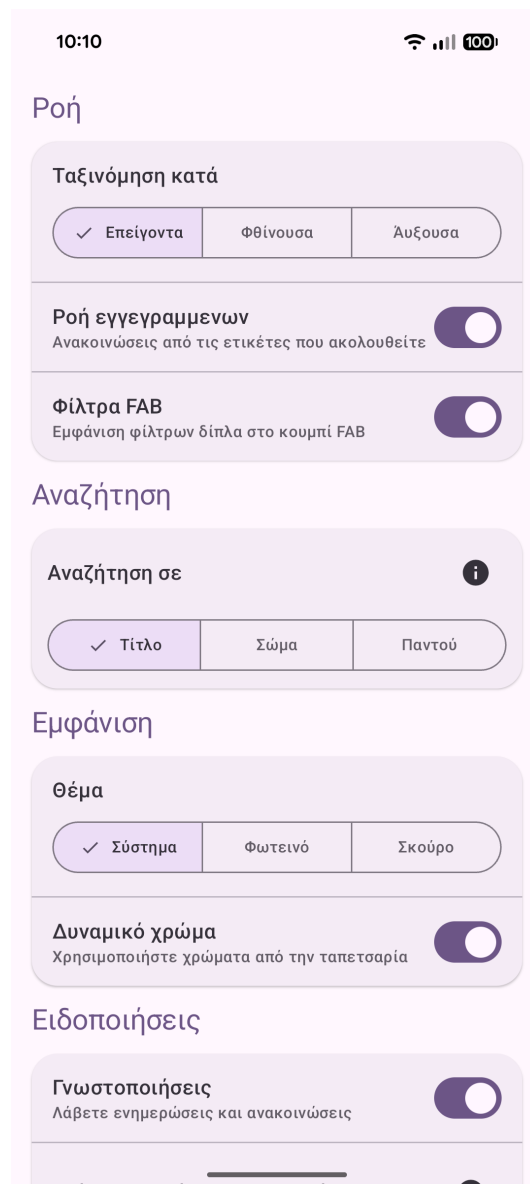


Σχήμα 6.6: Οθόνη προφίλ χρήστη

6.6.5 Οθόνη Ρυθμίσεων

Η οθόνη ρυθμίσεων (feature:settings) παρέχει τέσσερις βασικές κατηγορίες επιλογών:

- **Ρυθμίσεις ροής** : Επιλογή ταξινόμησης, ενεργοποίηση ροής «Εγγεγραμμένα» και ενεργοποίηση εμφάνισης φίλτρων στο FAB.
- **Ρυθμίσεις αναζήτησης** : Επιλογή σε ποια πεδία θα γίνει η αναζήτηση κείμενου (Τίτλο / Σώμα / Παντού).
- **Εμφάνιση**: Θέμα (Φωτεινό / Σκοτεινό / Σύστημα) και Dynamic Color.
- **Ειδοποιήσεις**: Ενεργοποίηση Ειδοποιήσεων, Συχνότητα συγχρονισμού.



Σχήμα 6.7: Οθόνη ρυθμίσεων θέμα, συχνότητα ειδοποιήσεων

6.7 Σύνοψη

Το παρόν κεφάλαιο παρουσίασε το UI Layer της εφαρμογής IEE IHU. Ακολουθώντας αυστηρά το μοτίβο UDF, τα **ViewModels** εκθέτουν StateFlow και τα **Composables** αντιδρούν δυναμικά. Παράλληλα, το **Type-Safe Navigation** οργανώνει με ασφάλεια τη ροή, ενώ το **Material 3** εξασφαλίζει την ενιαία αισθητική. Τέλος, η παρουσίαση των βασικών οθονών ανέδειξε την πρακτική εφαρμογή αυτών των αρχιτεκτονικών επιλογών στην τελική διεπαφή του χρήστη. Το Κεφάλαιο 7 εξετάζει τις οριζόντιες λειτουργίες που διατρέχουν την εφαρμογή: τις ειδοποιήσεις, την απόδοση, και τις δοκιμές.

Κεφάλαιο 7

Οριζόντιες Λειτουργίες

7.1 Εισαγωγή

Τα Κεφάλαια 4--6 ανέλυσαν την εφαρμογή κατά layer. Ωστόσο, ορισμένες λειτουργίες δεν ανήκουν αποκλειστικά σε κάποιο layer. Αυτές αναλύονται στο παρόν κεφάλαιο:

- **Ειδοποιήσεις:** Background συγχρονισμός μέσω WorkManager που αγγίζει Data, Domain και UI layer.
- **Απόδοση:** R8 compiler, Baseline Profile, Startup Profile και Macrobenchmark που βελτιστοποιούν την απόδοση και μετρούν τους χρόνους εκκίνησης.
- **Στρατηγική δοκιμών:** Ο τρόπος με τον οποίο τα data, database και network modules ελέγχονται σε απομόνωση μέσω Unit Tests.

7.2 Σύστημα Ειδοποιήσεων και WorkManager

Η ενημέρωση του χρήστη για νέες ανακοινώσεις αποτελεί κύριο στόχο της εφαρμογής. Λόγω έλλειψης υποστήριξης push notifications από το υπάρχον Backend του ΔΙΠΑΕ, αναπτύχθηκε ένας εναλλακτικός μηχανισμός **polling** βασισμένος στον WorkManager. Η αρχιτεκτονική του **core:notifications** module έχει σχεδιαστεί έτσι ώστε να επιτρέπει τη μελλοντική αντικατάστασή του χωρίς αντίκτυπο στο UI ή στο domain layer. Ακολουθεί η ανάλυση του συστήματος, από τον πυρήνα της λογικής του έως την εκτέλεση και τον χρονοπρογραμματισμό.

7.2.1 Επιχειρησιακή Λογική Αναζήτησης

Η επιχειρησιακή λογική της αναζήτησης νέων ειδοποιήσεων ενθυλακώνεται πλήρως στην κλάση `CheckNewAnnouncementsUseCase`. Η λειτουργία της σχεδιάστηκε με γνώμονα την ελαχιστοποίηση χρήσης πόρων (δίκτυο, μπαταρία) και εκτελείται σε τρία κρίσιμα στάδια:

1. **Συγχρονισμός & Πρόωρος Τερματισμός:** Αρχικά ανανεώνει τις ετικέτες που ακολουθεί ο χρήστης και παρεμποδίζει την εκτέλεση σε περίπτωση ληγμένου session (σφάλμα αυθεντικοποίησης). Εάν ο χρήστης δεν ακολουθεί καμία ετικέτα ή αν είναι η πρώτη εκτέλεση,

καταγράφει την τρέχουσα ώρα και τερματίζει πρόωρα, αποφεύγοντας άσκοπες κλήσεις δικτύου.

2. **Στοχευμένη Ανάκτηση:** Ζητά από το `AnnouncementRepository` αυστηρά τις ανακοινώσεις που ανήκουν στις εγγεγραμμένες ετικέτες του χρήστη και που αναρτήθηκαν μετά την τελευταία επιτυχημένη αναζήτηση, αποτρέποντας διπλές ειδοποιήσεις.
3. **Ενημέρωση Κατάστασης:** Μετά την επιτυχή λήψη, ανανεώνει τη χρονική σφραγίδα για τον επόμενο κύκλο.

```
class CheckNewAnnouncementsUseCase @Inject constructor(
    private val tagsRepository: TagsRepository,
    private val announcementRepository: AnnouncementRepository,
    private val userPreferencesRepository: UserPreferencesRepository,
) {
    suspend operator fun invoke(): Result<List<Announcement>, NetworkError> {
        val refreshResult = tagsRepository.refreshSubscribedTags()
        if (refreshResult is NetworkError.Authentication) return refreshResult

        val subscribedTagIds = tagsRepository.getSubscribedTagsIds()
        val lastCheckedTime = userPreferencesRepository.getLastCheckTime()

        if (subscribedTagIds.isEmpty() || lastCheckedTime == null) {
            userPreferencesRepository.setLastCheckTime(Clock.System.now())
            return Result.Success(emptyList())
        }

        val announcementResult = announcementRepository.fetchAnnouncements(
            page = 1,
            perPage = 20,
            sortType = SortType.DESC,
            tagIds = subscribedTagIds,
            updatedAfter = lastCheckedTime
        )

        when (announcementResult) {
            is Result.Success -> {
                userPreferencesRepository.setLastCheckTime(Clock.System.now())
                return Result.Success(announcementResult.data)
            }
            is Result.Error -> {
                return announcementResult
            }
        }
    }
}
```

7.2.2 Εκτέλεση στο Παρασκήνιο

Η εκτέλεση της λογικής ανάκτησης στο παρασκήνιο ανατίθεται στον **AnnouncementAlertWorker**. Μέσω του annotation **@HiltWorker**, το Hilt εγχύει τις απαραίτητες εξαρτήσεις όπως γίνεται και με τα ViewModels στα προηγούμενα παραδείγματα, ενώ η χρήση του **CoroutineWorker** επιτρέπει την ασύγχρονη εκτέλεση μέσω Coroutines.

Ο Worker διατηρείται απλός, χωρίς πολύπλοκη λογική. Καλεί το **CheckNewAnnouncementsUseCase** και χειρίζεται το αποτέλεσμα. Ιδιαίτερη προσοχή έχει δοθεί στα σφάλματα αυθεντικοποίησης, όπου η εργασία τερματίζεται και ο χρήστης ειδοποιείται να συνδεθεί ξανά:

```
@HiltWorker
class AnnouncementAlertWorker @AssistedInject constructor(
    @Assisted context: Context,
    @Assisted workerParams: WorkerParameters,
    private val notifier: Notifier,
    private val crashlytics: Crashlytics,
    private val checkNewAnnouncementsUseCase: CheckNewAnnouncementsUseCase,
) : CoroutineWorker(context, workerParams) {

    override suspend fun doWork(): Result = try {
        when (val newAnnouncements = checkNewAnnouncementsUseCase()) {
            is Success -> {
                newAnnouncements.data.forEach { announcement ->
                    notifier.sendAnnouncementNotification(
                        announcementId = announcement.id,
                        title = announcement.title,
                        body = announcement.preview
                    )
                }
                return Result.success()
            }
            is Error -> {
                if (newAnnouncements.error is NetworkError.Authentication) {
                    notifier.sendGeneralNotification(
                        titleResId = R.string.title_logged_out,
                        bodyResId = R.string.body_logged_out,
                        uri = R.string.url_logged_out
                    )
                    return Result.failure()
                } else {
                    return Result.retry()
                }
            }
        }
    } catch (e: Exception) {
        crashlytics.recordException(e)
        return Result.retry()
    }
}
```

7.2.3 Χρονοπρογραμματισμός και Διαχείριση

Η διαδικασία του polling ενορχηστρώνεται κεντρικά μέσω του **WorkScheduler**. Εκεί καθορίζονται οι περιορισμοί εκτέλεσης, όπως η απαίτηση για ενεργή σύνδεση δικτύου και η πολιτική επανάληψης σε περίπτωση σφάλματος.

```
override fun scheduleAnnouncementAlerts(intervalMinutes: Int) {

    val request = PeriodicWorkRequestBuilder<AnnouncementAlertWorker>(
        repeatInterval = intervalMinutes.toLong(),
        repeatIntervalTimeUnit = TimeUnit.MINUTES
    )
```

```

        ).setConstraints(
            Constraints.Builder()
                .setRequiredNetworkType(NetworkType.CONNECTED)
                .build()
        ).setBackoffCriteria(
            backoffPolicy = BackoffPolicy.LINEAR,
            backoffDelay = 15,
            timeUnit = TimeUnit.MINUTES
        ).build()

        workManager.enqueueUniquePeriodicWork(
            uniqueWorkName = ANNOUNCEMENT_REFRESH_WORK_NAME,
            existingPeriodicWorkPolicy = ExistingPeriodicWorkPolicy.UPDATE,
            request = request
        )
    }

```

Για να προσφερθεί ευελιξία, ο χρήστης μπορεί να επιλέξει τη συχνότητα ελέγχου (από 15 λεπτά έως 12 ώρες) μέσα από την οθόνη των ρυθμίσεων (**feature:settings**). Η αλλαγή αυτής της προτίμησης εκτελείται μέσω του `SetAnnouncementCheckIntervalUseCase`, το οποίο αφενός αποθηκεύει τη νέα τιμή στο **Proto DataStore** (ώστε να ενημερωθεί το UI), και αφετέρου αναπρογραμματίζει άμεσα τον **Worker**.

```

class SetAnnouncementCheckIntervalUseCase @Inject constructor(
    private val workScheduler: WorkScheduler,
    private val userPreferencesRepository: UserPreferencesRepository
) {
    suspend operator fun invoke(minutes: Int) {
        workScheduler.scheduleAnnouncementAlerts(minutes)
        userPreferencesRepository.setAnnouncementCheckInterval(minutes)
    }
}

```

7.3 Βελτιστοποίηση Απόδοσης

7.3.1 Code Shrinking και Βελτιστοποίηση με R8

Ο **R8 compiler** [33] είναι το προεπιλεγμένο εργαλείο του Android που αναλαμβάνει τη συρρίκνωση (shrinking), τη βελτιστοποίηση (optimization) και την απόκρυψη (obfuscation) του παραγόμενου κώδικα.

Δεδομένου ότι η εφαρμογή διατίθεται ως ανοιχτού κώδικα, η απόκρυψη του κώδικα για την αποτροπή δεν αποτελεί ζητούμενο. Συνεπώς, το R8 δεν χρησιμοποιείται για απόκρυψη κώδικα, αλλά αξιοποιείται αποκλειστικά για τη μείωση του μεγέθους και τη βελτιστοποίηση της απόδοσης μέσω:

- **Code shrinking:** Αφαίρεση αχρησιμοποίητου κώδικα (κλάσεων, μεθόδων και πεδίων), μειώνοντας αισθητά το τελικό μέγεθος του APK.
- **Obfuscation:** Σύντμηση των ονομάτων κλάσεων, μεθόδων και μεταβλητών.

- **Optimization:** Στατική ανάλυση και βελτιστοποίηση του παραγόμενου bytecode για πιο αποδοτική και ταχύτερη εκτέλεση.

Οι κανόνες ProGuard (ProGuard rules) διατηρούνται στο ελάχιστο δυνατό. Βιβλιοθήκες όπως το Hilt και το Room παράγουν τον κώδικα μέσω **annotation processing** [34], τον οποίο ο R8 εντοπίζει και διατηρεί αυτόματα. Ωστόσο, κρίνεται απαραίτητη η προσθήκη συγκεκριμένων εξαιρέσεων [35] για τις διεπαφές του Retrofit και τις κλάσεις των Protocol Buffers (Proto). Καθώς οι τεχνολογίες αυτές βασίζονται στο **αντανάκλαση** (Reflection) κατά τον χρόνο εκτέλεσης, ο R8 δεν μπορεί να εντοπίσει στατικά τη χρήση τους και ενδέχεται να τις αφαιρέσει εσφαλμένα. Τέλος, χρησιμοποιείται η οδηγία **-assumenosideeffects** για την αυτόματη διαγραφή όλων των κλήσεων **Log.d**. Αυτό διασφαλίζει ότι τυχόν διαγνωστικά μηνύματα που μπορεί να παρέμειναν από τη φάση της ανάπτυξης θα αφαιρεθούν πλήρως από την τελική έκδοση της εφαρμογής.

```
-keepclassmembers class com.kastik.apps.core.datastore.proto.* { *; }
-keep interface com.kastik.apps.core.network.api.** { *; }

-assumenosideeffects class android.util.Log {
    public static int d(java.lang.String, java.lang.String);
}
```

7.3.2 Baseline Profile

Τα Baseline Profile [7] επιτρέπουν την προ-μεταγλώττιση (Ahead-of-Time Compilation) κρίσιμων μονοπατιών κώδικα κατά την εγκατάσταση, αυξάνοντας δραστικά την απόδοση. Χωρίς Baseline Profile, το Android Runtime [36] (ART) χρησιμοποιεί Just-in-Time Compilation εισάγοντας καθυστέρηση. Το module **benchmark** ορίζει τα κρίσιμα μονοπάτια μέσω BaselineProfileRule [37]. Το BaselineProfileRule είναι ένα test που «καταγράφει» ποια μέρη κώδικα εκτελούνται κατά τις βασικές ροές. Το baselineProfileRule.collect() εκτελεί την εφαρμογή και καταγράφει ακριβώς ποιες κλάσεις και μέθοδοι εκτελέστηκαν.

```
class SettingsBaselineProfile {

    @get:Rule
    val baselineProfileRule = BaselineProfileRule()

    @Test
    fun generate() = baselineProfileRule.collect(
        "com.kastik.apps",
    ) {
        launchAppAndDismissSigningDialog()
        navigateToSettings()
        toggleSortingOptions()
        toggleFabFilters()
        toggleSearchFieldOptions()
        toggleThemeOptions()
        toggleDynamicTheme()
    }
}
```

Το παραγόμενο αρχείο `baseline-prof.txt` συμπεριλαμβάνεται στο τελικό εκτελέσιμο αρχείο και κατά την εγκατάσταση από το Play Store, το ART προ-μεταγλωττίζει τα αντίστοιχα τμήματα bytecode.

7.3.3 Startup Profile

Τα **Startup Profile** λειτουργούν συμπληρωματικά με τα Baseline Profiles, αλλά εξυπηρετούν διαφορετικό σκοπό. Αν και οι δύο τεχνικές βασίζονται στην ιχνηλάτηση (tracing) κλήσεων μεθόδων, παρουσιάζουν δύο θεμελιώδεις διαφορές:

- **Εύρος Καταγραφής:** Κατά τη δημιουργία ενός Baseline Profile, η καταγραφή διατρέχει σχεδόν ολόκληρη την εφαρμογή για να συμπεριλάβει όλες τις κρίσιμες διαδρομές χρήστη (critical user journeys). Αντίθετα, στα Startup Profile ιχνηλατούνται αυστηρά μόνο οι μέθοδοι που εκτελούνται από τα σημεία εκκίνησης της εφαρμογής (π.χ. Launcher, Deep Links).
- **Χρήση και Διανομή:** Το Baseline Profile ενσωματώνεται στο τελικό εκτελέσιμο αρχείο, επιτρέποντας στο Android Runtime να εκτελέσει Ahead-of-Time (AOT) μεταγλώττιση στη συσκευή. Αντιθέτως, το Startup Profile **δεν** διανέμεται στους χρήστες. Χρησιμοποιείται αποκλειστικά από τον compiler για να βελτιστοποιήσει την τοποθεσία του κώδικα εκκίνησης στο τελικό αρχείο DEX [38], μειώνοντας δραστικά τον χρόνο φόρτωσης στη μνήμη.

Η δημιουργία του επιτυγχάνεται επίσης μέσω του `BaselineProfileRule`, περνώντας όμως την παράμετρο `includeInStartupProfile`. Το αποτέλεσμα αποθηκεύεται αυτόματα ως `startup-prof.txt`

```
class StartupProfile {
    @get:Rule
    val baselineProfileRule = BaselineProfileRule()

    @Test
    fun generateLauncherStartupProfile() = baselineProfileRule.collect(
        packageName = "com.kastik.apps",
        includeInStartupProfile = true,
    ) {
        uiAutomator {
            startActivityAndWait()
        }
    }

    @Test
    fun generateAnnouncementStartupProfile() = baselineProfileRule.collect(
        packageName = "com.kastik.apps",
        includeInStartupProfile = true,
    ) {
        val intent = Intent(Intent.ACTION_VIEW,
            ↪ "https://aboard.iee.ihu.gr/announcements/65163".toUri())
        intent.setPackage("com.kastik.apps")
        startActivityAndWait(intent)
    }
}
```

7.3.4 Macrobenchmark Tests

Το ίδιο **benchmark** module περιλαμβάνει macrobenchmark tests για μέτρηση χρόνου εκκίνησης. Κάθε test εκτελείται πολλαπλές φορές για αξιόπιστα αποτελέσματα. Η παράμετρος `CompilationMode.None()` απενεργοποιεί κάθε AOT. Αυτό χρησιμοποιείται ως βάση σύγκρισης χωρίς Baseline Profile. Το `CompilationMode.Partial()` χρησιμοποιεί AOT με Baseline Profile, ώστε να μετρηθεί η βελτίωση:

```
class StartupBenchmark {
    @get:Rule
    val benchmarkRule = MacrobenchmarkRule()

    @Test
    fun startupWithoutPreCompilation() = startup(CompilationMode.None())

    @Test
    fun startupPrecompiledWithBaselineProfile() =
        startup(CompilationMode.Partial(baselineProfileMode = Require))

    private fun startup(compilationMode: CompilationMode) =
        ↪ benchmarkRule.measureRepeated(
            packageName = "com.kastik.apps",
            metrics = listOf(StartupTimingMetric()),
            compilationMode = compilationMode,
            iterations = 20,
            startupMode = COLD,
            setupBlock = {
                pressHome()
            },
        ) {
            startActivityAndWait()
        }
}
```

7.4 Στρατηγική Δοκιμών

Η στρατηγική ελέγχου της εφαρμογής επικεντρώνεται στην εξασφάλιση της ορθότητας της διαχείρισης δεδομένων. Αποτελείται από unit tests που ελέγχουν τα Repositories, τους Mappers, καθώς και τη λειτουργία των τοπικών και απομακρυσμένων πηγών δεδομένων (Database, Network) σε πλήρη απομόνωση από το Android SDK. Χρησιμοποιούνται τα εξής βασικά εργαλεία:

- **MockK**: για τη δημιουργία «ψεύτικων» αντικειμένων (mocks).
- **runTest & TestDispatcher**: για την εκτέλεση coroutines σε ελεγχόμενο εικονικό χρόνο.
- **Google Truth**: για την πραγματοποίηση ξεκάθαρων και ευανάγνωστων (fluent) assertions.

Ακολουθεί ένα χαρακτηριστικό παράδειγμα δοκιμής από το Data layer. Εδώ επαληθεύεται η ορθή λειτουργία του `SortTypeConverter`, ο οποίος χρησιμοποιείται από τη βάση δεδομένων Room για τη μετατροπή των τύπων ταξινόμησης:

```
class SortTypeConverterTest {
    private val converter = SortTypeConverter()

    @Test
    fun mapsSortTypeToName() {
        SortType.entries.forEach { type ->
            val result = converter.fromSortType(type)
            assertThat(result).isEqualTo(type.name)
        }
    }
}
```

7.5 Σύνοψη

Το παρόν κεφάλαιο ολοκλήρωσε την παρουσίαση της εφαρμογής IEE IHU με τις οριζόντιες λειτουργίες που διαπερνούν τα τρία layers των Κεφαλαίων 4--6.

Ο **AnnouncementAlertWorker** αναπληρώνει αποδοτικά την απουσία push notifications από το Backend. Στον τομέα της απόδοσης, ο **R8 compiler** εξασφαλίζει το βέλτιστο μέγεθος και ταχύτητα εκτέλεσης, τα **Baseline Profile** και τα **Startup Profile** βελτιώνουν την απόδοση και την ταχύτητα εκκίνησης, η οποία αξιολογείται μέσω των **macrobenchmarks**, και τέλος τα unit tests επιβεβαιώνουν τη σωστή λειτουργία της εφαρμογής.

Έχοντας τώρα πλήρη εικόνα της αρχιτεκτονικής (Κεφάλαιο 3), Data Layer (Κεφάλαιο 4), Domain Layer (Κεφάλαιο 5), UI Layer (Κεφάλαιο 6) και οριζόντιες λειτουργίες, το Κεφάλαιο 8 παρουσιάζει τα αποτελέσματα αυτών των δοκιμών και μετρήσεων, αξιολογώντας πόσο καλά λειτουργούν αυτές οι επιλογές στην πράξη.

Κεφάλαιο 8

Αποτελέσματα

8.1 Εισαγωγή

Στα Κεφάλαια 3--7 παρουσιάστηκε η εφαρμογή IEE IHU από θεμέλια έως λεπτομέρεια: η αρχιτεκτονική βάση (Κεφάλαιο 3), το Data Layer (Κεφάλαιο 4), το Domain Layer (Κεφάλαιο 5), το UI Layer (Κεφάλαιο 6), και οι οριζόντιες λειτουργίες (Κεφάλαιο 7). Κάθε σχεδιαστική απόφαση ελήφθη με συγκεκριμένους στόχους: ταχύτητα, αξιοπιστία, επεκτασιμότητα και ελάχιστη κατανάλωση πόρων.

Το παρόν κεφάλαιο εξετάζει εάν και κατά πόσο αυτοί οι στόχοι επιτεύχθηκαν στην πράξη. Η αξιολόγηση οργανώνεται σε δύο άξονες:

1. **Λειτουργική επαλήθευση:** Επιβεβαίωση ότι τα βασικά σενάρια χρήσης λειτουργούν ορθά.
2. **Απόδοση :** Μετρήσεις χρόνου εκκίνησης με **Macrobenchmark**, πριν και μετά την εφαρμογή **Baseline** και **Startup Profile**.

Για τα **macrobenchmark tests** αξιοποιήθηκε η βιβλιοθήκη **Macrobenchmark** που ορίστηκε στο `module benchmark` (βλ. §7.3).

8.2 Λειτουργική Επαλήθευση

8.2.1 Στρατηγική Δοκιμών και Κάλυψη

Ενώ η στρατηγική δοκιμών που καθορίστηκε στο Κεφάλαιο 7 (§7.4) εστιάζει στην υλοποίηση, το παρόν κεφάλαιο παραθέτει και αναλύει τα τελικά αποτελέσματα.

8.2.2 Αποτελέσματα Unit Tests

Τα test εκτελούνται αυτοματοποιημένα τόσο σε τοπικό επίπεδο όσο και στο περιβάλλον συνεχούς ενσωμάτωσης (Continuous Integration) μέσω της πλατφόρμας **GitHub Actions**.

Οι δοκιμές αυτές επαληθεύουν την ορθότητα της λογικής στο Data Layer της εφαρμογής, εξασφαλίζοντας την αξιόπιστη διαχείριση, αποθήκευση και μετατροπή των δεδομένων. Συνολικά υλοποιήθηκαν 114 δοκιμές, οι οποίες κάλυψαν:

- **Δίκτυο (Network Module / `core:network`):** Επαλήθευση των Remote Data Sources, τη σωστή λειτουργία του `TokenInterceptor` και των `serializers`.
- **Βάση Δεδομένων (Database Module / `core:database`):** Δοκιμές στα DAOs και στους `Type Converters` για την ορθή αποθήκευση και ανάκτηση τοπικών δεδομένων.
- **Επίπεδο Δεδομένων (Data Module / `core:data`):** Επαλήθευση των Mappers για τη σωστή μετατροπή οντοτήτων από το δίκτυο και τη βάση σε domain models, καθώς και επαλήθευση των λειτουργιών των `Repositories` και του `RemoteMediator`.

Τα συγκεντρωτικά αποτελέσματα παρουσιάζονται στον Πίνακα 8.1.

Module	Πλήθος	Στόχος Επαλήθευσης	Περιβάλλον Εκτέλεσης	Ποσοστό Επιτυχίας
<code>core:network</code>	18	Remote Sources, Interceptors, Serializers	Local / GitHub CI	100%
<code>core:database</code>	23	DAOs, Room Converters, Entities	Local / GitHub CI	100%
<code>core:data</code>	73	Mappers, Repositories, RemoteMediator	Local / GitHub CI	100%
Σύνολο	114			100%

Πίνακας 8.1: Συγκεντρωτικά αποτελέσματα αυτοματοποιημένων δοκιμών ανά module

8.2.3 Σενάρια Χρήσης και Λειτουργική Ορθότητα

Πέρα από τις αυτοματοποιημένες δοκιμές στο επίπεδο των δεδομένων, η συμπεριφορά της εφαρμογής και η αλληλεπίδρασή της με το UI επαληθεύτηκαν μέσω χειροκίνητων δοκιμών. Οι δοκιμές αυτές πραγματοποιήθηκαν σε φυσική συσκευή προκειμένου να διασφαλιστεί η ορθή εκτέλεση υπό πραγματικές συνθήκες hardware.

Συγκεκριμένα, η εφαρμογή αξιολογήθηκε χειροκίνητα στα εξής επτά βασικά σενάρια χρήσης:

1. **Αρχική φόρτωση ανακοινώσεων :** Αξιολόγηση της συμπεριφοράς της εφαρμογής με και χωρίς διαθέσιμα τοπικά δεδομένα.
2. **Αναζήτηση και φιλτράρισμα :** Επαλήθευση της ορθότητας των άμεσων αποτελεσμάτων (Quick Results) και αυτών που επιστρέφονται από το Backend.
3. **Προβολή λεπτομερειών :** Επιλογή ανακοίνωσης και επιβεβαίωση της ορθής δρομολόγησης και λειτουργίας των deep links.
4. **Λήψη συνημμένων:** Επιβεβαίωση της ορθής αποθήκευσης αρχείων στη συσκευή.
5. **Εγγραφή σε ετικέτες :** Επαλήθευση του συγχρονισμού εγγραφών με το Backend και της άμεσης ενημέρωσης της προσωποποιημένης ροής «Εγγεγραμμένα».

6. **Ειδοποιήσεις και πλοήγηση:** Επιβεβαίωση ότι ο `AnnouncementAlertWorker` εντοπίζει επιτυχώς νέες ανακοινώσεις και ότι η αλληλεπίδραση με την ειδοποίηση μεταφέρει τον χρήστη στη σωστή ανακοίνωση.
7. **Αυθεντικοποίηση και διαχείριση συνεδρίας :** Επαλήθευση της επιτυχούς σύνδεσης, της τοπικής αποθήκευσης του JWT και της αυτόματης ενσωμάτωσής του στις κλήσεις δικτύου, καθώς και της ανανέωσής του σε περίπτωση λήξης.

Και τα επτά σενάρια ολοκληρώθηκαν επιτυχώς, το περιβάλλον και τα αναλυτικά αποτελέσματα των δοκιμών αποτυπώνονται στον Πίνακα 8.2.

#	Σενάριο Ελέγχου	Συσκευή Δοκιμής	Έκδοση Android	Αποτέλεσμα
1	Αρχική φόρτωση ανακοινώσεων	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
2	Αναζήτηση & Φιλτράρισμα	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
3	Προβολή λεπτομερειών & Deep Links	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
4	Λήψη και αποθήκευση συνημμένων	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
5	Εγγραφή σε ετικέτες και συγχρονισμός	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
6	Ειδοποιήσεις	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass
7	Αυθεντικοποίηση και Διαχείριση JWT	Google Pixel 8	Android 17 (QPR1 Beta 2)	Pass

Πίνακας 8.2: Αποτελέσματα χειροκίνητων δοκιμών σεναρίων χρήσης σε φυσικό hardware

8.2.4 Επαλήθευση Offline-First Συμπεριφοράς

Η **offline-first** προσέγγιση αξιολογήθηκε σε δύο σενάρια που αντικατοπτρίζουν τις πραγματικές συνθήκες χρήσης ενός κινητού τηλεφώνου: σταθερή σύνδεση και απώλεια σύνδεσης.

- **Πλήρης σύνδεση:** Η εφαρμογή ανακτά νέα δεδομένα από το Backend, τα αποθηκεύει στο Room και το UI ενημερώνεται δυναμικά.
- **Χωρίς σύνδεση (offline):** Το UI τροφοδοτείται αποκλειστικά από τα cached δεδομένα. Ο χρήστης μπορεί να περιηγηθεί στις λίστες <<Αρχική>>, <<Εγγεγραμμένα>>, να ανοίξει ανακοινώσεις, να κάνει αναζήτηση και να προβάλλει το προφίλ του.

Σκοπός της δοκιμής ήταν να επιβεβαιωθεί ότι η τοπική βάση παραμένει η μοναδική πηγή αλήθειας (Single Source of Truth) για το UI, ανεξαρτήτως διαθεσιμότητας δικτύου. Τα αποτελέσματα συνοψίζονται στον Πίνακα 8.3.

Κατάσταση	Σενάριο	Συσκευή Δοκιμής	Διαθέσιμες Λειτουργίες	Αποτέλεσμα
Online	Πλήρης σύνδεση	Google Pixel 8 Android 17 (QPR1 Beta 2)	Πλήρης συγχρονισμός, δυναμική ενημέρωση λιστών.	Pass
Offline	Απώλεια σύνδεσης	Google Pixel 8 Android 17 (QPR1 Beta 2)	Περιήγηση στις λίστες, προβολή λεπτομερειών, τοπική αναζήτηση.	Pass

Πίνακας 8.3: Αξιολόγηση συμπεριφοράς offline-first αρχιτεκτονικής

Και τα δύο σενάρια ανταποκρίθηκαν πλήρως στην αναμενόμενη συμπεριφορά, επιβεβαιώνοντας ότι η offline-first προσέγγιση παρέχει συνεκτική εμπειρία χρήστη, ανεξάρτητα από τη διαθεσιμότητα δικτύου.

8.3 Απόδοση: Μετρήσεις Macrobenchmark

8.3.1 Μεθοδολογία Μέτρησης

Οι δοκιμές απόδοσης εκτελέστηκαν σε φυσική συσκευή **Google Pixel 8**, με λειτουργικό σύστημα **Android 17 (QPR1 Beta 2)**, με $n = 20$ επαναλήψεις για τους χρόνους εκκίνησης και χρησιμοποιήθηκε η μετρική **StartupTimingMetric**, η οποία μετρά τον χρόνο από τη στιγμή που το σύστημα ξεκινά την εφαρμογή έως ότου η **πρώτη οθόνη** σχεδιαστεί πλήρως. Αυτός είναι ο χρόνος που αντιλαμβάνεται ο χρήστης από το πάτημα του εικονιδίου έως την εμφάνιση περιεχομένου. Για την εξαγωγή των μετρήσεων που αφορούσαν τη μη χρήση του Startup Profile, οι δοκιμές εκτελέστηκαν παραλείποντας το αρχείο `startup-prof.txt` κατά τη μεταγλώττιση.

8.3.2 Χρόνοι Εκκίνησης

Οι παρακάτω μετρήσεις εστιάζουν αποκλειστικά σε **Cold Start**, δηλαδή το άνοιγμα της εφαρμογής από το μηδέν, όταν αυτή δεν βρίσκεται ήδη στη μνήμη. Η **διάμεσος** (median) αντιπροσωπεύει την τυπική απόδοση αποφεύγοντας την επιρροή ακραίων τιμών, ενώ το **Max** καταγράφει τη χειρότερη δυνατή καθυστέρηση που παρατηρήθηκε.

Σενάριο	Min (ms)	Median (ms)	Max (ms)
Χωρίς Προφίλ	481.6	520.4	598.2
Μόνο Baseline Profile	462.8	487.8	532.1
Μόνο Startup Profile	482.9	505.3	544.1
Baseline Profile και Startup Profile	451.7	477.5	525.8

Πίνακας 8.4: Χρόνοι εκκίνησης ανά διαμόρφωση

Όπως παρατηρείται στον Πίνακα 8.4, ο συνδυασμός Baseline και Startup Profile αποδίδει τους βέλτιστους χρόνους εκκίνησης, μειώνοντας τη διάμεσο στα 477.5 ms έναντι των 520.4 ms (χωρίς προφίλ). Ενώ η μεμονωμένη χρήση οποιουδήποτε προφίλ προσφέρει μετρήσιμη βελτίωση, η συνδυαστική τους εφαρμογή εξασφαλίζει τη μέγιστη ταχύτητα, ελαχιστοποιώντας τόσο τον τυπικό όσο και τον μέγιστο (Max) χρόνο αναμονής του χρήστη.

8.4 Συζήτηση και Περιορισμοί

Τα αποτελέσματα παρουσιάζουν συνολικά θετική εικόνα. Ωστόσο, αξίζει να επισημανθούν ορισμένοι περιορισμοί.

Polling αντί Push Notifications. Ο `AnnouncementAlertWorker` παρέχει αξιόπιστες ειδοποιήσεις, αλλά εισάγει καθυστέρηση ανάλογη της ρυθμιζόμενης συχνότητας. Εάν το Backend αναβαθμιστεί με υποστήριξη FCM, η αντικατάσταση δεν θα απαιτεί αλλαγές στο UI ή στο domain layer, χάρη στην αποσύζευξη που παρέχει το `core:notifications` module.

Μετρήσεις σε περιορισμένο hardware. Τα macrobenchmark tests πραγματοποιήθηκαν σε μια μόνο συσκευή. Σε χαμηλού κόστους συσκευές με λιγότερους πόρους, οι απόλυτοι χρόνοι εκκίνησης μπορεί να είναι υψηλότεροι, αν και η σχετική βελτίωση από τα Baseline Profile και Startup Profile αναμένεται να διατηρηθεί.

Αναζήτηση: τοπική vs. Backend. Η υβριδική αναζήτηση παρέχει γρήγορα quick results, αλλά η πληρότητά τους περιορίζεται στα ήδη cached δεδομένα. Πολύ πρόσφατες ανακοινώσεις που δεν έχουν ακόμα συγχρονιστεί τοπικά ενδέχεται να απουσιάζουν από τα quick results κάτι που αντισταθμίζεται από τα αποτελέσματα Backend.

Έλλειψη υποστήριξης λειτουργικού συστήματος iOS. Η εφαρμογή δεν καλύπτει χρήστες iOS, οι οποίοι αντιπροσωπεύουν σημαντικό τμήμα του φοιτητικού πληθυσμού. Αν και το Android κατέχει τη μεγαλύτερη μερίδα αγοράς παγκοσμίως [39], η απουσία iOS εφαρμογής αποτελεί πρακτικό περιορισμό.

Λειτουργία αποκλειστικά για ανάγνωση Η εφαρμογή λειτουργεί αποκλειστικά ως εργαλείο προβολής ανακοινώσεων. Δεν παρέχεται δυνατότητα δημιουργίας ή επεξεργασίας ανακοινώσεων από καθηγητές ή διαχειριστές μέσω της κινητής εφαρμογής.

Αποκλειστική εξάρτηση από ένα κεντρικό σύστημα Η εφαρμογή εξαρτάται από ένα συγκεκριμένο Backend σύστημα. Η διαθεσιμότητα και η λειτουργία της εξαρτώνται άμεσα από τη λειτουργία αυτού του συστήματος, χωρίς εναλλακτική πηγή δεδομένων.

8.5 Σύνοψη

Στο παρόν κεφάλαιο αξιολογήθηκε η εφαρμογή IEE IHU σε δύο επίπεδα: λειτουργική επαλήθευση και απόδοση.

Τα **αυτοματοποιημένα** και **χειροκίνητα** test επιβεβαίωσαν την ορθή λειτουργία της εφαρμογής. Τα **macrobenchmark αποτελέσματα** έδειξαν βελτίωση Cold Start κατά **8.24%** με Baseline και Startup Profile.

Συνολικά, η εφαρμογή ανταποκρίνεται στους στόχους που τέθηκαν κατά τον αρχιτεκτονικό σχεδιασμό, επιβεβαιώνοντας την καταλληλότητα του **Modern Android Development** για την ανάπτυξη αξιόπιστων, αποδοτικών και χρήσιμων εφαρμογών σε ακαδημαϊκό περιβάλλον.

Στο Κεφάλαιο 9 παρουσιάζονται τα τελικά συμπεράσματα: καθώς έχει τεκμηριωθεί η ορθή και αποδοτική λειτουργία της εφαρμογής, στο επόμενο κεφάλαιο αναλύεται η συνολική της συνεισφορά και προτείνονται μελλοντικές επεκτάσεις.

Κεφάλαιο 9

Συμπεράσματα και Μελλοντική Εργασία

9.1 Συμπεράσματα

Στο πλαίσιο της παρούσας εργασίας αναπτύχθηκε μια εφαρμογή για το λειτουργικό σύστημα Android, η οποία επιτρέπει στους φοιτητές του Τμήματος Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙΠΑΕ να έχουν άμεση πρόσβαση σε πανεπιστημιακές ανακοινώσεις, χωρίς να απαιτείται περιοδική επίσκεψη σε ιστοσελίδες που δεν είναι βελτιστοποιημένες για κινητές συσκευές.

Η ανάπτυξη της εφαρμογής IEE IHU ανταποκρίθηκε πλήρως στους στόχους που τέθηκαν στο Κεφάλαιο 1. Η εφαρμογή υλοποιεί ένα πλήρες σύνολο λειτουργιών: προβολή και σταδιακή φόρτωση ανακοινώσεων, υβριδική αναζήτηση, φιλτράρισμα ανά ετικέτα ή καθηγητή, εγγραφή σε ετικέτες, λήψη ειδοποιήσεων και αυθεντικοποίηση μέσω JWT. Η δημοσίευσή της στο Google Play Store και η διάθεσή της ως αποθετήριο ανοιχτού κώδικα επιτρέπουν τη χρήση από πραγματικούς χρήστες και τη μελέτη της υλοποίησης από την κοινότητα.

Η εφαρμογή κατασκευάστηκε αποκλειστικά με τεχνολογίες του πλαισίου **Modern Android Development**: Kotlin, Jetpack Compose, MVVM + Clean Architecture, Hilt, Room, Paging 3, WorkManager και Proto DataStore. Από τεχνική σκοπιά, τα αποτελέσματα αξιολόγησης επιβεβαιώνουν ποσοτικά την επιτυχία των τριών βασικών αρχιτεκτονικών στοιχείων:

- **Απόδοση:** Τα Baseline και Startup Profile μείωσαν μετρήσιμα τον χρόνο Cold Start κατά **8.24%** (από ~520 ms σε ~477 ms).
- **Αξιοπιστία εκτός σύνδεσης:** Η offline-first αρχιτεκτονική επιτρέπει την εμφάνιση δεδομένων σε όλα τα σενάρια συνδεσιμότητας.
- **Ορθότητα και Απομόνωση:** Η multi-module δομή επέτρεψε τη συστηματική δοκιμή. Το 100% των unit tests στο Data Layer (Repositories, DAOs, Mappers, Data Sources) πέρασε επιτυχώς.

Η πιο σημαντική τεχνική αδυναμία, δηλαδή ο μηχανισμός polling αντί για push notifications, δεν αποτελεί αρχιτεκτονικό ελάττωμα, αλλά περιορισμό που επιβλήθηκε από το υπάρχον Backend. Η αποσύζευξη μέσω του `core:notifications` module έχει ήδη ετοιμαστεί ώστε αυτός ο περιορισμός να αρθεί μελλοντικά χωρίς ανασχεδιασμό.

Συνολικά, η εργασία αποδεικνύει ότι το πλαίσιο **Modern Android Development** επιτρέπει την ανάπτυξη εφαρμογών υψηλής απόδοσης και αξιόπιστης offline λειτουργίας, με την προϋπόθεση ότι οι αρχιτεκτονικές επιλογές γίνονται με γνώμονα συγκεκριμένους, μετρήσιμους στόχους. Για την πληρέστερη εικόνα των δυνατοτήτων και του τρόπου λειτουργίας της εφαρμογής από την πλευρά του τελικού χρήστη, παρατίθεται αναλυτικό εγχειρίδιο χρήσης στο Παράρτημα Α.

9.2 Συνεισφορά της Εργασίας

Η παρούσα εργασία συμβάλλει τόσο σε πρακτικό όσο και σε τεχνολογικό επίπεδο.

Σε πρακτικό επίπεδο, η εφαρμογή αντιμετωπίζει ένα συγκεκριμένο πρόβλημα που αντιμετωπίζουν οι φοιτητές του ΔΙΠΑΕ: την έλλειψη αποδοτικού, mobile-first τρόπου πρόσβασης στις πανεπιστημιακές ανακοινώσεις. Η ενσωμάτωση προσωποποιημένων ειδοποιήσεων ανά ετικέτα μειώνει περαιτέρω τον «θόρυβο» πληροφοριών, επιτρέποντας στον κάθε φοιτητή να λαμβάνει μόνο τις ανακοινώσεις που τον αφορούν.

Σε τεχνολογικό επίπεδο, η εργασία αποτελεί μια ολοκληρωμένη μελέτη περίπτωσης (case study) για τη δημιουργία εφαρμογής Android με multi-module δομή, αποδεικνύοντας στην πράξη:

- Τη μείωση του επαναλαμβανόμενου κώδικα Gradle και τη βελτίωση των χρόνων μεταγλώττισης μέσω της multi-module δομής με convention plugins.
- Την επίτευξη offline-first συμπεριφοράς με ελάχιστη πολυπλοκότητα στο UI μέσω της συνδυαστικής χρήσης Paging 3 + RemoteMediator + Room.
- Την ομαλή ενσωμάτωση των Baseline και Startup Profile στη ροή ανάπτυξης μέσω του module benchmark και την ποσοτική μέτρηση της βελτίωσης απόδοσής τους.
- Την απλοποίηση της διαχείρισης κατάστασης και την ευκολία δοκιμής των ViewModels μέσω του μοτίβου sealed class UiState + StateFlow + UDF.

Η εφαρμογή αποτελεί επίσης ένα αξιόπιστο σημείο αναφοράς για μελλοντικές εφαρμογές παρόμοιας κλίμακας. Η αρχιτεκτονική της άντλησε στοιχεία από το **Now in Android** [40], την επίσημη εφαρμογή αναφοράς (reference app) της Google, η οποία αποτελεί τον επίσημο οδηγό αρχιτεκτονικής.

9.3 Μελλοντική Εργασία

Υπάρχουν αρκετές κατευθύνσεις για τη μελλοντική επέκταση και βελτίωση της εφαρμογής.

- **Υποστήριξη iOS μέσω Kotlin & Compose Multiplatform:** Η τρέχουσα υλοποίηση καλύπτει μόνο Android. Η τεχνολογία **Kotlin Multiplatform (KMP)** [41] επιτρέπει τη συγγραφή της βασικής λογικής της εφαρμογής (Domain και Data layer) σε Kotlin. Επιπλέον, με τη χρήση του **Compose Multiplatform** [42], ο διαμοιρασμός του κώδικα μπορεί να επεκταθεί και στο επίπεδο της διεπαφής χρήστη (UI Layer). Αυτή η προσέγγιση θα επέκτεινε την εφαρμογή στους φοιτητές που χρησιμοποιούν iOS και θα μπορούσε ακόμα να υποστηρίξει και δημιουργία εφαρμογής για Desktop.

- **Ενσωμάτωση Firebase Cloud Messaging:** Το **Firebase Cloud Messaging** [31] είναι μια υπηρεσία της Google που επιτρέπει σε διακομιστές να αποστέλλουν ειδοποιήσεις άμεσα στις συσκευές των χρηστών. Αντί ο `AnnouncementAlertWorker` να ελέγχει περιοδικά αν υπάρχουν νέες ανακοινώσεις, ο διακομιστής θα στέλνει αυτόματα ειδοποίηση στη συσκευή μόλις δημοσιευτεί νέα ανακοίνωση, μειώνοντας την κατανάλωση μπαταρίας και την καθυστέρηση.
- **Λειτουργίες δημιουργίας ανακοινώσεων:** Με την προσθήκη κατάλληλου συστήματος διαχείρισης δικαιωμάτων, ώστε μόνο εξουσιοδοτημένοι χρήστες να μπορούν να δημιουργούν ανακοινώσεις, η εφαρμογή θα μετατρέποταν σε πλήρες εργαλείο διαχείρισης. Αυτό θα απαιτούσε νέα feature modules και νέα Backend endpoints με υποστήριξη HTTP POST/PUT.
- **Εξατομικευμένες Συστάσεις:** Ένα **σύστημα συστάσεων** (recommendation system) θα μπορούσε να αναλύσει το ιστορικό πλοήγησης και τις ενεργές συνδρομές του χρήστη για να αναδείξει ανακοινώσεις αυξημένης σχετικότητας στην καρτέλα «Εγγεγραμμένα», παρόμοια με τους αλγόριθμους σύγχρονων ψηφιακών πλατφορμών.

Βιβλιογραφία

- [1] Google, *Modern Android Development*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/modern-android-development>
- [2] Google, *Jetpack Compose*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/jetpack/compose>
- [3] Microsoft, *The MVVM Pattern*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>
- [4] Google, *Save data in a local database using Room*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/training/data-storage/room>
- [5] Google, *Paging 3 library overview*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/paging/v3-overview>
- [6] Google, *Schedule tasks with WorkManager*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/workmanager>
- [7] Google, *Baseline Profiles*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/performance/baselineprofiles>
- [8] Google, *Android's Kotlin-first approach*. επίσκεψη 12 Μαρ. 2026. διεύθυν.: <https://developer.android.com/kotlin/first>
- [9] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017, isbn: 9780134494166.
- [10] Google, *Dependency injection with Hilt*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/training/dependency-injection/hilt-android>
- [11] Google, *Guide to Android app modularization*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/modularization>
- [12] Google, *Macrobenchmark library*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/performance/benchmarking/macrobenchmark-overview>
- [13] Google, *Guide to app architecture*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/architecture>
- [14] Google, *Flutter*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://flutter.dev/>
- [15] Meta, *React Native*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://reactnative.dev/>

- [16] Google, *Load and display paged data from network and database*. επίσκεψη 20 Μάι. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/paging/v3-network-db>
- [17] JetBrains, *Null safety*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://kotlinlang.org/docs/null-safety.html>
- [18] JetBrains, *Coroutines guide*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://kotlinlang.org/docs/coroutines-guide.html>
- [19] JetBrains, *Asynchronous Flow*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://kotlinlang.org/docs/flow.html>
- [20] Google, *StateFlow and SharedFlow*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/kotlin/flow/stateflow-and-sharedflow>
- [21] Google, *ViewModel overview*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/viewmodel>
- [22] Google, *UI layer*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/architecture/ui-layer>
- [23] Google, *Domain layer*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/architecture/domain-layer>
- [24] Google, *Data layer*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/architecture/data-layer>
- [25] Google, *State and Jetpack Compose*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/develop/ui/compose/architecture>
- [26] Google, *Navigate with Compose*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/develop/ui/compose/navigation>
- [27] Square, *Retrofit Introduction*. επίσκεψη 24 Μαρ. 2026. διεύθυν.: <https://square.github.io/retrofit/>
- [28] Google, *Page from network and database*. επίσκεψη 17 Απρ. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/views/paging/v3-network-db-views>
- [29] Google, *DataStore*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/topic/libraries/architecture/datastore>
- [30] Google, *Material Design 3*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://m3.material.io/>
- [31] Google, *Firebase Cloud Messaging*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://firebase.google.com/docs/cloud-messaging>
- [32] Google, *Choose relationship types between objects*. επίσκεψη 16 Μάι. 2026. διεύθυν.: <https://developer.android.com/training/data-storage/room/relationships>
- [33] Google, *Shrink, obfuscate, and optimize your app*. επίσκεψη 20 Μαρ. 2026. διεύθυν.: <https://developer.android.com/build/shrink-code>

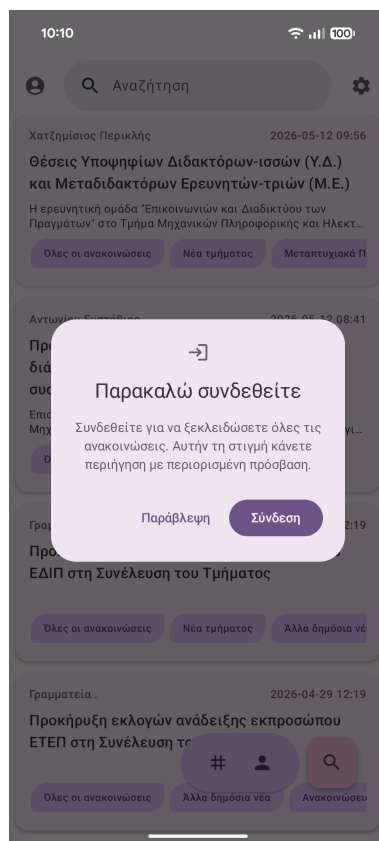
- [34] JetBrains, *KSP Overview*. επίσκεψη 19 Μάι. 2026. διεύθν.: <https://kotlinlang.org/docs/ksp-overview.html>
- [35] Google, *About Keep Rules*. επίσκεψη 19 Μάι. 2026. διεύθν.: <https://developer.android.com/topic/performance/app-optimization/keep-rules-overview>
- [36] Google, *Android runtime and Dalvik*. επίσκεψη 20 Μάι. 2026. διεύθν.: <https://source.android.com/docs/core/runtime>
- [37] Google, *App startup time*. επίσκεψη 20 Μαρ. 2026. διεύθν.: <https://developer.android.com/topic/performance/vitals/launch-time>
- [38] Google, *Dalvik Executable Format*. επίσκεψη 19 Μάι. 2026. διεύθν.: <https://source.android.com/docs/core/runtime/dex-format>
- [39] StatCounter, *Mobile Operating System Market Share Worldwide*. επίσκεψη 20 Μάι. 2026. διεύθν.: <https://gs.statcounter.com/os-market-share/mobile/worldwide>
- [40] Google, *Now in Android*. επίσκεψη 20 Μαρ. 2026. διεύθν.: <https://github.com/android/nowinandroid>
- [41] JetBrains, *Kotlin Multiplatform*. επίσκεψη 20 Μαρ. 2026. διεύθν.: <https://www.jetbrains.com/kotlin-multiplatform/>
- [42] JetBrains, *Compose Multiplatform*. επίσκεψη 20 Μάι. 2026. διεύθν.: <https://kotlinlang.org/compose-multiplatform/>

Παράρτημα Α

Εγχειρίδιο χρήσης

Α.1 Εκκίνηση Εφαρμογής και Ταυτοποίηση

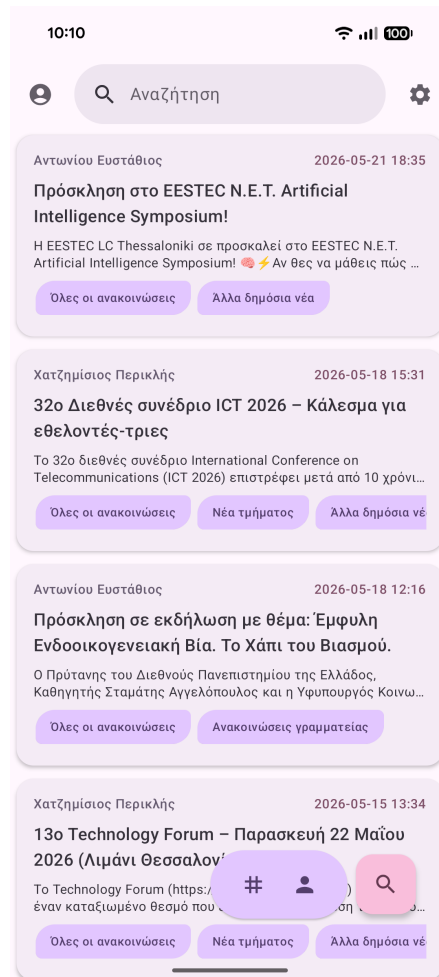
Η διαδικασία εισόδου στην εφαρμογή προσφέρει την επιλογή σύνδεσης ή χρήσης ως επισκέπτη. Κατά την πρώτη εκκίνηση της εφαρμογής, εμφανίζεται ένα αναδυόμενο παράθυρο σύνδεσης, το οποίο ο χρήστης μπορεί να παραλείψει.



Σχήμα Α.1: Αναδυόμενο παράθυρο προτροπής για σύνδεση

A.1.1 Περιήγηση ως Επισκέπτης

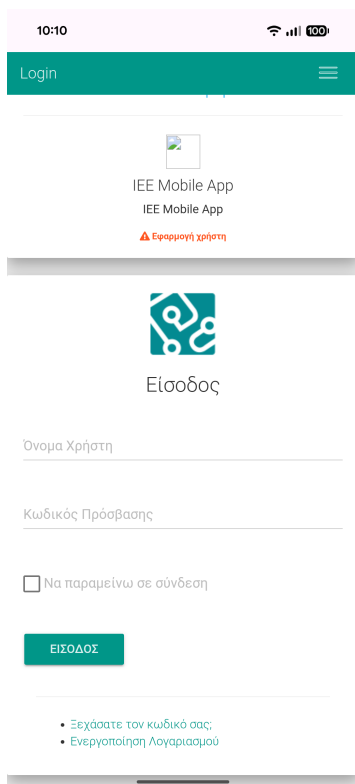
Εάν ο χρήστης επιλέξει να παραλείψει τη σύνδεση, προβάλλεται η αρχική οθόνη, η οποία περιλαμβάνει μόνο τις δημόσιες ανακοινώσεις, ενώ η προσωποποιημένη καρτέλα «Εγγεγραμμένα» δεν είναι διαθέσιμη.



Σχήμα Α.2: Αρχική οθόνη μη συνδεδεμένου χρήστη

A.1.2 Σύνδεση στον Λογαριασμό

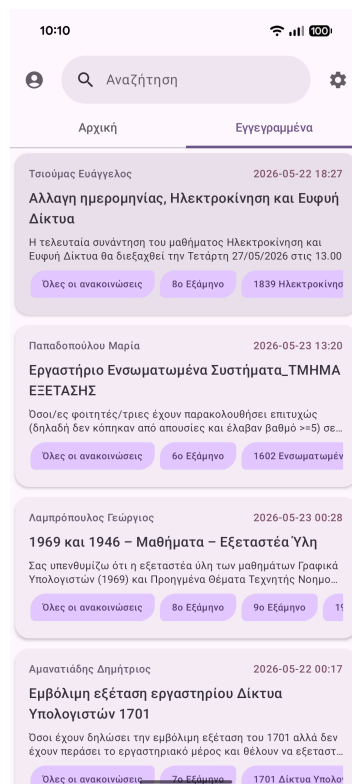
Με την επιλογή της «Σύνδεσης», ανοίγει η ιστοσελίδα ταυτοποίησης στον προεπιλεγμένο περιηγητή. Μετά την επιτυχή ταυτοποίηση και κατά την επιστροφή στην εφαρμογή, προβάλλεται μια οθόνη φόρτωσης. Πλέον, ο χρήστης έχει πρόσβαση στις ιδιωτικές ανακοινώσεις και μπορεί να επιλέξει την προσωποποιημένη καρτέλα «Εγγεγραμμένα».



Σχήμα Α.3: Οθόνη πιστοποίησης



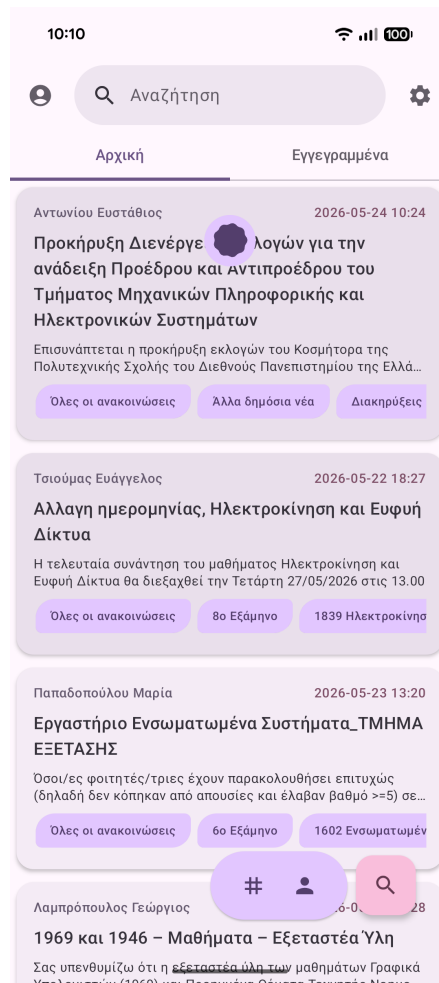
Σχήμα Α.4: Οθόνη φόρτωσης



Σχήμα Α.5: Οθόνη «Εγγεγραμμένα»

Α.1.3 Ανανέωση Περιεχομένου

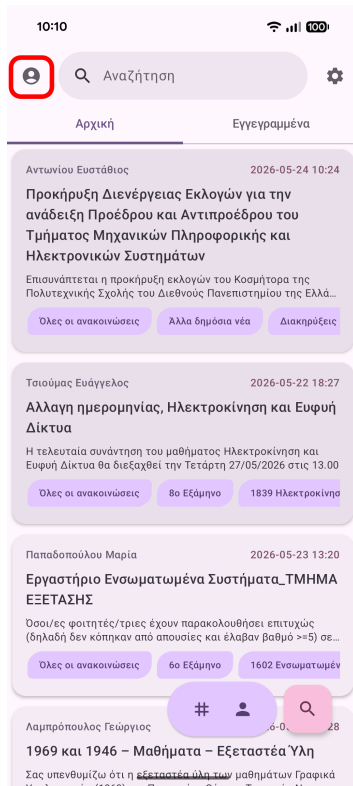
Η ανανέωση του περιεχομένου πραγματοποιείται αυτόματα κατά το άνοιγμα της εφαρμογής, καθώς και χειροκίνητα μέσω *pull-to-refresh*.



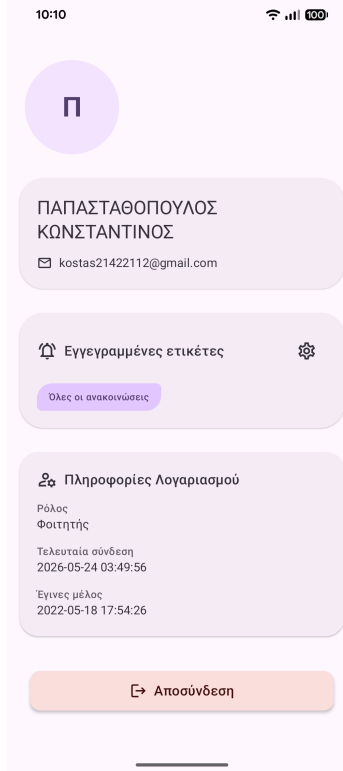
Σχήμα Α.6: Ανανέωση περιεχομένου με κίνηση pull-to-refresh

Α.2 Προφίλ Χρήστη και Διαχείριση Ενδιαφερόντων

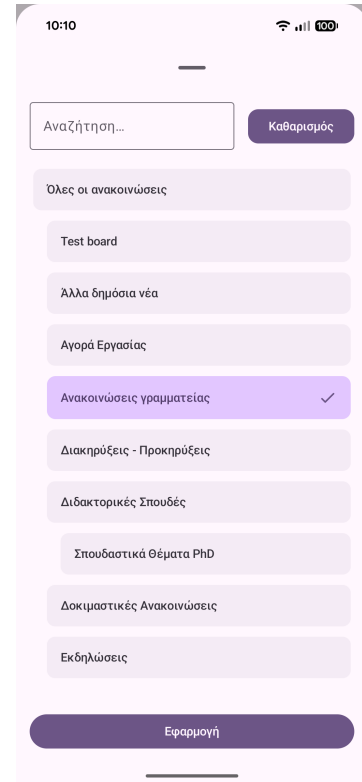
Ο συνδεδεμένος χρήστης μπορεί να προβάλει τα στοιχεία του και να διαχειριστεί τις ειδοποιήσεις που επιθυμεί να λαμβάνει. Πατώντας το εικονίδιο των ρυθμίσεων στην ενότητα «Εγγεγραμμένες Ετικέτες», αναδύεται το μενού επιλογών, από όπου ο χρήστης μπορεί να επιλέξει τις ετικέτες που τον ενδιαφέρουν.



Σχήμα Α.7: Πρόσβαση στο Προφίλ



Σχήμα Α.8: Προφίλ χρήστη

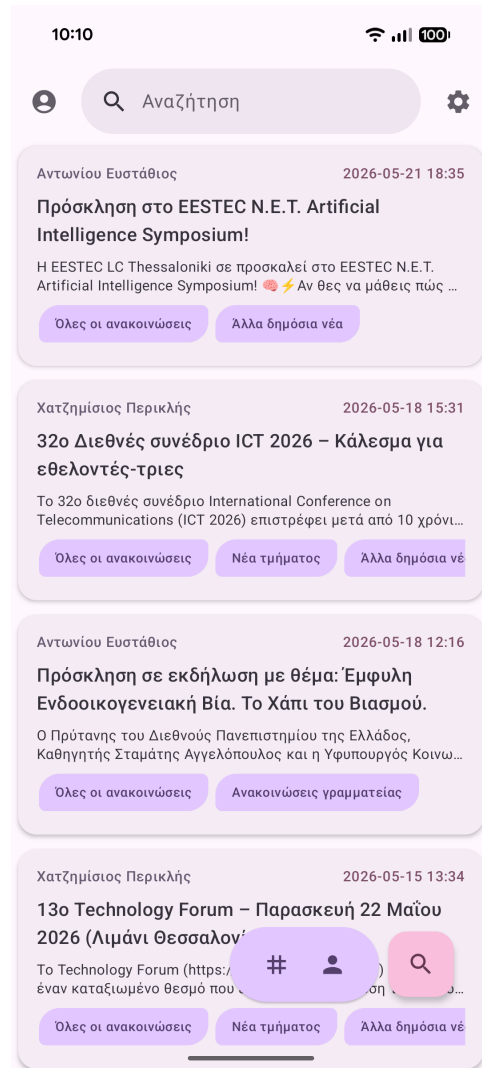


Σχήμα Α.9: Μενού διαχείρισης

Επιπλέον, μέσω της οθόνης του Προφίλ, ο χρήστης έχει τη δυνατότητα να αποσυνδεθεί από τον λογαριασμό του. Πατώντας τη σχετική επιλογή, η τρέχουσα συνεδρία τερματίζεται και η εφαρμογή επιστρέφει σε κατάσταση επισκέπτη.



Σχήμα Α.10: Αποσύνδεση χρήστη



Σχήμα Α.11: Αρχική οθόνη επισκέπτη

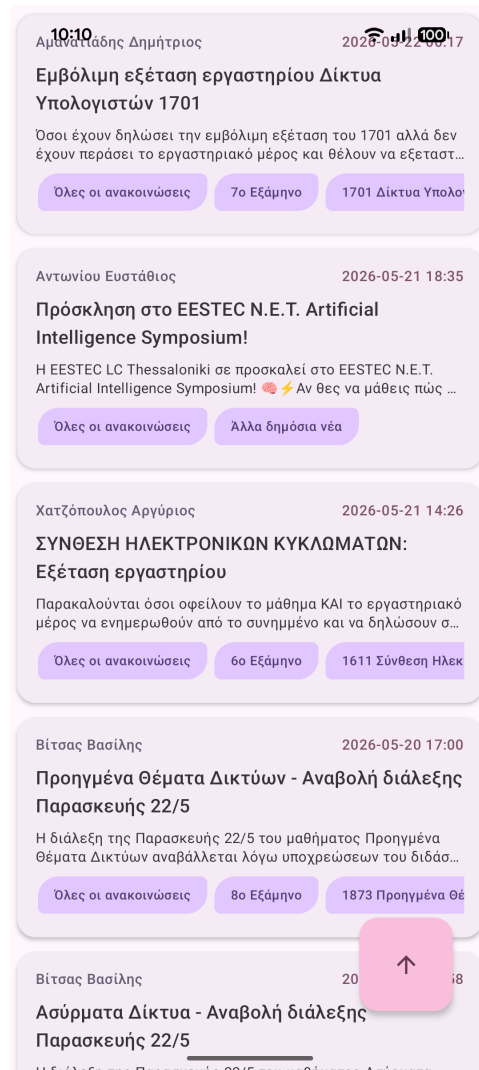
Α.3 Πλοήγηση και Αναζήτηση Ανακοινώσεων

Α.3.1 Προβολή Αρχικής Σελίδας

Υπάρχουν δύο κύριες ροές ανακοινώσεων: η «Αρχική» και τα «Εγγεγραμμένα». Καθώς ο χρήστης πραγματοποιεί κύλιση (*scroll*) στην Αρχική λίστα ανακοινώσεων, το αιωρούμενο κουμπί ενεργειών (FAB) αναπτύσσεται και αποκρύπτει τα φίλτρα. Με το πάτημα του FAB, η λίστα μεταβαίνει αυτόματα στην κορυφή.



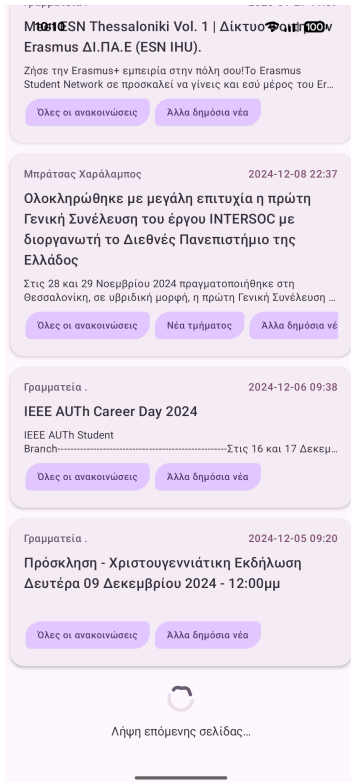
Σχήμα Α.12: Συμπτυγμένο FAB



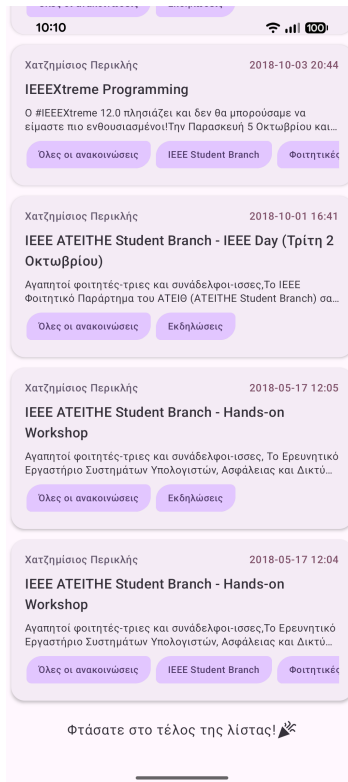
Σχήμα Α.13: Αναπτυγμένο FAB

Α.3.2 Δυναμική Φόρτωση Αποτελεσμάτων

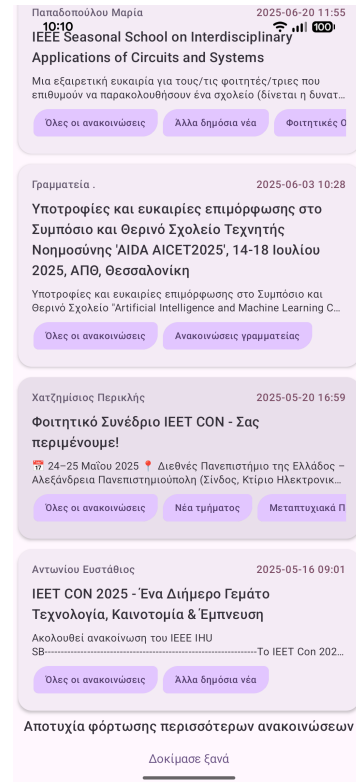
Οι χρήστες μπορούν να περιηγηθούν σε επόμενες σελίδες κάνοντας κύλιση προς τα κάτω, όπου και εμφανίζεται το σχετικό μήνυμα φόρτωσης. Σε περίπτωση που ο χρήστης φτάσει στο τέλος της λίστας, εμφανίζεται σχετικό ενημερωτικό μήνυμα. Σε περίπτωση αποτυχίας φόρτωσης, εμφανίζεται αντίστοιχο μήνυμα σφάλματος με κουμπί επανάληψης.



Σχήμα Α.14: Φόρτωση επιπλέον ανακοινώσεων



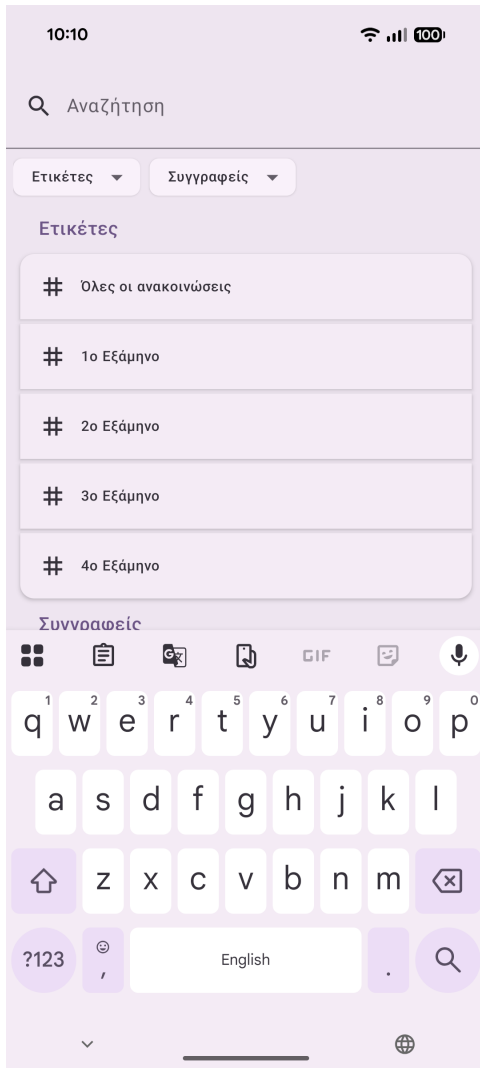
Σχήμα Α.15: Μήνυμα τέλους



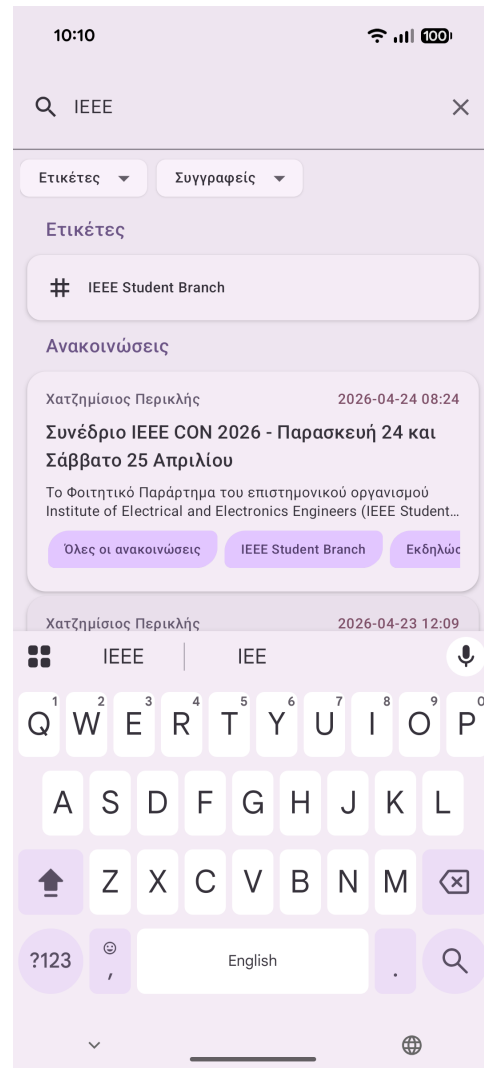
Σχήμα Α.16: Μήνυμα αποτυχίας

Α.3.3 Αναζήτηση

Επιλέγοντας τη γραμμή αναζήτησης, εμφανίζεται μια λίστα με αποτελέσματα. Κατά την πληκτρολόγηση, τα αποτελέσματα φιλτράρονται δυναμικά βάσει του κειμένου και περιλαμβάνουν Ετικέτες, Συγγραφείς και Ανακοινώσεις.



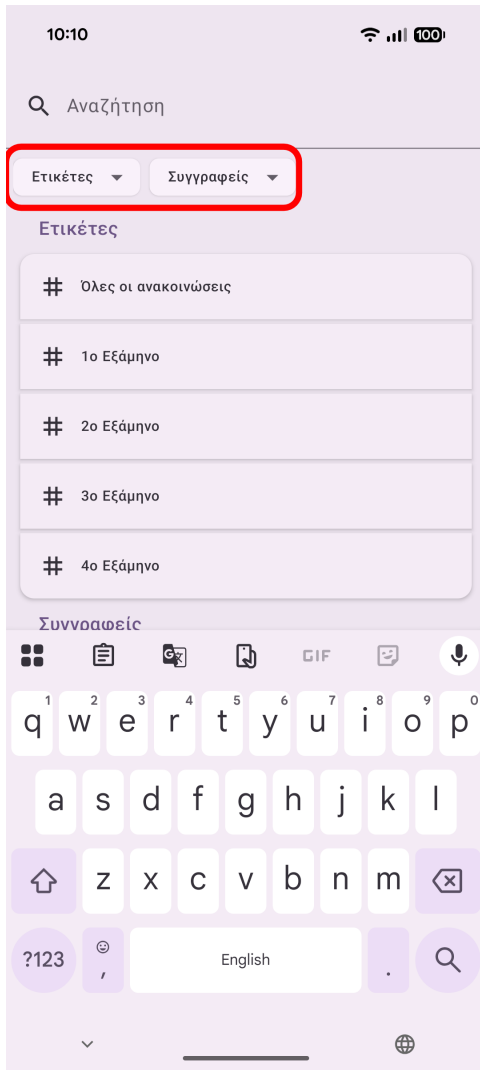
Σχήμα Α.17: Αναπτυγμένη γραμμή αναζήτησης με προτεινόμενα αποτελέσματα



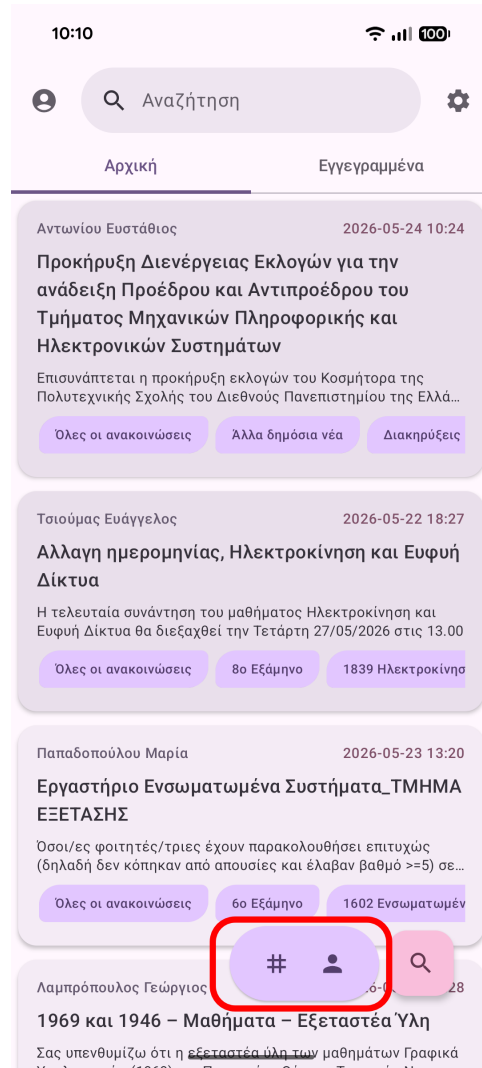
Σχήμα Α.18: Φιλτράρισμα αποτελεσμάτων κατά την πληκτρολόγηση

Α.3.4 Εφαρμογή Φίλτρων

Διατίθενται επίσης πλαίσια φίλτρων για Ετικέτες και Συγγραφείς. Αυτά είναι προσβάσιμα από τη γραμμή αναζήτησης, καθώς και δίπλα από το κουμπί FAB.



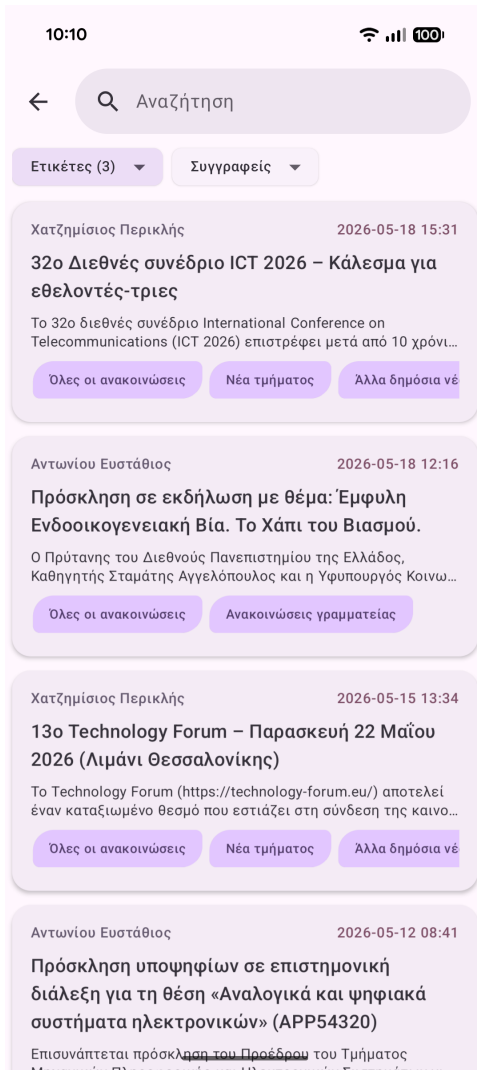
Σχήμα Α.19: Κουμπιά πρόσβασης στα μενού φίλτρων μέσω της γραμμής αναζήτησης



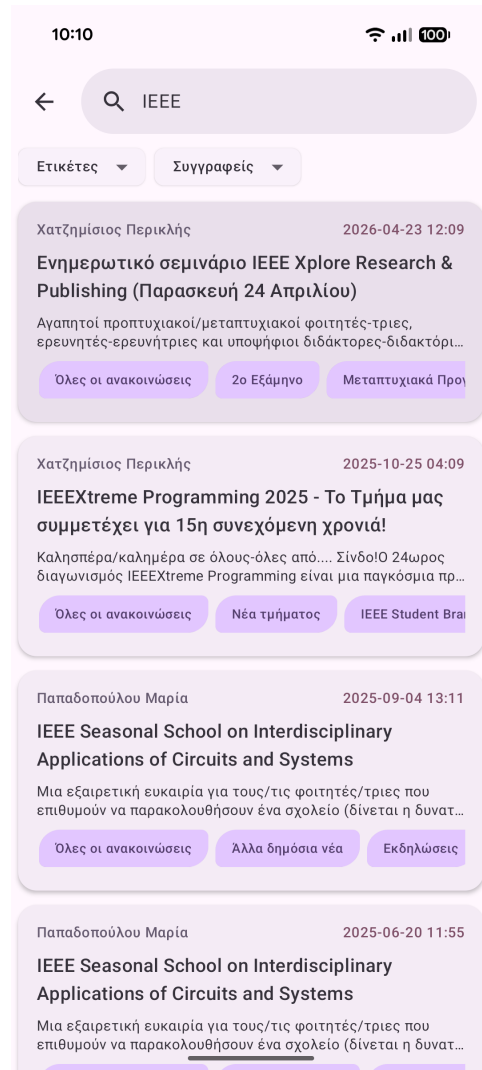
Σχήμα Α.20: Κουμπιά πρόσβασης στα φίλτρα δίπλα από το κουμπί FAB

Α.3.5 Προβολή Αποτελεσμάτων

Μετά την υποβολή της αναζήτησης, φορτώνεται η οθόνη των αποτελεσμάτων. Τα ενεργά φίλτρα εμφανίζονται κάτω από τη γραμμή αναζήτησης, ενώ το ενεργό κείμενο αναζήτησης διατηρείται μέσα σε αυτήν.

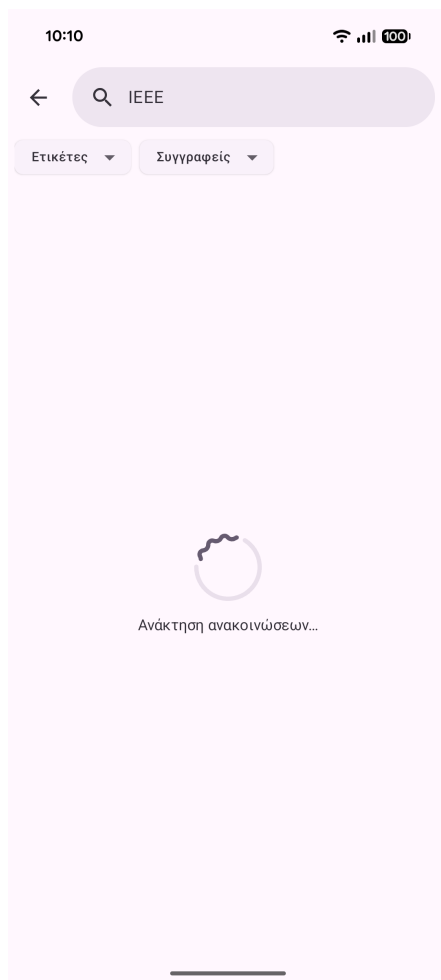


Σχήμα Α.21: Ενεργά φίλτρα ετικετών



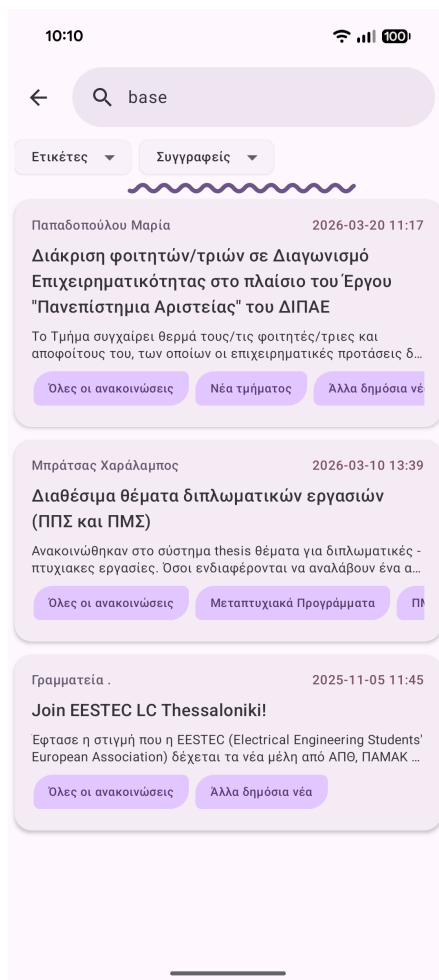
Σχήμα Α.22: Ενεργή αναζήτηση κειμένου

Κατά την πρώτη αναζήτηση βάσει κάποιου φίλτρου, προβάλλεται μια οθόνη φόρτωσης.



Σχήμα Α.23: Οθόνη φόρτωσης αρχικών αποτελεσμάτων αναζήτησης

Σε περίπτωση που υπάρχουν ήδη δεδομένα για την αναζήτηση, αυτά εμφανίζονται αμέσως. Ταυτόχρονα, εμφανίζεται ένας δείκτης προόδου υποδεικνύοντας ότι τα αποτελέσματα ανανεώνονται.



Σχήμα Α.24: Ενημέρωση αποθηκευμένων αποτελεσμάτων αναζήτησης

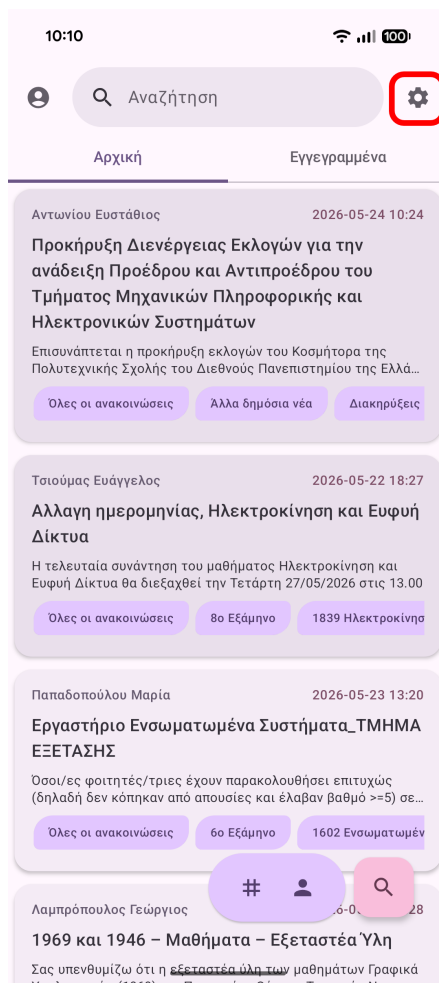
Τέλος, σε περίπτωση απουσίας αποτελεσμάτων βάσει των κριτηρίων που τέθηκαν, προβάλλεται σχετικό ενημερωτικό μήνυμα.



Σχήμα Α.25: Μήνυμα απουσίας αποτελεσμάτων αναζήτησης

A.4 Ρυθμίσεις Εφαρμογής

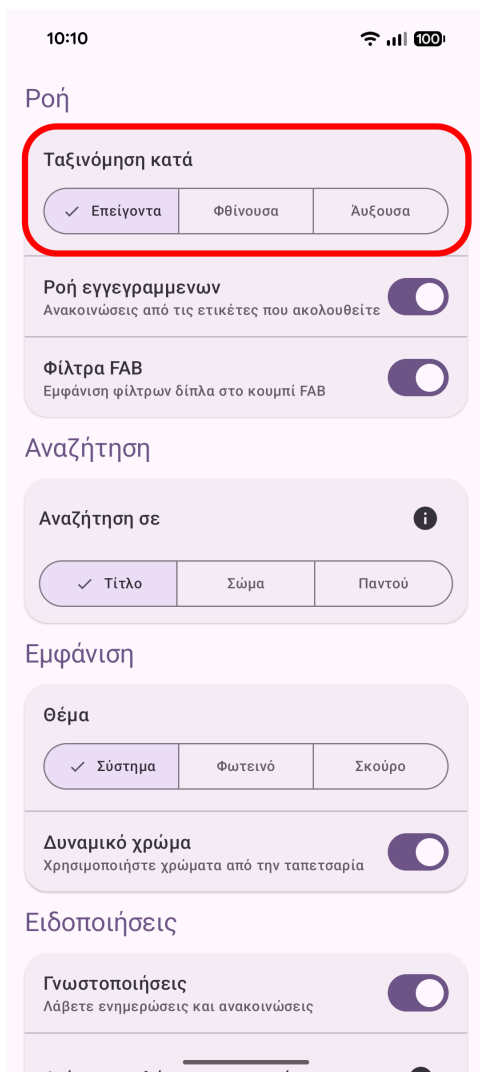
Η οθόνη ρυθμίσεων επιτρέπει την πλήρη παραμετροποίηση της εφαρμογής.



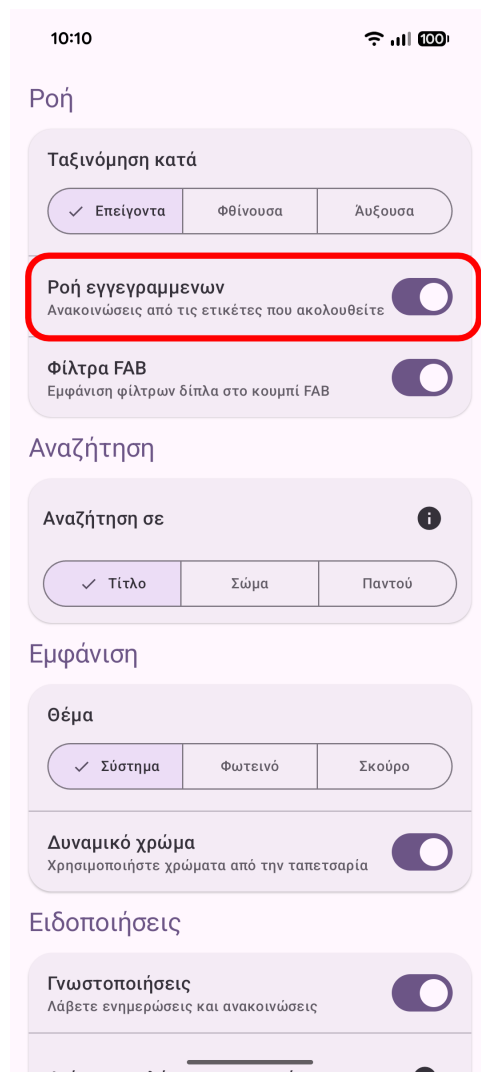
Σχήμα Α.26: Επισήμανση του εικονιδίου ρυθμίσεων στην αρχική οθόνη

A.4.1 Επιλογές Ροών

Ο χρήστης μπορεί να αλλάξει την ταξινόμηση των ροών και να απενεργοποιήσει τη ροή «Εγγεγραμμένα».

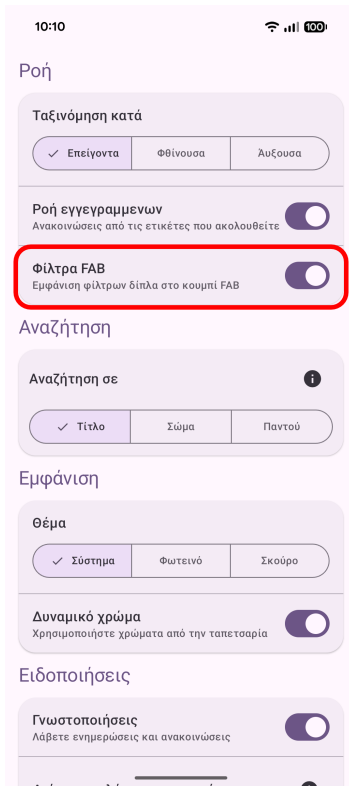


Σχήμα Α.27: Επιλογή ταξινόμησης των ροών



Σχήμα Α.28: Ενεργοποίηση της ροής «Εγγεγραμμένα»

Επιπλέον, παρέχεται η δυνατότητα απενεργοποίησης των φίλτρων που εμφανίζονται δίπλα στο αιωρούμενο κουμπί ενεργειών (FAB).



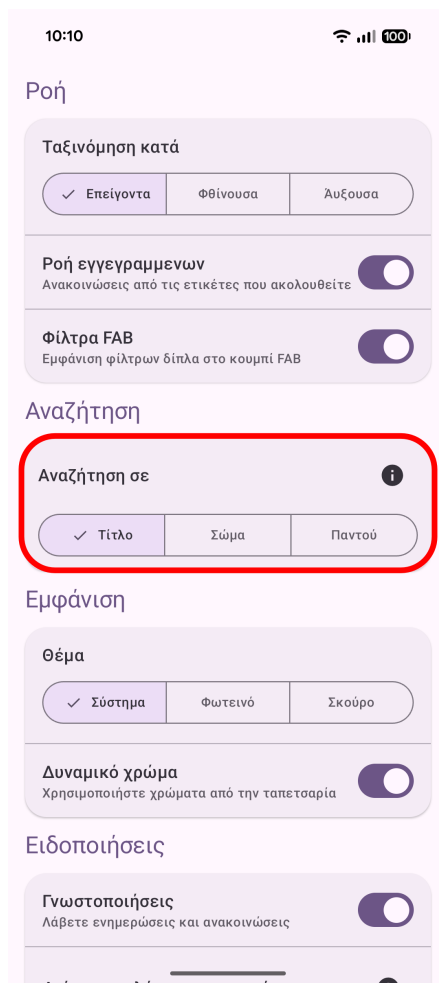
Σχήμα Α.29: Ρύθμιση εμφάνισης φίλτρων δίπλα στο FAB



Σχήμα Α.30: Αρχική οθόνη με απενεργοποιημένα τα φίλτρα του FAB

Α.4.2 Επιλογές Αναζήτησης

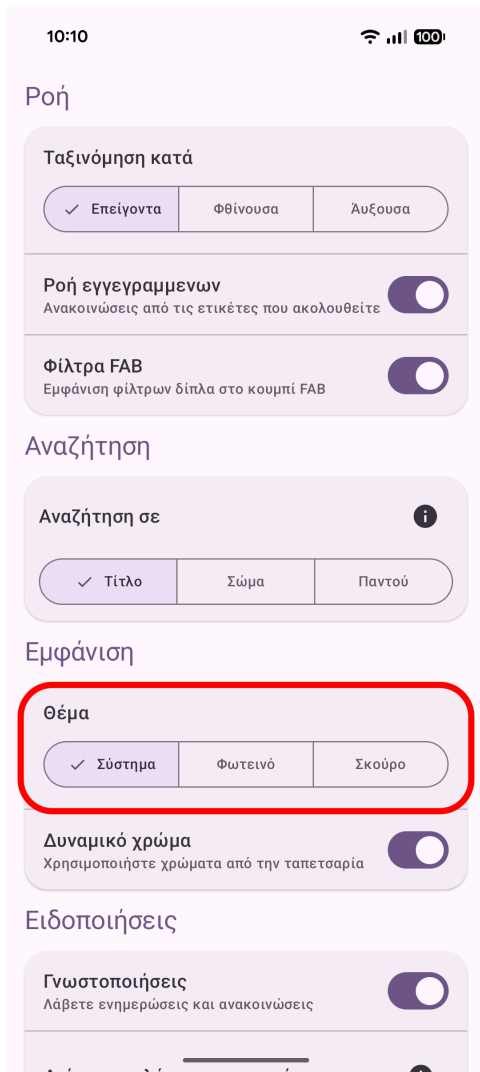
Από αυτή την ενότητα, ο χρήστης μπορεί να προσαρμόσει την εμφάνιση της αναζήτησης.



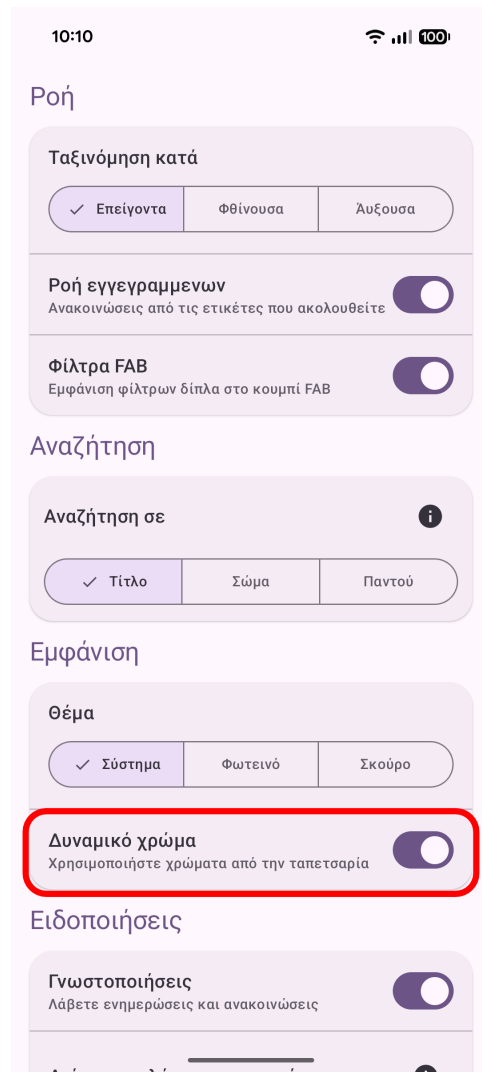
Σχήμα Α.31: Επιλογή εμβέλειας αναζήτησης

Α.4.3 Επιλογές Εμφάνισης

Ο χρήστης έχει τη δυνατότητα να αλλάξει το θέμα εμφάνισης της εφαρμογής, καθώς και να απενεργοποιήσει το Δυναμικό Χρώμα (Dynamic Color).



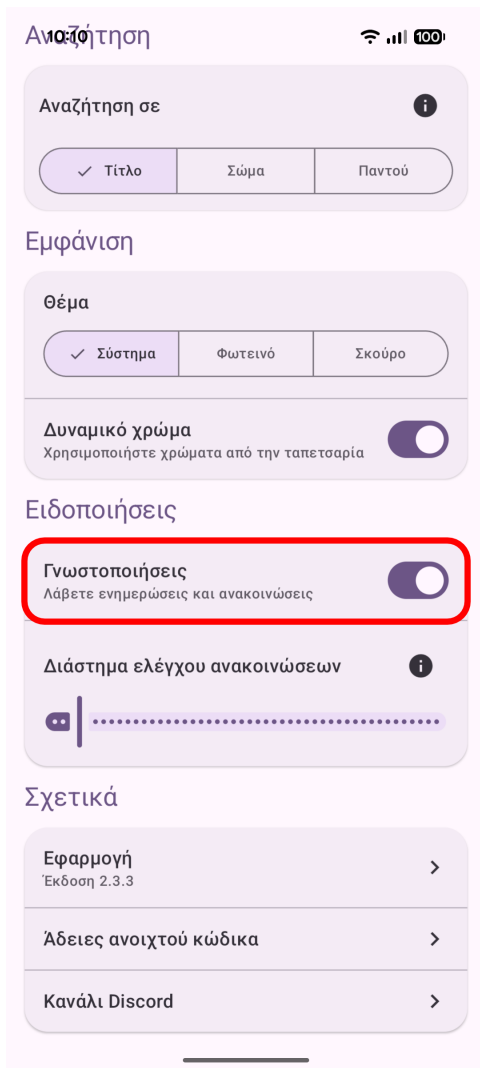
Σχήμα Α.32: Επιλογή θέματος εμφάνισης



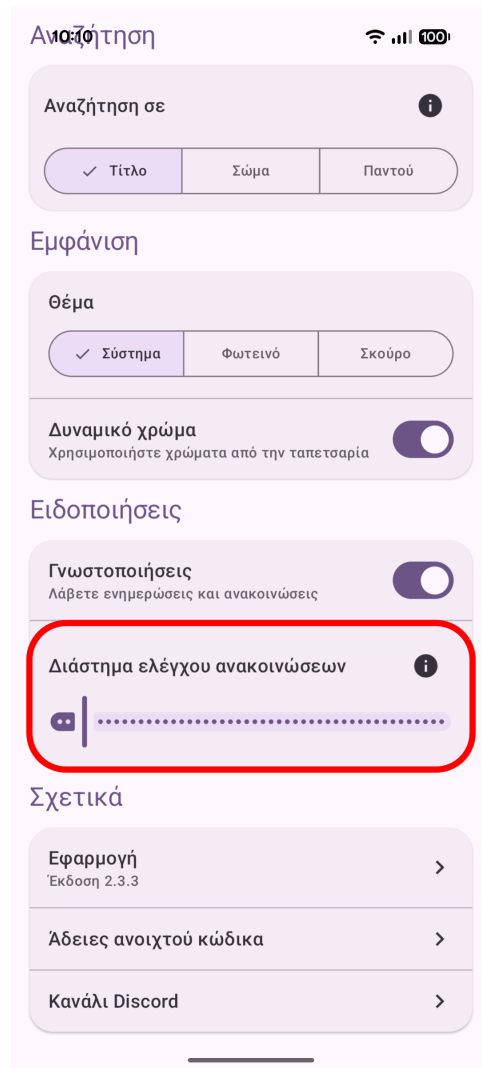
Σχήμα Α.33: Ρύθμιση δυναμικού χρώματος

Α.4.4 Επιλογές Ειδοποιήσεων

Ο χρήστης μπορεί να μεταβεί στις ρυθμίσεις ειδοποιήσεων της συσκευής του όπως επίσης να επιλέξει τη συχνότητα με την οποία η συσκευή θα ελέγχει στο παρασκήνιο για νέες ανακοινώσεις.



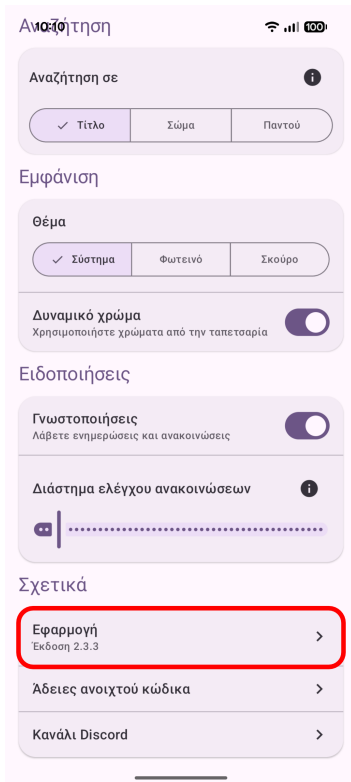
Σχήμα Α.34: Μετάβαση στις ρυθμίσεις ειδοποιήσεων



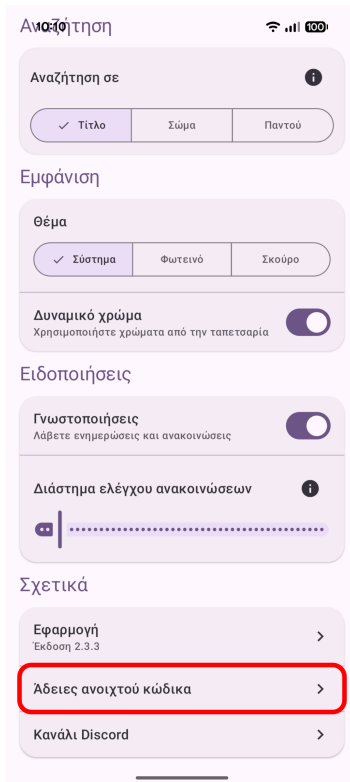
Σχήμα Α.35: Ρύθμιση συχνότητας ελέγχου για νέες ειδοποιήσεις

A.4.5 Σχετικά με την Εφαρμογή

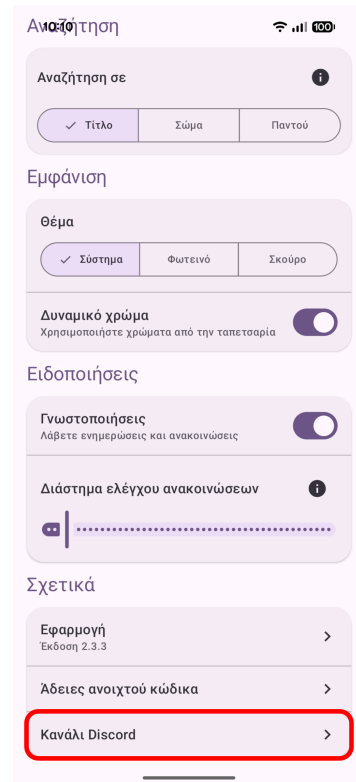
Στην ενότητα αυτή, ο χρήστης μπορεί να δει την τρέχουσα έκδοση της εφαρμογής. Πατώντας πάνω της, μεταφέρεται στο αποθετήριο του *project* στο GitHub. Επιπλέον, δίνεται η δυνατότητα στον χρήστη να προβάλει τις άδειες ανοιχτού κώδικα (*open-source licenses*) των βιβλιοθηκών που χρησιμοποιεί η εφαρμογή. Τέλος, παρέχεται σύνδεσμος για μετάβαση στο επίσημο κανάλι του Discord, με σκοπό την υποστήριξη των χρηστών.



Σχήμα Α.36: Προβολή έκδοσης και σύνδεσμος GitHub



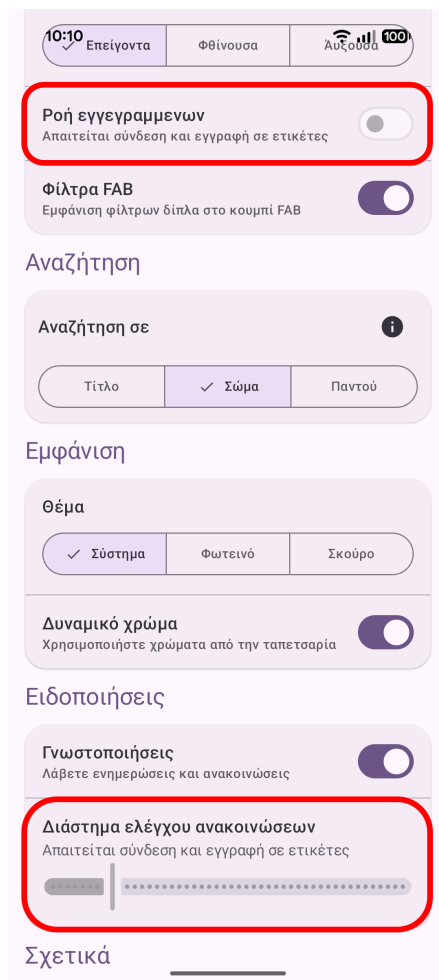
Σχήμα Α.37: Προβολή των αδειών ανοιχτού κώδικα



Σχήμα Α.38: Σύνδεσμος Discord

Α.4.6 Περιορισμοί Λειτουργίας Επισκέπτη

Αν ο χρήστης δεν έχει συνδεθεί, η δυνατότητα ενεργοποίησης της καρτέλας «Εγγεγραμμένα», καθώς και ο καθορισμός του χρονικού διαστήματος ελέγχου για νέες ειδοποιήσεις απενεργοποιούνται, εμφανίζοντας σχετική ενημέρωση στην οθόνη.



Σχήμα Α.39: Απενεργοποιημένη επιλογή «Εγγεγραμμένα» και ρύθμιση ειδοποιήσεων για μη συνδεδεμένους χρήστες