ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σύγχρονες τεχνικές automation testing: Ανάλυση και μελέτη περίπτωσης

Της φοιτήτριας
Μαργαρίτας Νεκταρίας Αλεξίου
Αρ. Μητρώου: 164852

Επιβλέπων Καθηγητής
Αντώνης Σιδηρόπουλος

Σεπτέμβριος 2025

Τίτλος Π.Ε.: Σύγχρονες τεχνικές automation testing: Ανάλυση και μελέτη περίπτωσης
Κωδικός Π.Ε.: 24142
Ονοματεπώνυμο φοιτήτριας: Αλεξίου Μαργαρίτα Νεκταρία
Ονοματεπώνυμο εισηγητή: Σιδηρόπουλος Αντώνης
Ημερομηνία ανάληψης Π.Ε: 08/03/2024
Ημερομηνία περάτωσης Π.Ε: 11/09/2025

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

Πρόλογος

Στην πρακτική μου εργασία, ξεκίνησα να ασχολούμαι με το Automation Testing. Εργάζομαι ως Test Automation Engineer τα τελευταία 3 χρόνια και έχω συμμετάσχει σε ποικίλα έργα, άλλα στημένα ήδη από συναδέλφους με legacy κώδικα, και άλλα πιο πρόσφατα αλλά με πρακτικές που συχνά επιδέχονταν βελτίωσης. Όταν κλήθηκα να αποφασίσω για το θέμα εργασίας, επέλεξα τη δημιουργία ενός νέου project αυτοματοποίησης κώδικα. Σκοπός μου ήταν, να έχω την ευκαιρία να εφαρμόσω από την αρχή και όσο καλύτερα μπορώ, τις πρακτικές που έχω μάθει. Η πτυχιακή αυτή, με βοήθησε να συνδυάσω την εμπειρία μου με την ακαδημαϊκή μελέτη, εμβαθύνοντας τόσο στη θεωρητική πλευρά του λογισμικού ελέγχου όσο και την πρακτική εφαρμογή σύγχρονων τεχνικών.

Περίληψη

Η παρούσα πτυχιακή εργασία εξετάζει σύγχρονες τεχνικές automation testing και τις εφαρμόζει σε μελέτη περίπτωσης πάνω στον πίνακα ανακοινώσεων της πλατφόρμας Aboard του Τμήματος Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων. Στο θεωρητικό μέρος συνοψίζονται τα θεμέλια ελέγχου και ποιότητας λογισμικού, η κατηγοριοποίηση δοκιμών, καθώς και μοντέλα/μεθοδολογίες ανάπτυξης (Agile, TDD, BDD, CI/CD), με έμφαση στο shift-left και στο continuous testing.

Στο πρακτικό μέρος υλοποιήθηκαν δύο συμπληρωματικές σουίτες: (α) API testing με BDD (Reqnroll/SpecFlow), NUnit, RestSharp και ανάλυση JSON, και (β) UI testing με Selenium WebDriver 4 ακολουθώντας το Page Object Model, explicit waits και παραμετροποίηση μέσω appsettings. Λήφθηκε υπόψη, η ιδιαιτερότητα του authentication (token διαθέσιμο μέσω browser session).

Τα αποτελέσματα δείχνουν ότι ο συνδυασμός API και UI tests προσφέρει σφαιρική εικόνα ποιότητας: τα API tests επιβεβαίωσαν τη συνέπεια και την ορθότητα των αποκρίσεων, ενώ τα UI tests επαλήθευσαν κρίσιμες ροές (login, φιλτράρισμα, ταξινόμηση, πλοήγηση). Αναδείχθηκαν πρακτικά ζητήματα, όπως αστοχίες στην ταξινόμηση ημερομηνιών και η χρήση μη βέλτιστων locators. Η αρχιτεκτονική (POM, με κοινούς helpers και καθαρή ρύθμιση) βελτίωσε συντηρησιμότητα, σταθερότητα και φορητότητα.

Τέλος, προτείνονται επεκτάσεις με role-based σενάρια (γραμματεία/καθηγητής) και κάλυψη μεταβαλλόμενων endpoints (POST/PUT/DELETE) σε ελεγχόμενο περιβάλλον, ένταξη σε CI/CD με headless εκτέλεση. Συνολικά, η εργασία αναδεικνύει τη σημαντικότητα του στοχευμένου συνδυασμού API και UI αυτοματοποιημένων ελέγχων, με σκοπό την ενίσχυση της ποιότητας της πλατφόρμας.

«Modern automation testing techniques: Analysis and case study»

«Margarita Nektaria Alexiou»

Abstract

This thesis examines modern automation testing techniques and applies them to a case study on the announcements board of the Aboard platform of the Department of Information and Electronic Engineering. The theoretical part summarizes the foundations of software quality and testing, the main test categories, and prevalent development models/methods (Agile, TDD, BDD, CI/CD), with emphasis on shift-left and continuous testing.

The practical part implements two complementary test suites: (a) API testing using BDD (Reqnroll/SpecFlow), NUnit, RestSharp, and JSON assertions; and (b) UI testing with Selenium WebDriver 4 following the Page Object Model, explicit waits, and configuration via appsettings. The authentication peculiarity—an access token available only through the browser session—was explicitly considered in the design.

Results show that combining API and UI tests provides a comprehensive view of quality: API tests verified response correctness and consistency, while UI tests validated critical flows (login, filtering, sorting, pagination). The work surfaced practical issues, such as inconsistencies in date sorting and sub-optimal locators. The chosen architecture (POM with shared helpers and clean configuration) improved maintainability, stability, and portability of the suite.

Finally, the thesis outlines extensions: role-based scenarios (e.g., secretariat/lecturer) and coverage of mutating endpoints (POST/PUT/DELETE) in a controlled environment, as well as CI/CD integration with headless execution. Overall, the study highlights the value of a targeted combination of API and UI automated tests to strengthen the quality of the platform.

Ευχαριστίες

Θα ήθελα να εκφράσω την ειλικρινή μου ευγνωμοσύνη στους γονείς μου, οι οποίοι με στήριξαν έμπρακτα στην απόφασή μου ξεκινήσω μία δεύτερη σχολή.

Στη συνέχεια, θα ευχαριστήσω όλους τους αγαπημένους μου φίλους, που είναι δίπλα μου αυτά τα χρόνια και με «πιέζουν» με πολύ γλυκό τρόπο, να τελειώσω τη σχολή. Ιδιαίτερα, θα ήθελα να αναφέρω την Ιωάννα και τη Φανή, των οποίων η συμβολή, η παρουσία και η εμψύχωση έπαιξαν ουσιαστικό ρόλο στην πορεία μου.

Τέλος, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Αντώνη Σιδηρόπουλο, για την καθοδήγηση και εμπιστοσύνη μου έδειξε κατά τη διάρκεια της εκπόνησης της παρούσας εργασίας.

Περιεχόμενα

Πρόλογος iv
Περίληψη v
Abstract vi
Ευχαριστίες vii
Περιεχόμενα viii
Κατάλογος Εικόνων x
Συντομογραφίες xi
Κεφάλαιο 1ο: Έλεγχος και Ποιότητα Λογισμικού 1
1.1 Εισαγωγή 1
1.2 Ιστορική Αναδρομή στον Έλεγχο και την Ποιότητα Λογισμικού 1
1.3 Η Σημασία της Διασφάλισης Ποιότητας και του Ελέγχου Λογισμικού 1
1.4 Ο Κύκλος Ζωής του Λογισμικού και ο Ρόλος του Testing 2
1.5 Συμπέρασμα 2
Κεφάλαιο 2ο: Σύγχρονες Πρακτικές Testing 3
2.1 Εισαγωγή 3
2.2 Κατηγορίες Ελέγχου Λογισμικού 3
2.2.1 Λειτουργικές vs. Μη Λειτουργικές Δοκιμές 3
2.2.2 Δοκιμές βάσει προσέγγισης 4
2.2.3 Δοκιμές βάσει στόχου (Goal-based Testing) 4
2.3 Χειροκίνητος vs. Αυτοματοποιημένος Έλεγχος 5
2.4 Οφέλη του Αυτοματοποιημένου Ελέγχου 5
2.5 Προκλήσεις και Περιορισμοί του Automation Testing 5
2.6 Σύγχρονα Εργαλεία για Automation Testing 5
2.7 Συμπέρασμα 6
Κεφάλαιο 3ο: Μοντέλα και Μεθοδολογίες Ανάπτυξης Λογισμικού 7
3.1 Εισαγωγή 7
3.2 Παραδοσιακά Μοντέλα Ανάπτυξης Λογισμικού 7
3.2.1 Waterfall Model 7
3.2.2 V-Model 7
3.2.3 Spiral Model 7
3.3 Agile Development και Testing 7
3.4 Test-Driven Development (TDD) 8

3.5 Behavior-Driven Development (BDD) 8
3.6 Continuous Integration & Continuous Deployment (CI/CD) 8
3.7 AI-Driven Testing (Δοκιμές με Τεχνητή Νοημοσύνη) 8
3.8 Συμπέρασμα 9
Κεφάλαιο 4ο: Μελέτη περίπτωσης: Πίνακας ανακοινώσεων Aboard 10
4.1 Πίνακας ανακοινώσεων Aboard 10
4.1.1 Εισαγωγή 10
4.1.2 Ρόλοι και Τύποι Χρηστών 10
4.1.3 Λειτουργία και Διασύνδεση API 10
4.2 API Testing 11
4.2.1 Εισαγωγή 11
4.2.2 Περιγραφή του Aboard API 11
4.2.3 Εργαλεία και Τεχνολογίες 13
4.2.4 Υλοποίηση 14
4.2.5 Αποτελέσματα και παρατηρήσεις 18
4.2.6 Συμπεράσματα 21
4.3 UI Testing - Selenium 22
4.3.1 Εισαγωγή 22
4.3.2 Σχεδιασμός και Τεχνολογίες 22
4.3.3 Δομή του Project 23
4.3.4 Σενάρια και Υλοποίηση 27
4.3.5 Αποτελέσματα και παρατηρήσεις 32
4.3.6 Συμπεράσματα 37
Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης 38
5.1 Εισαγωγή 38
5.2 Συμπεράσματα από τη μελέτη περίπτωσης (API και UI) 38
5.3 Προτάσεις βελτίωσης και μελλοντικής επέκτασης 39
5.4 Επίλογος 39
ΒΙΒΛΙΟΓΡΑΦΙΑ 40

Κατάλογος Εικόνων

Εικόνα 1: Απόκτηση Authentication Token από browser 11
Εικόνα 2: API Documentation 12
Εικόνα 3: Σενάριο γραμμένο με δομή Gherking 14
Εικόνα 4: Αρνητικά Σενάρια 14
Εικόνα 5: Σενάριο με διαφορετικές παραμέτρους 15
Εικόνα 6: Επαναχρησιμοποιήσιμο βήμα 15
Εικόνα 7: Τεστ με μεγάλο όγκο δεδομένων στην απάντηση 16
Εικόνα 8: Βήμα που ελέγχει την απάντηση από τοπικό αρχείο 16
Εικόνα 9: Τελική Δομή 18
Εικόνα 10: Αποτελέσματα API 19
Εικόνα 11: Τεστ που αποτυγχάνουν 20
Εικόνα 12: Μήνυμα αποτυχίας 1 20
Εικόνα 13: Μήνυμα αποτυχίας 2 21
Εικόνα 14: Δομή 23
Εικόνα 15: BaseTest Κλάση 24
Εικόνα 16: Αρχικοποίηση του WebDriverWait στη σελίδα AnnouncementsPage.cs 25
Εικόνα 17: Παραδείγμα χρήσης του WebDriverWait στη σελίδα AnnouncementsPageValidator.cs 25
Εικόνα 18: Χρήση locators με By στη σελίδα AnnouncementsPageElementMap.cs 25
Εικόνα 19: Appsettings.Template.json 26
Εικόνα 20: ConfigReader.cs 27
Εικόνα 21: Login Tests 28
Εικόνα 22: Κεντρική σελίδα Ανακοινώσεων 29
Εικόνα 23: Μενού Φίλτρων 30
Εικόνα 24: Σενάριο FilterByText 31
Εικόνα 25: Σφάλμα σε ταξινόμηση 1 33
Εικόνα 26: Σφάλμα σε ταξινόμηση 2 33
Εικόνα 27: Λεπτομέρειες αποτυχίας του σεναρίου 34
Εικόνα 28: Μέθοδος ελέγχου ταξινόμησης 34
Εικόνα 29: Στοιχεία με εύκολο εντοπισμό 35
Εικόνα 30: Στοιχεία με περίπλοκο εντοπισμό 35
Εικόνα 31: Διπλή χρήση id σε στοιχείο της σελίδας 36
Εικόνα 32: Έλεγχος κειμένου Τίτλων 37

Συντομογραφίες

Π.Ε.	Πτυχιακή Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
SDLC	Software Development Life Cycle
CMM	Capability Maturity Model
CI/CD	Continuous Integration and Continuous Delivery/Deployment
QA	Quality Assurance
UAT	User Acceptance Testing
TDD	Test-Driven Development
BDD	Behavior-Driven Development
AI	Artificial Intelligence

Κεφάλαιο 1ο: Έλεγχος και Ποιότητα Λογισμικού

1.1 Εισαγωγή

Στη σύγχρονη εποχή της ψηφιακής τεχνολογίας, το λογισμικό αποτελεί αναφαίρετο μέρος τόσο των επιχειρηματικών, όσο και των προσωπικών δραστηριοτήτων. Η ποιότητα του λογισμικού έχει καθοριστικό ρόλο στη λειτουργικότητα, στην απόδοση και στην ασφάλεια των συστημάτων. Η διασφάλιση ποιότητας και οι διαδικασίες ελέγχου λογισμικού αποτελούν κρίσιμα στάδια του κύκλου ζωής ανάπτυξης λογισμικού (Software Development Life Cycle - SDLC), καθώς συμβάλλουν στην ανίχνευση και διόρθωση σφαλμάτων πριν την παράδοση στον τελικό χρήστη [1].

Η αυτοματοποίηση του ελέγχου λογισμικού (automation testing) έχει εξελιχθεί σε βασική πρακτική στον χώρο της ανάπτυξης λογισμικού, εξασφαλίζοντας υψηλότερη αποτελεσματικότητα και αξιοπιστία. Το κεφάλαιο αυτό εξετάζει την ιστορική εξέλιξη του ελέγχου λογισμικού, τη σημασία της διασφάλισης ποιότητας και τις τεχνικές που χρησιμοποιούνται σήμερα για την επίτευξη ενός ασφαλούς και λειτουργικού προϊόντος.

1.2 Ιστορική Αναδρομή στον Έλεγχο και την Ποιότητα Λογισμικού

Η έννοια του ελέγχου λογισμικού χρονολογείται από τις πρώτες δεκαετίες της επιστήμης των υπολογιστών. Στη δεκαετία του 1950, όταν τα πρώτα προγράμματα γράφονταν κυρίως σε assembly γλώσσες, η διαδικασία δοκιμών ήταν εξαιρετικά περιορισμένη και βασιζόταν κυρίως στην εμπειρική παρατήρηση.

Τη δεκαετία του 1970, με την ανάπτυξη πιο προηγμένων γλωσσών προγραμματισμού και μεθοδολογιών ανάπτυξης λογισμικού, η ανάγκη για πιο δομημένες διαδικασίες ελέγχου έγινε εμφανής. Οι πρώτες θεωρητικές βάσεις για την αξιολόγηση της ποιότητας λογισμικού τέθηκαν από ερευνητές όπως ο Glenford J. Myers, ο οποίος στο βιβλίο του *The Art of Software Testing* [2] ανέδειξε τη σημασία του ελέγχου λογισμικού ως αναπόσπαστο μέρος της ανάπτυξης λογισμικού.

Αργότερα, στις δεκαετίες του 1980 και 1990, η αυξημένη χρήση των προσωπικών υπολογιστών και η επέκταση του διαδικτύου οδήγησαν σε μεγαλύτερη ζήτηση για αξιόπιστο λογισμικό. Οι έλεγχοι λογισμικού συστηματοποιήθηκαν μέσω των προτύπων όπως το ISO 9001 και το Capability Maturity Model (CMM), ενώ η έννοια του αυτοματοποιημένου ελέγχου άρχισε να κερδίζει έδαφος [3].

Σήμερα, η ενσωμάτωση πρακτικών όπως DevOps και Continuous Integration/Continuous Deployment (CI/CD) έχει καταστήσει τον αυτοματοποιημένο έλεγχο αναπόσπαστο μέρος της διαδικασίας ανάπτυξης λογισμικού [4].

1.3 Η Σημασία της Διασφάλισης Ποιότητας και του Ελέγχου Λογισμικού

Η διασφάλιση ποιότητας λογισμικού (Software Quality Assurance) αναφέρεται στη συστηματική προσέγγιση για την εξασφάλιση της αξιοπιστίας και της αποδοτικότητας ενός λογισμικού. Ο ποιοτικός έλεγχος λογισμικού στοχεύει σε:

- **Αποφυγή σφαλμάτων:** Η έγκαιρη ανίχνευση προβλημάτων εξοικονομεί κόστος και χρόνο κατά την ανάπτυξη.
- **Ασφάλεια δεδομένων:** Ειδικά σε κλάδους όπως η χρηματοοικονομική τεχνολογία και η υγειονομική περίθαλψη, η ασφάλεια του λογισμικού είναι ζωτικής σημασίας [5].

- **Απόδοση και αποδοτικότητα:** Τα δοκιμασμένα και βελτιστοποιημένα λογισμικά ανταποκρίνονται καλύτερα στις απαιτήσεις των χρηστών.
- **Συμμόρφωση με κανονισμούς και πρότυπα:** Ορισμένα είδη λογισμικού πρέπει να τηρούν συγκεκριμένα πρότυπα (π.χ. ISO 27001 για την ασφάλεια πληροφοριών).

Οι τεχνικές ελέγχου λογισμικού περιλαμβάνουν τον στατικό έλεγχο (static testing), τον δυναμικό έλεγχο (dynamic testing), τον έλεγχο μονάδων (unit testing), τις δοκιμές απόδοσης (performance testing) και τις δοκιμές ασφάλειας (security testing). Ο αυτοματοποιημένος έλεγχος (test automation) έχει αναδειχθεί ως μια από τις πιο κρίσιμες και καινοτόμες μεθόδους διασφάλισης ποιότητας, καθώς επιτρέπει τη γρήγορη και επαναλαμβανόμενη εκτέλεση δοκιμών, μειώνοντας έτσι τα περιθώρια ανθρώπινου λάθους.

1.4 Ο Κύκλος Ζωής του Λογισμικού και ο Ρόλος του Testing

Ο κύκλος ζωής ανάπτυξης λογισμικού (SDLC) αποτελεί το βασικό πλαίσιο που ακολουθούν οι εταιρείες για τη δημιουργία ποιοτικού λογισμικού. Περιλαμβάνει τα εξής στάδια:

1. **Ανάλυση απαιτήσεων:** Προσδιορισμός των λειτουργικών και μη λειτουργικών απαιτήσεων του συστήματος.
2. **Σχεδίαση:** Δημιουργία της αρχιτεκτονικής και της δομής του λογισμικού.
3. **Υλοποίηση:** Ανάπτυξη του λογισμικού με χρήση προγραμματιστικών γλωσσών και εργαλείων.
4. **Έλεγχος και διασφάλιση ποιότητας:** Εκτέλεση στατικών και δυναμικών δοκιμών για την επαλήθευση της λειτουργικότητας, συμπεριλαμβανομένων των Shift-left testing και Continuous Testing, που διασφαλίζουν ότι ο έλεγχος ξεκινά από τα πρώτα στάδια του κύκλου ζωής.
5. **Διάθεση και συντήρηση:** Ανάπτυξη του λογισμικού σε παραγωγικό περιβάλλον και διαχείριση αναβαθμίσεων.

Το Shift-left testing είναι μια στρατηγική που μετατοπίζει τις δοκιμές προς τα πρώτα στάδια του κύκλου ζωής ανάπτυξης λογισμικού, ενσωματώνοντας τον έλεγχο ήδη από τη φάση του σχεδιασμού και της αρχικής ανάπτυξης. Αυτή η προσέγγιση βοηθά στον έγκαιρο εντοπισμό σφαλμάτων, μειώνοντας το κόστος και τον χρόνο επιδιόρθωσης [6]. Σε συνδυασμό με το Continuous Testing, όπου οι δοκιμές εκτελούνται σε όλο τον κύκλο ζωής ανάπτυξης και όχι μόνο πριν την παραγωγή, οι στρατηγικές αυτές ενισχύουν τη σταθερότητα και την απόδοση του τελικού προϊόντος.

1.5 Συμπέρασμα

Η εξέλιξη του ελέγχου και της διασφάλισης ποιότητας λογισμικού ακολουθεί την πρόοδο της βιομηχανίας της πληροφορικής. Από τις πρώτες δεκαετίες της πληροφορικής έως τις σύγχρονες πρακτικές DevOps και CI/CD, η ανάγκη για αξιόπιστο λογισμικό έχει οδηγήσει στην ανάπτυξη τεχνικών ελέγχου και αυτοματοποίησης. Ο κύκλος ζωής ανάπτυξης λογισμικού αποτελεί τον βασικό μηχανισμό διαχείρισης της ανάπτυξης, με τον έλεγχο να παίζει καίριο ρόλο στη διασφάλιση της ποιότητας. Η ενσωμάτωση τεχνικών όπως το Shift-left testing και το Continuous Testing βελτιώνει την αποτελεσματικότητα του testing και μειώνει το κόστος διόρθωσης σφαλμάτων. Στα επόμενα κεφάλαια, θα αναλυθούν οι σύγχρονες τεχνικές automation testing, οι τεχνολογίες που τις υποστηρίζουν και μια μελέτη περίπτωσης που θα καταδείξει την αποτελεσματικότητά τους.

Κεφάλαιο 2ο: Σύγχρονες Πρακτικές Testing

2.1 Εισαγωγή

Η διασφάλιση ποιότητας λογισμικού αποτελεί κρίσιμο στάδιο στη διαδικασία ανάπτυξης, εξασφαλίζοντας την αξιοπιστία, τη λειτουργικότητα και την απόδοση των εφαρμογών. Με την εξέλιξη της τεχνολογίας και την αυξημένη πολυπλοκότητα των συστημάτων, οι μέθοδοι testing έχουν προσαρμοστεί στις σύγχρονες απαιτήσεις, ενσωματώνοντας νέες τεχνικές και εργαλεία. Σε αυτό το κεφάλαιο, θα εξετάσουμε τις διαφορετικές κατηγορίες testing, τη διάκριση μεταξύ χειροκίνητου και αυτοματοποιημένου ελέγχου, καθώς και τα πλεονεκτήματα και τις προκλήσεις της αυτοματοποίησης.

2.2 Κατηγορίες Ελέγχου Λογισμικού

Οι τεχνικές testing διακρίνονται σε διάφορες κατηγορίες, με βάση τη λειτουργικότητα, το επίπεδο ολοκλήρωσης και τη φύση των δοκιμών.

2.2.1 Λειτουργικές vs. Μη Λειτουργικές Δοκιμές

Οι λειτουργικές δοκιμές (Functional Testing) εστιάζουν στον έλεγχο της σωστής λειτουργίας του λογισμικού, επαληθεύοντας αν κάθε χαρακτηριστικό ανταποκρίνεται στις προδιαγραφές του συστήματος [2]. Περιλαμβάνουν:

- **Unit Testing** (Δοκιμές Μονάδας): Αφορούν τον έλεγχο μεμονωμένων τμημάτων ή μονάδων κώδικα, συνήθως από τους ίδιους τους προγραμματιστές, με στόχο την ανίχνευση σφαλμάτων στη βασική λογική του συστήματος.
- **Integration Testing** (Δοκιμές Ενσωμάτωσης): Σε αυτό το στάδιο ελέγχεται η ορθή επικοινωνία και αλληλεπίδραση μεταξύ διαφορετικών μονάδων ή συστατικών του συστήματος, διασφαλίζοντας τη σωστή λειτουργία τους ως ενιαίου συνόλου.
- **System Testing** (Δοκιμές Συστήματος): Πραγματοποιείται σε ολοκληρωμένο επίπεδο, όπου ελέγχεται το σύστημα στο σύνολό του για να επαληθευτεί ότι λειτουργεί σύμφωνα με τις προδιαγραφές και τις προσδοκίες.
- **User Acceptance Testing – UAT** (Δοκιμές Αποδοχής Χρήστη): Σε αυτό το στάδιο, οι τελικοί χρήστες εκτελούν δοκιμές για να επικυρώσουν ότι το λογισμικό ικανοποιεί τις πραγματικές τους ανάγκες και τις απαιτήσεις της επιχείρησης.
- **Regression Testing** (Επαναληπτικός έλεγχος): Αυτός ο τύπος δοκιμών διασφαλίζει ότι οι νέες αλλαγές ή προσθήκες στον κώδικα δεν επηρεάζουν αρνητικά την υπάρχουσα λειτουργικότητα, ώστε να διατηρηθεί η σταθερότητα του συστήματος.

Οι μη λειτουργικές δοκιμές (Non-Functional Testing) εστιάζουν σε στοιχεία όπως η απόδοση, η ασφάλεια και η εμπειρία του χρήστη, διασφαλίζοντας ότι το λογισμικό ικανοποιεί μη λειτουργικές απαιτήσεις [5]. Περιλαμβάνουν:

- **Performance Testing** (Δοκιμές Απόδοσης): Αφορούν τη μέτρηση της απόκρισης του συστήματος υπό διαφορετικά επίπεδα φόρτου εργασίας, συμπεριλαμβανομένων δοκιμών αντοχής (stress testing) και σταθερότητας (stability testing), με στόχο την αξιολόγηση της συμπεριφοράς του συστήματος σε πραγματικές συνθήκες λειτουργίας.

- **Security Testing** (Δοκιμές Ασφάλειας): Σε αυτές τις δοκιμές, σκοπός είναι να εντοπίζονται και αξιολογούνται τυχόν ευπάθειες του συστήματος που θα μπορούσαν να εκμεταλλευτούν κακόβουλοι χρήστες, διασφαλίζοντας την προστασία των δεδομένων και την ακεραιότητα του συστήματος.
- **Usability Testing** (Δοκιμές Χρηστικότητας): Εξετάζουν την ευκολία χρήσης και τη διαισθητικότητα της διεπαφής χρήστη, βεβαιώνοντας ότι η εμπειρία του τελικού χρήστη είναι ομαλή, αποτελεσματική και ικανοποιητική.
- **Compatibility Testing** (Δοκιμές Συμβατότητας): Εστιάζουν στον έλεγχο της σωστής λειτουργίας του λογισμικού σε διαφορετικά λειτουργικά συστήματα, συσκευές και περιβάλλοντα, καθώς και τη διαλειτουργικότητα με άλλα συστήματα και εφαρμογές.
- **Scalability Testing** (Δοκιμές κλιμάκωσης): Σε αυτού του τύπου δοκιμών, ελέγχεται η ικανότητα του συστήματος να διαχειρίζεται αυξανόμενα φορτία εργασίας και χρηστών, εξετάζοντας την απόδοσή του υπό συνθήκες αυξημένης ζήτησης.

2.2.2 Δοκιμές βάσει προσέγγισης

- **White-box Testing:** Ο έλεγχος πραγματοποιείται με γνώση του εσωτερικού κώδικα, επιτρέποντας στους testers να αναλύουν τις εσωτερικές δομές και ροές δεδομένων.
- **Black-box Testing:** Οι δοκιμές επικεντρώνονται στη λειτουργικότητα χωρίς γνώση του κώδικα, προσομοιώνοντας την εμπειρία του τελικού χρήστη. Χρησιμοποιούνται ευρέως σε System και UAT Testing.
- **Grey-box Testing:** Συνδυάζει στοιχεία από τις δύο προηγούμενες προσεγγίσεις, επιτρέποντας στους testers να εκμεταλλευτούν τόσο τη λειτουργική όσο και τη δομική ανάλυση του συστήματος [7]. Είναι ιδανικό για Penetration testing και API testing.

2.2.3 Δοκιμές βάσει στόχου (Goal-based Testing)

Οι δοκιμές βάσει στόχου αποτελούν μια ευέλικτη κατηγορία. Εστιάζουν σε συγκεκριμένους τεχνικούς ή επιχειρησιακούς στόχους. Αυτές οι εξειδικευμένες προσεγγίσεις προσφέρουν στους ομάδες QA (Quality Assurance) την ευελιξία να προσαρμόσουν τις στρατηγικές ανάλογα με τους συγκεκριμένους στόχους σε κάθε φάσης ανάπτυξης [8].

- **Smoke Testing** (Δοκιμές Καπνού): Πρόκειται για μια γρήγορη και επιφανειακή διαδικασία επικύρωσης που ελέγχει εάν οι κρίσιμες λειτουργίες του συστήματος λειτουργούν σωστά μετά μια νέα έκδοση. Αυτές οι δοκιμές χρησιμεύουν ως "πρώτη γραμμή άμυνας", επιτρέποντας στους δοκιμαστές να εντοπίσουν τυχόν σοβαρά ζητήματα πριν προχωρήσουν σε πιο λεπτομερείς ελέγχους.
- **Sanity Testing** (Δοκιμές Συνέπειας): Αφορούν στοχευμένους ελέγχους που πραγματοποιούνται μετά από συγκεκριμένες αλλαγές ή διορθώσεις στον κώδικα. Σε αντίθεση με τις πλήρεις δοκιμές επανέλεγχου, οι sanity suites επικεντρώνονται αποκλειστικά στις τροποποιημένες λειτουργίες, εξασφαλίζοντας ότι οι αλλαγές δεν έχουν εισάγει νέα προβλήματα [9].
- **Exploratory Testing** (Διερευνητικές Δοκιμές): Αυτή η προσέγγιση βασίζεται στην εμπειρία και τη διαίσθηση των δοκιμαστών, οι οποίοι εκτελούν δοκιμές χωρίς προκαθορισμένα σενάρια. Ιδιαίτερα χρήσιμη σε Agile περιβάλλοντα, το exploratory testing επιτρέπει την ανακάλυψη μη

προβλεπόμενων σφαλμάτων καθώς και την αξιολόγηση της συμπεριφοράς του συστήματος υπό πραγματικές συνθήκες χρήσης.

2.3 Χειροκίνητος vs. Αυτοματοποιημένος Έλεγχος

Ο έλεγχος λογισμικού μπορεί να πραγματοποιηθεί είτε χειροκίνητα είτε μέσω αυτοματοποιημένων εργαλείων. Ο **χειροκίνητος έλεγχος (Manual Testing)** βασίζεται σε ανθρώπινη παρέμβαση και είναι ιδιαίτερα χρήσιμος σε περιπτώσεις που απαιτούν διερευνητικές δοκιμές, έλεγχο χρηστικότητας και αξιολόγηση αισθητικής και αλληλεπίδρασης. Ωστόσο, είναι χρονοβόρος και επιρρεπής σε ανθρώπινα λάθη [3].

Ο **αυτοματοποιημένος έλεγχος (Automation Testing)** χρησιμοποιεί λογισμικό για την εκτέλεση δοκιμών χωρίς ανθρώπινη παρέμβαση. Αυτός ο τύπος testing είναι αποτελεσματικός για επαναλαμβανόμενες δοκιμές και ενσωματώνεται εύκολα σε περιβάλλοντα συνεχούς ανάπτυξης [4]. Τα κύρια χαρακτηριστικά του αυτοματοποιημένου ελέγχου περιλαμβάνουν ταχύτερη εκτέλεση και ακρίβεια, δυνατότητα δοκιμών σε πολλαπλές πλατφόρμες και επαναληψιμότητα των σεναρίων δοκιμών.

2.4 Οφέλη του Αυτοματοποιημένου Ελέγχου

Η αυτοματοποίηση δοκιμών έχει σημαντικά πλεονεκτήματα, ειδικά σε περιβάλλοντα Agile και DevOps. Ένα από τα βασικά οφέλη της είναι η μείωση του χρόνου εκτέλεσης δοκιμών, καθώς οι αυτοματοποιημένες δοκιμές εκτελούνται γρήγορα και αποδοτικά, επιτρέποντας ταχύτερη ανάπτυξη λογισμικού. Επιπλέον, προσφέρει βελτιωμένη κάλυψη δοκιμών, καθώς μπορούν να εκτελεστούν χιλιάδες δοκιμές ταυτόχρονα, εξασφαλίζοντας μεγαλύτερη ακρίβεια.

Παράλληλα, συμβάλλει στην ελαχιστοποίηση των ανθρώπινων λαθών, καθώς η χρήση λογισμικού μειώνει την πιθανότητα παράλειψης σφαλμάτων λόγω απροσεξίας. Τέλος, η αυτοματοποίηση είναι απαραίτητη για τη συνεχή ενσωμάτωση και ανάπτυξη (CI/CD), επιτρέποντας την ταχεία διάθεση νέων εκδόσεων λογισμικού [10].

2.5 Προκλήσεις και Περιορισμοί του Automation Testing

Παρά τα αναμφισβήτητα πλεονεκτήματα της αυτοματοποιημένης δοκιμαστικής διαδικασίας, η εφαρμογή της στην πράξη συναντά αρκετές δυσκολίες που απαιτούν προσεκτική αντιμετώπιση. Ένα από τα κύρια εμπόδια είναι το υψηλό αρχικό κόστος υλοποίησης, καθώς η δημιουργία αποτελεσματικών σεναρίων δοκιμών απαιτεί σημαντικό χρονικό διάστημα ανάπτυξης, εξειδικευμένο προσωπικό και επένδυση σε κατάλληλα εργαλεία [11].

Επιπλέον, η διαδικασία συντήρησης των αυτοματοποιημένων δοκιμών αποτελεί συνεχή πρόκληση. Καθώς το λογισμικό υφίσταται επαναλαμβανόμενες ενημερώσεις και βελτιώσεις, τα σενάρια δοκιμών χρήζουν συνεχούς αναθεώρησης για να παραμείνουν σχετικά και λειτουργικά.

Αξιοσημείωτο είναι ότι η αυτοματοποίηση εμφανίζει ιδιαίτερες δυσκολίες σε συγκεκριμένους τομείς, όπως οι δοκιμές χρηστικότητας (usability testing), όπου η ανθρώπινη κρίση και η ποιοτική αξιολόγηση της εμπειρίας χρήστη παραμένουν αναντικατάστατα στοιχεία.

2.6 Σύγχρονα Εργαλεία για Automation Testing

Η επιλογή κατάλληλων εργαλείων αυτοματοποιημένου ελέγχου αποτελεί καίριο παράγοντα για την επιτυχή εφαρμογή της μεθοδολογίας. Στον τομέα των δοκιμών web εφαρμογών, εργαλεία όπως το

Selenium, το Cypress και το Playwright έχουν κερδίσει ευρεία αποδοχή, λόγω της ευελιξίας και των προηγμένων δυνατοτήτων τους [12]. Για την εκτέλεση δοκιμών σε επίπεδο μονάδας (unit testing), οι προγραμματιστές μπορούν να αξιοποιήσουν ευρέως χρησιμοποιούμενες βιβλιοθήκες όπως η JUnit (για Java), η TestNG και η PyTest (για Python), οι οποίες προσφέρουν ισχυρά πλαίσια για τη δημιουργία και εκτέλεση δοκιμαστικών περιπτώσεων [13].

Στην περίπτωση των mobile εφαρμογών, το Appium έχει καθιερωθεί ως η κύρια λύση αυτοματοποίησης, υποστηρίζοντας και τις δύο κύριες πλατφόρμες (iOS και Android) [14]. Για τις δοκιμές απόδοσης και φόρτου (performance testing), εργαλεία όπως το JMeter και το LoadRunner παρέχουν ολοκληρωμένες λύσεις για την προσομοίωση υψηλής κίνησης και την ανίχνευση σημείων συμφόρησης [15]. Η στρατηγική επιλογή και συνδυασμός αυτών των εργαλείων μπορεί να οδηγήσει σε βελτιστοποιημένη διαδικασία ελέγχου, εξασφαλίζοντας υψηλά ποιοτικά αποτελέσματα.

2.7 Συμπέρασμα

Οι σύγχρονες πρακτικές testing έχουν εξελιχθεί σημαντικά για να ανταποκριθούν στις απαιτήσεις της ταχείας ανάπτυξης λογισμικού. Η μετάβαση από τον χειροκίνητο στον αυτοματοποιημένο έλεγχο, σε συνδυασμό με τη χρήση σύγχρονων εργαλείων και προσεγγίσεων, έχει οδηγήσει σε πιο αποτελεσματικές και αξιόπιστες διαδικασίες διασφάλισης ποιότητας. Στο επόμενο κεφάλαιο, θα εξετάσουμε τα μοντέλα ανάπτυξης λογισμικού και τη σύνδεσή τους με τις πρακτικές testing.

Κεφάλαιο 3ο: Μοντέλα και Μεθοδολογίες Ανάπτυξης Λογισμικού

3.1 Εισαγωγή

Η ανάπτυξη λογισμικού έχει εξελιχθεί σημαντικά τις τελευταίες δεκαετίες, με τις μεθοδολογίες και τα μοντέλα ανάπτυξης να προσαρμόζονται στις αυξανόμενες απαιτήσεις της αγοράς. Από τα παραδοσιακά γραμμικά μοντέλα μέχρι τις σύγχρονες Agile και DevOps προσεγγίσεις, κάθε μεθοδολογία έχει συγκεκριμένα πλεονεκτήματα και περιορισμούς. Παράλληλα, οι τεχνικές ελέγχου λογισμικού έχουν εξελιχθεί για να ταιριάζουν στις διαφορετικές ανάγκες κάθε μοντέλου. Σε αυτό το κεφάλαιο, θα αναλύσουμε τα βασικά μοντέλα ανάπτυξης λογισμικού, τη σχέση τους με το testing και τις σύγχρονες μεθοδολογίες όπως το Agile, το Test-Driven Development (TDD), το Behavior-Driven Development (BDD) και τις πρακτικές Continuous Integration/Continuous Deployment (CI/CD), με έμφαση στην ενσωμάτωση του DevSecOps και AI-driven testing.

3.2 Παραδοσιακά Μοντέλα Ανάπτυξης Λογισμικού

3.2.1 Waterfall Model

Το μοντέλο Waterfall είναι ένα από τα πρώτα επίσημα μοντέλα ανάπτυξης λογισμικού. Ακολουθεί μια αυστηρή, γραμμική προσέγγιση, όπου κάθε στάδιο (ανάλυση απαιτήσεων, σχεδίαση, υλοποίηση, έλεγχος, διάθεση και συντήρηση) ολοκληρώνεται πλήρως πριν προχωρήσουμε στο επόμενο. Το testing πραγματοποιείται στο τέλος, γεγονός που μπορεί να οδηγήσει σε υψηλό κόστος επιδιόρθωσης σφαλμάτων. Το μοντέλο αυτό, αν και προσφέρει σαφή δομή, αποδεικνύεται δυσπροσάρμοστο σε αλλαγές απαιτήσεων, με μελέτες να δείχνουν ότι το 65% των σφαλμάτων εντοπίζεται μόνο στη φάση δοκιμών. Ωστόσο, παραμένει εφαρμόσιμο σε έργα με σταθερές απαιτήσεις και αυστηρούς κανονιστικούς ελέγχους, όπως συστήματα ιατρικών συσκευών [16].

3.2.2 V-Model

Το V-Model (Verification and Validation) αποτελεί μια παραλλαγή του Waterfall, όπου το testing είναι ενσωματωμένο σε κάθε στάδιο της ανάπτυξης. Για κάθε φάση ανάπτυξης, υπάρχει μια αντίστοιχη δραστηριότητα δοκιμών, βελτιώνοντας την ποιότητα του λογισμικού ήδη από τα πρώτα στάδια. Παρά την προηγμένη ενσωμάτωση δοκιμών, το V-Model απαιτεί εκτενή τεκμηρίωση, γεγονός που περιορίζει την εφαρμογή του σε έργα με δυναμικές απαιτήσεις [18].

3.2.3 Spiral Model

Το Spiral Model εισάγει την έννοια της διαχείρισης κινδύνου, με την ανάπτυξη να πραγματοποιείται σε επαναλαμβανόμενες φάσεις (iterations). Το testing γίνεται σε κάθε κύκλο, μειώνοντας τον κίνδυνο εμφάνισης σοβαρών σφαλμάτων αργότερα στη διαδικασία. Σε έργα υψηλού κινδύνου όπως τα αμυντικά συστήματα, το Spiral Model έχει αποδειχθεί ιδιαίτερα αποτελεσματικό λόγω της επαναληπτικής αξιολόγησης κινδύνου [18].

3.3 Agile Development και Testing

Το Agile Development αποτελεί τη σύγχρονη προσέγγιση στην ανάπτυξη λογισμικού, δίνοντας έμφαση στην προσαρμοστικότητα, τη συνεχή επικοινωνία και τις γρήγορες παραδόσεις λειτουργικού λογισμικού [19]. Σύμφωνα με έρευνα του VersionOne [20] το 72% των ομάδων που υιοθετούν Agile, αναφέρουν βελτίωση στην ποιότητα του λογισμικού μέσω της συνεχούς διαδικασίας δοκιμών.

Στο πλαίσιο του Agile, οι μέθοδοι Scrum και Kanban καθορίζουν τη δομή των εργασιών. Ενώ το Scrum οργανώνει την ανάπτυξη σε αυστηρά χρονικά πλαίσια (sprints), το Kanban επιτρέπει μια πιο ρευστή προσέγγιση, με την κοινή τους παράμετρο να είναι η ενσωμάτωση δοκιμών σε κάθε στάδιο της ανάπτυξης. Το Agile Testing χαρακτηρίζεται από συνεχείς ελέγχους και στενή συνεργασία μεταξύ των testers και των developers, βελτιώνοντας την ποιότητα του τελικού προϊόντος.

3.4 Test-Driven Development (TDD)

Το TDD είναι μια μεθοδολογία όπου το testing προηγείται της ανάπτυξης του κώδικα [21]. Ο κύκλος ανάπτυξης ακολουθεί τη λογική Red-Green-Refactor:

- Red: Γράφεται μια αποτυχημένη δοκιμή.
- Green: Γράφεται ο ελάχιστος κώδικας για να περάσει η δοκιμή.
- Refactor: Ο κώδικας βελτιστοποιείται χωρίς να αλλάξει η λειτουργικότητα.

Το TDD εξασφαλίζει υψηλή ποιότητα κώδικα, αποδοτικότερο debugging και καλύτερη κάλυψη των απαιτήσεων. Μελέτες δείχνουν ότι ο κώδικας που αναπτύσσεται με TDD έχει κατά 40-50% λιγότερα σφάλματα σε σύγκριση με παραδοσιακές μεθόδους ωστόσο απαιτεί κατά 15-30% περισσότερο χρόνο ανάπτυξης [22].

3.5 Behavior-Driven Development (BDD)

Το BDD αποτελεί επέκταση του TDD, με έμφαση στην αναγνωσιμότητα και τη συνεργασία μεταξύ τεχνικών και μη τεχνικών μελών μιας ομάδας. Χρησιμοποιεί φυσική γλώσσα μέσω εργαλείων όπως Cucumber και SpecFlow, κάνοντας τις δοκιμές πιο κατανοητές και προσιτές σε όλα τα μέλη της ομάδας. Εργαλεία όπως το Cucumber μεταφράζουν τις απαιτήσεις σε εκτελέσιμες δοκιμές με Gherkin syntax, δημιουργώντας μια γέφυρα ανάμεσα σε επιχειρησιακούς και τεχνικούς φορείς.

3.6 Continuous Integration & Continuous Deployment (CI/CD)

Η ενσωμάτωση του testing στη διαδικασία CI/CD διασφαλίζει ότι οι αλλαγές στον κώδικα ελέγχονται και αναπτύσσονται συνεχώς, βελτιώνοντας τη σταθερότητα του λογισμικού [4].

- **Continuous Integration (CI)**: Ο κώδικας ενσωματώνεται συνεχώς στο κύριο αποθετήριο, με αυτοματοποιημένες δοκιμές να εκτελούνται αυτόματα.
- **Continuous Deployment (CD)**: Κάθε επιτυχημένη αλλαγή στον κώδικα προωθείται αυτόματα στο production περιβάλλον.

Παρά το κυρίαρχο ρόλο του Jenkins, νεότερες πλατφόρμες όπως GitHub Actions και GitLab CI/CD κερδίζουν έδαφος λόγω της εγγενούς ενσωμάτωσης με git repositories.

3.7 AI-Driven Testing (Δοκιμές με Τεχνητή Νοημοσύνη)

Η τεχνητή νοημοσύνη (AI) αρχίζει να έχει καθοριστικό ρόλο στο testing, επιτρέποντας πιο αποδοτικές και ευφυείς δοκιμές. Οι τεχνικές AI-driven testing χρησιμοποιούν αλγόριθμους μηχανικής μάθησης για τον εντοπισμό προτύπων σφαλμάτων, τη δημιουργία σεναρίων δοκιμών και την ανάλυση αποτελεσμάτων. Με τη χρήση AI, μπορούν να εντοπιστούν προβλήματα ταχύτερα και με μεγαλύτερη ακρίβεια, μειώνοντας τον ανθρώπινο παράγοντα και ενισχύοντας τη σταθερότητα του λογισμικού, ιδιαίτερα σε περιβάλλοντα συνεχούς ενσωμάτωσης. Αυτές οι πρακτικές επιταχύνουν την αναγνώριση

σφαλμάτων και βελτιώνουν τη συντηρησιμότητα του κώδικα, ειδικά σε περιβάλλοντα CI/CD, όπου η ταχύτητα και η ακρίβεια είναι κρίσιμες [23].

3.8 Συμπέρασμα

Η εξέλιξη των μοντέλων ανάπτυξης λογισμικού αντανακλά τη δυναμική φύση της τεχνολογικής καινοτομίας. Από τα αυστηρά γραμμικά μοντέλα της δεκαετίας του 1970 έως τις ευέλικτες σύγχρονες πρακτικές, η βιομηχανία έχει αναγνωρίσει ότι η ποιότητα του λογισμικού δεν είναι απλώς ένα τελικό στάδιο ελέγχου, αλλά μια συνεχής διαδικασία που διαπερνά όλα τα στάδια του κύκλου ζωής.
Οι σύγχρονες μεθοδολογίες, όπως το Agile, το TDD και οι πρακτικές CI/CD, δεν έχουν απλώς βελτιώσει την αποδοτικότητα, αλλά έχουν μεταμορφώσει τον τρόπο σκέψης των ομάδων ανάπτυξης. Σύμφωνα με μελέτες, η υιοθέτηση DevOps και συνεχών δοκιμών μπορεί να προσφέρει σημαντικά οφέλη, όπως:

- **30% μείωση του κόστους ανάπτυξης**, χάρη στη βελτιστοποίηση διαδικασιών και την αποτελεσματικότερη χρήση πόρων [24].
- **50% μείωση του κόστους συντήρησης**, μέσω της έγκαιρης ανίχνευσης και επίλυσης προβλημάτων [24].
- **58% των οργανισμών αναφέρουν βελτίωση στην ποιότητα του λογισμικού**, ως αποτέλεσμα της υιοθέτησης DevOps [25].
- **55% μείωση των ελαττωμάτων** χάρη στην εφαρμογή continuous integration [25].

Ωστόσο, η επιλογή της κατάλληλης μεθοδολογίας δεν είναι μια διαδικασία που ακολουθεί μια ενιαία προσέγγιση, αλλά εξαρτάται από διάφορους παράγοντες. Η πολυπλοκότητα του έργου διαδραματίζει καθοριστικό ρόλο, καθώς τα κρίσιμα συστήματα συχνά απαιτούν υβριδικές προσεγγίσεις που συνδυάζουν τη δομή του Waterfall με την ευελιξία του Agile. Παράλληλα, εταιρική κουλτούρα μπορεί να επηρεάσει την επιτυχία της μετάβασης σε Agile μεθοδολογίες, καθώς τέτοιες αλλαγές απαιτούν προσαρμογή όχι μόνο στις διαδικασίες αλλά και στη νοοτροπία των ομάδων. Επιπλέον, η διαθεσιμότητα πόρων αποτελεί πρακτικό εμπόδιο, καθώς οι σύγχρονες πρακτικές όπως το CI/CD και ο αυτοματοποιημένος έλεγχος απαιτούν επενδύσεις τόσο σε εργαλεία όσο και σε εξειδικευμένο προσωπικό, γεγονός που μπορεί να αποτελεί πρόκληση για μικρότερες ή πιο παραδοσιακές επιχειρήσεις [26].
Οι τρέχουσες τάσεις υποδεικνύουν ότι η ανάπτυξη λογισμικού κινείται προς:

- **Αυξημένη αυτοματοποίηση** μέσω τεχνικών όπως το AI-driven testing και αυτόματη παραγωγή κώδικα.
- **Πιο ενσωματωμένες πρακτικές ασφάλειας**, με το DevSecOps να διασφαλίζει ότι η ασφάλεια είναι μέρος του κύκλου ανάπτυξης και όχι ένα ξεχωριστό βήμα.
- **Δεδομενοκεντρικές προσεγγίσεις**, με αναλύσεις και μετρικές από CI/CD pipelines να διαμορφώνουν στρατηγικές ανάπτυξης.

Όπως επισημαίνει ο Ambler [26], η επιτυχία δεν προκύπτει από την αυστηρή εφαρμογή μιας ενιαίας μεθοδολογίας, αλλά από την ευελιξία στην προσαρμογή των αρχών στις ανάγκες και το πλαίσιο κάθε έργου.

Κεφάλαιο 4ο: Μελέτη περίπτωσης: Πίνακας ανακοινώσεων Aboard

4.1 Πίνακας ανακοινώσεων Aboard

4.1.1 Εισαγωγή

Η πλατφόρμα Aboard αποτελεί το βασικό ψηφιακό μέσο επικοινωνίας μεταξύ φοιτητών, διδακτικού προσωπικού και διοικητικών υπηρεσιών του Διεθνούς Πανεπιστημίου της Ελλάδος. Μέσα από αυτήν, οι χρήστες έχουν πρόσβαση σε ανακοινώσεις που αφορούν ακαδημαϊκά θέματα, διοικητικές διαδικασίες, εκδηλώσεις, επαγγελματικά ζητήματα, καθώς και άλλες δραστηριότητες του πανεπιστημίου.

Η συγκεκριμένη εργασία εστιάζει στη σελίδα ανακοινώσεων, η οποία λειτουργεί ως κεντρικός πίνακας ενημέρωσης. Η σελίδα εμφανίζει τις πιο πρόσφατες ανακοινώσεις, οργανωμένες σε θεματικές κατηγορίες (π.χ. Γραμματεία, Νέα τμήματος, Πτυχιακή/Διπλωματική Εργασία), παρέχει δυνατότητες φιλτραρίσματος και πλοήγησης σε παλαιότερες σελίδες. Επιπλέον, διαθέτει δυνατότητα επιλογής της προβολής των ενημερώσεων σε πλέγμα είτε σε λίστα.

4.1.2 Ρόλοι και Τύποι Χρηστών

Στην πλατφόρμα διακρίνονται δύο βασικοί τύποι χρηστών:

- **Δημόσιοι επισκέπτες**: Μπορούν να δουν επιλεγμένες ανακοινώσεις που είναι διαθέσιμες χωρίς να κάνουν σύνδεση.
- **Εγγεγραμμένοι χρήστες** (φοιτητές, καθηγητές, γραμματεία, προσωπικό): Με τη χρήση έγκυρων διαπιστευτηρίων, αποκτούν πρόσβαση σε ανακοινώσεις και κατηγορίες περιεχομένου που δεν είναι ορατές στο δημόσιο κοινό. Η αυθεντικοποίηση των χρηστών γίνεται με χρήση ενός κεντρικού SSO.

4.1.3 Λειτουργία και Διασύνδεση API

Η σελίδα των ανακοινώσεων λειτουργεί ως UI εφαρμογή που αλληλοεπιδρά με το backend μέσω REST API. Κάθε ενέργεια του χρήστη στο UI (π.χ. φόρτωση ανακοινώσεων, εφαρμογή φίλτρων, μετάβαση σε άλλη σελίδα) αντιστοιχεί σε κλήσεις API προς συγκεκριμένα endpoints. Έτσι, το περιεχόμενο που εμφανίζεται στη σελίδα είναι το αποτέλεσμα δυναμικής φόρτωσης δεδομένων, καθώς η παρουσία ενός διαθέσιμου API επιτρέπει τον έλεγχο και την αξιολόγηση των λειτουργιών του συστήματος χωρίς την ανάγκη εμπλοκής του UI.

Αυτός ο συνδυασμός UI και API πρόσβασης καθιστά τη σελίδα ανακοινώσεων ιδανικό παράδειγμα για τη μελέτη διαφορετικών προσεγγίσεων testing. Στα επόμενα κεφάλαια, η ίδια πλατφόρμα θα εξεταστεί:

- Μέσα από το πρίσμα του API Testing, εστιάζοντας στον έλεγχο της λειτουργικότητας των endpoints.
- Μέσα από το πρίσμα του UI Testing με Selenium, εστιάζοντας στην εμπειρία και αλληλεπίδραση του τελικού χρήστη.

4.2 API Testing

4.2.1 Εισαγωγή

Στο προηγούμενο κεφάλαιο παρουσιάστηκε η μελέτη περίπτωσης της διαδικτυακής εφαρμογής Aboard. Η επιλογή αυτή, έγινε επειδή αποτελεί ένα πραγματικό και ενεργό σύστημα, με ποικιλία API endpoints που καλύπτουν τόσο τα δημόσια όσο και τα προστατευμένα δεδομένα. Διαθέτει ποικιλίας που επιτρέπει την υλοποίηση πολλών σεναρίων ελέγχου, συμπεριλαμβανομένων ελέγχων λειτουργικότητας, ασφάλειας και διαχείρισης σφαλμάτων.

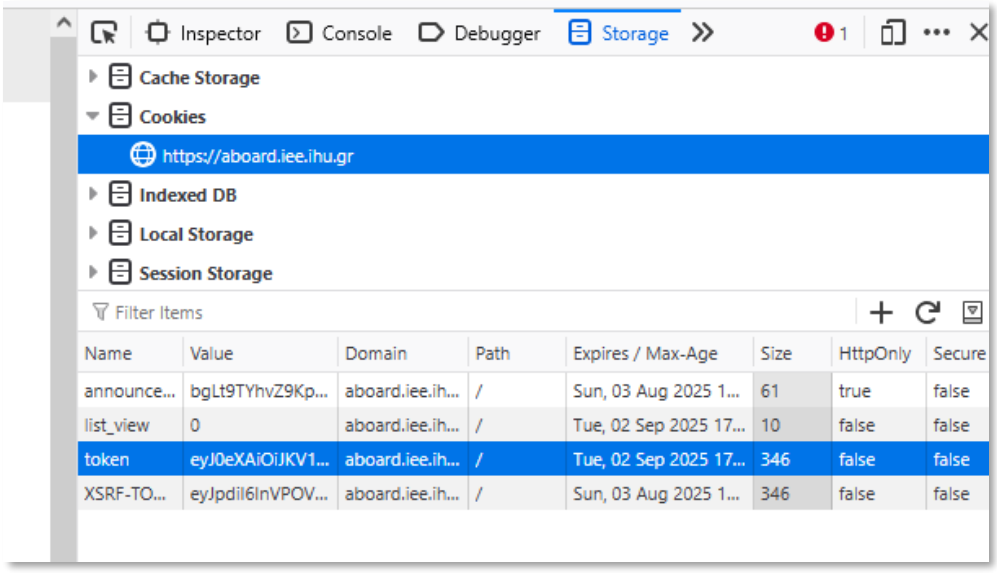
Η ύπαρξη του API καθιστά δυνατή την αξιολόγηση της λειτουργικότητας του συστήματος σε επίπεδο backend, προσφέροντας ταχύτητα στην εκτέλεση ελέγχων καθώς και εστίαση στη συμπεριφορά του συστήματος χωρίς την εμπλοκή του χρήστη [27]. Σε πολλές περιπτώσεις, ο εντοπισμός προβλημάτων μέσω API testing μπορεί να λειτουργήσει προληπτικά, καθώς σφάλματα που εντοπίζονται στο backend είναι συχνά ευκολότερο και λιγότερο δαπανηρό να διορθωθούν πριν εμφανιστούν στο UI.

Η επιλογή του API testing στην παρούσα εργασία, λειτουργεί συμπληρωματικά του UI testing, καθώς συνδέεται άμεσα με τα πλεονεκτήματα που προσφέρει έναντι του δεύτερου. Ενώ το UI testing επαληθεύει τη συμπεριφορά και τη χρηστικότητα της διεπαφής, το API testing επικεντρώνεται στην ορθότητα των αποκρίσεων, στην αξιοπιστία και στην τήρηση των συμφωνημένων προδιαγραφών, μειώνοντας παράλληλα τον χρόνο και την πολυπλοκότητα εκτέλεσης των δοκιμών.

4.2.2 Περιγραφή του Aboard API

Το Aboard API αποτελεί το backend σύστημα της πλατφόρμας και παρέχει endpoints για την ανάκτηση και διαχείριση ανακοινώσεων, καθώς και για την πρόσβαση σε πρόσθετες πληροφορίες όπως ετικέτες (tags) ή κατηγορίες.

Η αυθεντικοποίηση στο API είναι εξ ολοκλήρου frontend-based και απαιτεί ο χρήστης να εισέλθει στο UI, από όπου λαμβάνει το authentication token μέσω του sessionStorage ή localStorage του browser.



Εικόνα 1: Απόκτηση Authentication Token από browser

Το token αυτό χρησιμοποιείται στη συνέχεια για την πρόσβαση σε προστατευμένα endpoints, διαχωρίζοντας τη διαθεσιμότητα των δεδομένων:

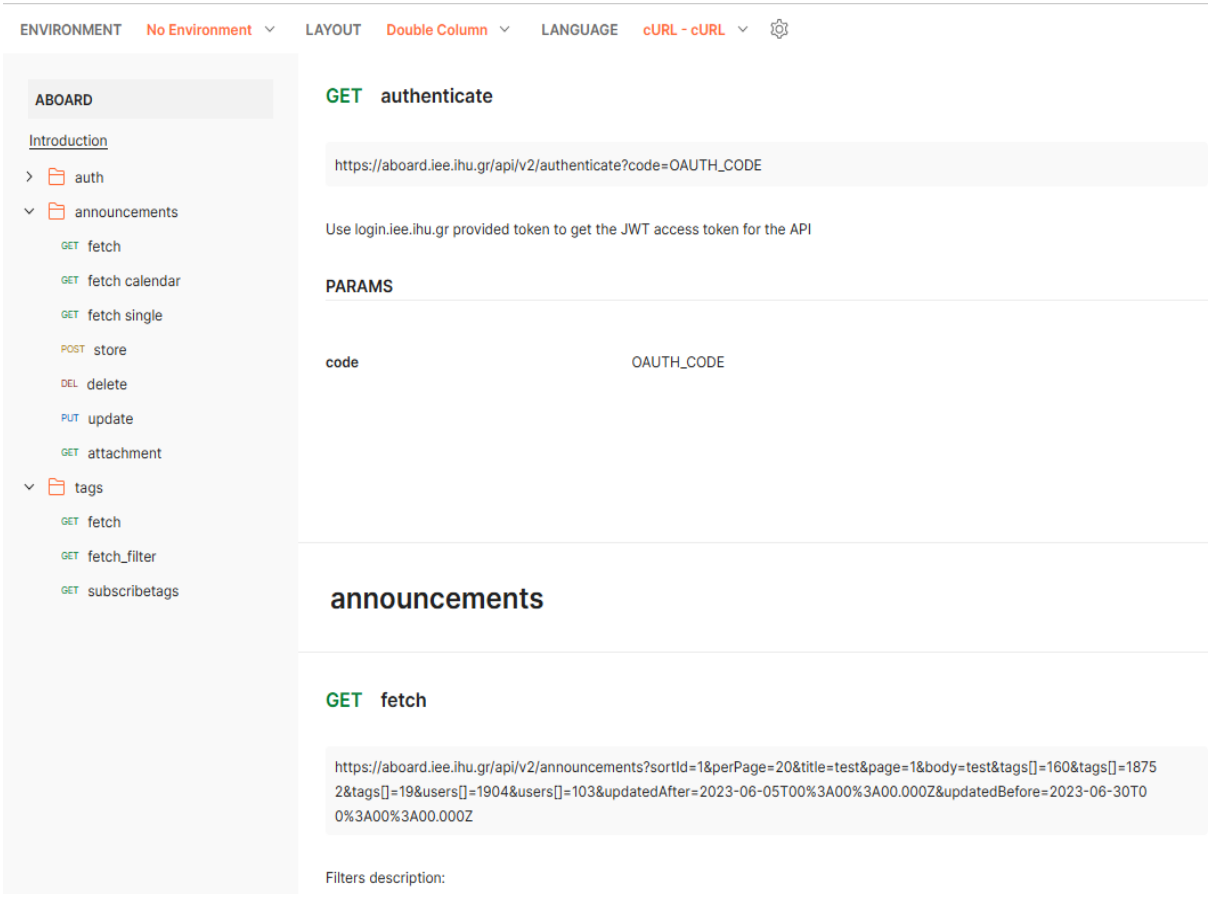
- Δημόσια endpoints, τα οποία είναι προσβάσιμα χωρίς έλεγχο ταυτότητας.
- Προστατευμένα endpoints, τα οποία απαιτούν έγκυρο token και επιστρέφουν διαφορετικό ή πλήρες σύνολο δεδομένων.

Αυτός ο μηχανισμός αυθεντικοποίησης δημιούργησε έναν περιορισμό στις πλήρως αυτοματοποιημένες δοκιμές, πιο συγκεκριμένα η λήψη του token δε μπορεί να γίνει αποκλειστικά με API κλήσεις μέσω εργαλείων όπως το Postman, καθώς απαιτεί είτε χειροκίνητη διαδικασία, είτε ειδικό script για αυτοματοποίηση.

Με βάση το διαθέσιμο documentation στο:

https://documenter.getpostman.com/view/10464469/2s93z6ejAD
Το API παρέχει endpoints όπως:

- /api/announcements για τη λήψη λίστας ανακοινώσεων.
- /api/announcements/{id} για τη λήψη λεπτομερειών συγκεκριμένης ανακοίνωσης.
- /api/tags για την ανάκτηση διαθέσιμων tags.



Εικόνα 2: API Documentation

4.2.3 Εργαλεία και Τεχνολογίες

Για την υλοποίηση του API testing χρησιμοποιήθηκε το Reqnroll (πρώην SpecFlow), το οποίο ακολουθεί τη μεθοδολογία BDD (Behavior-Driven Development) και επιτρέπει την περιγραφή των σεναρίων με τη χρήση της γλώσσας Gherkin. Η επιλογή του Reqnroll έγινε λόγω της ενσωμάτωσης του με το .NET, της υποστήριξης εργαλείων debugging και της ευκολίας δημιουργίας εύκολα αναγνώσιμων σεναρίων. Η χρήση BDD διευκολύνει την κατανόηση των σεναρίων τόσο από τεχνικούς όσο και από μη τεχνικούς συνεργάτες, διότι η περιγραφή γίνεται με φυσική γλώσσα [28].

Ως test framework επιλέχθηκε το NUnit, το οποίο υποστηρίζει δυνατότητες για assertions, tagging, παραμετροποίηση και ομαδοποίηση των tests σε διαφορετικές κατηγορίες [29]. Σε σύγκριση με εναλλακτικές όπως το MSTest ή το xUnit, το NUnit, προσφέρει μεγαλύτερη ευελιξία στις κατηγοριοποιήσεις και καλύτερη υποστήριξη από την κοινότητα για BDD σενάρια.

Για την εκτέλεση HTTP αιτημάτων και τη λήψη των αποκρίσεων χρησιμοποιήθηκε η βιβλιοθήκη RestSharp, η οποία παρέχει ευέλικτες μεθόδους για την αποστολή GET, POST και λοιπών τύπων αιτημάτων, όπως επίσης και για τον χειρισμό headers, authentication tokens και παραμέτρων.

Η βιβλιοθήκη Newtonsoft.Json χρησιμοποιήθηκε για την ανάλυση (parsing) των δεδομένων JSON και την εύκολη αναζήτηση ή τον έλεγχο συγκεκριμένων πεδίων στις αποκρίσεις [30].

Τέλος, το πακέτο Microsoft.Extensions.Configuration.Json επέτρεψε την αποθήκευση παραμέτρων όπως το BaseUrl και το authentication token σε αρχείο appsettings.json. Με αυτόν τον τρόπο αποφεύγεται το hardcoding, διευκολύνεται η αλλαγή των ρυθμίσεων χωρίς επέμβαση στον κώδικα και υποστηρίζεται η εκτέλεση των tests σε διαφορετικά περιβάλλοντα (π.χ. development, staging, production) με απλή αλλαγή στο αρχείο ρυθμίσεων.

Appsettings.json

```
{
  "AboardApi": {
    "BaseUrl": "https://aboard.iee.ihu.gr",
    "Token": "Insert valid token",
    "InvalidToken": ""
    }
}
```

4.2.3.1 Μεθοδολογία BDD

Η συγγραφή των σεναρίων ακολούθησε τη δομή του Gherkin, η οποία χωρίζει κάθε σενάριο σε τρία τμήματα: *Given* για τον ορισμό προϋποθέσεων, *When* για την ενέργεια που εκτελείται και *Then* για την επαλήθευση των αποτελεσμάτων. Με αυτή τη δομή διασφαλίζεται πως τα σενάρια παραμένουν κατανοητά, σαφή και εύκολα συντηρήσιμα.

```
1   Feature: Announcements
2
3     This feature tests the Aboard API endpoints, tries to fetch announcements.
4
5   @public @smoke
6   Scenario: 01 - Fetch all announcements (public access)
7       When I send a GET request to "/api/v2/announcements"
8       Then server responds with 200 OK
9       And the announcements list should not be empty
10
```

Εικόνα 3: Σενάριο γραμμένο με δομή Gherking

Η χρήση φυσικής γλώσσας στις περιγραφές, αποσκοπεί στη βοήθεια της γεφύρωση του χάσματος επικοινωνίας ανάμεσα σε αναλυτές, QA engineers και developers, προκειμένου όλοι να μπορούν να κατανοήσουν τι ελέγχει το κάθε σενάριο, χωρίς να χρειαστεί να διαβάσουν τον κώδικα.

Τα σενάρια οργανώθηκαν και κατηγοριοποιήθηκαν με tags όπως @public, @token και @smoke, ώστε να επιτρέπεται η στοχευμένη εκτέλεση ανάλογα με τις ανάγκες. Για παράδειγμα, η εκτέλεση μόνο των @smoke tests είναι χρήσιμη για γρήγορο έλεγχο βασικής λειτουργικότητας πριν από ένα release, ενώ τα @token σενάρια επικεντρώνονται σε endpoints που απαιτούν αυθεντικοποίηση.

4.2.4 Υλοποίηση

Η υλοποίηση ξεκίνησε με την καταγραφή των διαθέσιμων endpoints από το documentation. Το πρώτο στάδιο περιλάμβανε τη δημιουργία απλών σεναρίων για θετικούς ελέγχους (positive scenarios), τα οποία στόχευαν να επιβεβαιώσουν την ορθή λειτουργία κάθε endpoint με έγκυρα δεδομένα. Στα σενάρια αυτά ελέγχεται ότι η απάντηση έχει σωστό HTTP status code και παράλληλα ότι το περιεχόμενο των δεδομένων ακολουθεί τη σωστή δομή. Στη συνέχεια, δημιουργήθηκαν και αρνητικά σενάρια (negative scenarios), για να ελεγχθεί η συμπεριφορά του συστήματος σε περιπτώσεις εσφαλμένων αιτημάτων ή μη εξουσιοδοτημένης πρόσβασης. Για παράδειγμα, πραγματοποιήθηκαν δοκιμές κλήσεων χωρίς authentication token ή με λανθασμένο token, όπως επίσης και κλήσεις που χρησιμοποιούσαν μη έγκυρα Ids. Αυτές οι περιπτώσεις βοήθησαν στον εντοπισμό αποκλίσεων και πιθανών βελτιώσεων στον χειρισμό σφαλμάτων από την πλευρά του API.

DeleteAnnoun...ents.feature

```
1   Feature: DeleteAnnouncements
2
3     This feature tests the Aboard API endpoints, tries to delete an announcement
4     This will fail due to unauthorized user
5
6   @token
7   Scenario: 01 - Delete a specific announcement with student token - Failing Scenario
8       When I send a DELETE request to "/api/v2/announcements/64784"
9       Then server responds with 401 Unauthorized
10      And response message is "You must have admin and/or author rights in order to continue"
11
12  @public
13  Scenario: 02 - Delete a specific announcement without token - Failing Scenario
14      When I send a DELETE request to "/api/v2/announcements/64784"
15      Then server responds with 401 Unauthorized
16      And response message is "You must sign in in order to continue"
17
```

Εικόνα 4: Αρνητικά Σενάρια

Κατά τη διάρκεια, παρατηρήθηκε ότι πολλά βήματα στα αρχεία των σεναρίων ήταν επαναλαμβανόμενα, διαφέροντας μόνο στο πεδίο ή την τιμή που επαληθευόταν. Για την αποφυγή διπλότυπου κώδικα, υλοποιήθηκε μία πιο δυναμική προσέγγιση. Τα βήματα διαμορφώθηκαν ώστε να δέχονται ως παραμέτρους το όνομα του πεδίου και την αναμενόμενη τιμή. Συνεπώς, ένα μόνο βήμα μπορεί να χρησιμοποιηθεί για πολλούς διαφορετικούς ελέγχους, μειώνοντας σημαντικά την επανάληψη κώδικα και βελτιώνοντας τη συντηρησιμότητα του project.

```
25      @token @smoke
26      Scenario: 03 - Fetch a specific announcement with valid token
27          When I send a GET request to "/api/v2/tags/159"
28          Then server responds with 200 OK
29          And the tag has property id equal to 159
30          And the tag has property title equal to Πτυχιακή/Διπλωματική Εργασία
31          And the tag has property parent_id equal to 200
32          And the tag has property is_public equal to False
33          And the tag has property maillist_name equal to theses1
34
```

Εικόνα 5: Σενάριο με διαφορετικές παραμέτρους

```
15
16      [Then(@"the tag has property (.*) equal to (.*)")]
        0 references | Rita Alexiou, 21 days ago | 1 author, 1 change
17      public void ThenTheTagHasFieldEqualTo(string fieldName, string expectedValue)
18      {
19          var response = (RestSharp.RestResponse)_scenarioContext["Response"];
20          var json = JObject.Parse(response.Content);
21          var actualValue = json["data"]?[fieldName]?.ToString();
22
23          Assert.IsNotNull(actualValue, $"Field '{fieldName}' was not found in response");
24          Assert.AreEqual(expectedValue, actualValue, $"{fieldName} doesn't match. " +
25              $"Expected: {expectedValue}, Actual: {actualValue}");
26      }
```

Εικόνα 6: Επαναχρησιμοποιήσιμο βήμα

Παράλληλα, για αποκρίσεις που περιείχαν λίστες αντικειμένων, προστέθηκε μηχανισμός ελέγχου όλων των στοιχείων της λίστας. Αυτό συντελεί στον έλεγχο ολόκληρης της απάντησης, αντί μόνο του πρώτου αντικείμενου ή ενός τυχαίου δείγματος. Η δομή και οι τιμές κάθε αντικειμένου επαληθεύονται ξεχωριστά, διασφαλίζοντας τη συνοχή και την ποιότητα των δεδομένων σε όλη την απόκριση.

Στις πιο σύνθετες περιπτώσεις, δηλαδή όταν η απόκριση του API περιείχε μεγάλο όγκο δεδομένων ή πολύπλοκη δομή, ο έλεγχος σύγκρισης γίνεται με προαποθηκευμένα JSON αρχεία. Στην προσέγγιση αυτή, η αναμενόμενη απόκριση αποθηκευόταν τοπικά σε αρχείο και γινόταν σύγκριση με το πραγματικό αποτέλεσμα, ελέγχοντας τόσο τη δομή όσο και τις τιμές. Αυτή η μέθοδος αποδείχθηκε ιδιαίτερα χρήσιμη για regression testing, καθώς οποιαδήποτε αλλαγή στο API αναδεικνύεται άμεσα μέσα από τη διαφορά μεταξύ των δύο JSON.

```
35
36      # Filter tags
37
38      @public
39      Scenario: 04 - Fetch all filter tags (public access)
40          When I send a GET request to "/api/v2/filtertags"
41          Then server responds with 200 OK
42          And the response should match the data on file "FilterTagsPublic.json"
43
44      @token @smoke
45      Scenario: 05 - Fetch all filter tags with student token
46          When I send a GET request to "/api/v2/filtertags"
47          Then server responds with 200 OK
48          And the response should match the data on file "FilterTagsAuthenticatedStudent.json"
49
```

Εικόνα 7: Τεστ με μεγάλο όγκο δεδομένων στην απάντηση

```
110
111     [Then(@"the response should match the data on file ""(.*)""")]
        0 references | Rita Alexiou, 21 days ago | 1 author, 1 change
112     public void ThenTheResponseShouldMatchTheExpectedFile(string fileName)
113     {
114         var expectedJsonPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "Data", fileName);
115         var expectedJson = File.ReadAllText(expectedJsonPath);
116         var expectedResponse = JToken.Parse(expectedJson);
117
118         var response = (RestSharp.RestResponse)_scenarioContext["Response"];
119         var actualResponse = JToken.Parse(response.Content);
120
121         Assert.IsTrue(JToken.DeepEquals(expectedResponse, actualResponse), "JSON does not match.");
122     }
```

Εικόνα 8: Βήμα που ελέγχει την απάντηση από τοπικό αρχείο

Η σταδιακή εξέλιξη του κώδικα, από την αρχική υλοποίηση με απλά και επαναλαμβανόμενα βήματα μέχρι την τελική μορφή με παραμετροποιημένα και επαναχρησιμοποιήσιμα βήματα (steps), βελτίωσε σημαντικά την ευελιξία, την αναγνωσιμότητα και την επεκτασιμότητα των δοκιμών και του κώδικα. Το τελικό αποτέλεσμα είναι μια δυναμική σουίτα API tests, ικανή να καλύψει σενάρια διαφορετικού τύπου (λειτουργικά, ασφάλειας, regression, smoke) με μικρές μόνο προσθήκες στον υπάρχοντα κώδικα.

4.2.4.1 Χρήση Hooks και ScenarioContext

Κατά την ανάπτυξη των σεναρίων, κρίθηκε απαραίτητο να υπάρχει ένας μηχανισμός κεντρικής αρχικοποίησης και διαχείρισης δεδομένων, ώστε να αποφεύγεται η επανάληψη κώδικα και να διασφαλίζεται η συνέπεια κατά την εκτέλεση των τεστ. Για τον σκοπό αυτό αξιοποιήθηκαν τα Hooks του Reqnroll, τα οποία επιτρέπουν την εκτέλεση προκαθορισμένων ενεργειών πριν ή μετά από κάθε σενάριο.

Αναλυτικότερα, χρησιμοποιήθηκαν [BeforeScenario] hooks για την προετοιμασία των δεδομένων πρόσβασης ανάλογα με τον τύπο του σεναρίου. Στα σενάρια που έχουν το tag token, πριν την εκτέλεση, αποθηκεύεται ένα έγκυρο authentication token, το οποίο στη συνέχεια χρησιμοποιείται στις κλήσεις προς τα προστατευμένα endpoints. Από την άλλη, στα σενάρια με tag public, η ίδια μεταβλητή αρχικοποιείται με κενή τιμή (ή λανθασμένη τιμή αν επεξεργαστεί στο appsettings.json), προκειμένου να ελεγχθεί η συμπεριφορά του API χωρίς αυθεντικοποίηση.

Η μεταφορά δεδομένων μεταξύ των βημάτων επιτυγχάνεται μέσω του ScenarioContext, μιας ειδικής δομής που παρέχεται από το Reqnroll, λειτουργεί ως προσωρινή «αποθήκη» για δεδομένα που ισχύουν μόνο κατά την εκτέλεση ενός σεναρίου. Σε αυτό αποθηκεύονται κρίσιμες πληροφορίες όπως η

τελευταία HTTP απόκριση (Response) ή το authentication token, ώστε να είναι διαθέσιμες σε οποιοδήποτε βήμα χρειαστεί να γίνουν έλεγχοι ή να χρησιμοποιηθούν στις επόμενες κλήσεις.

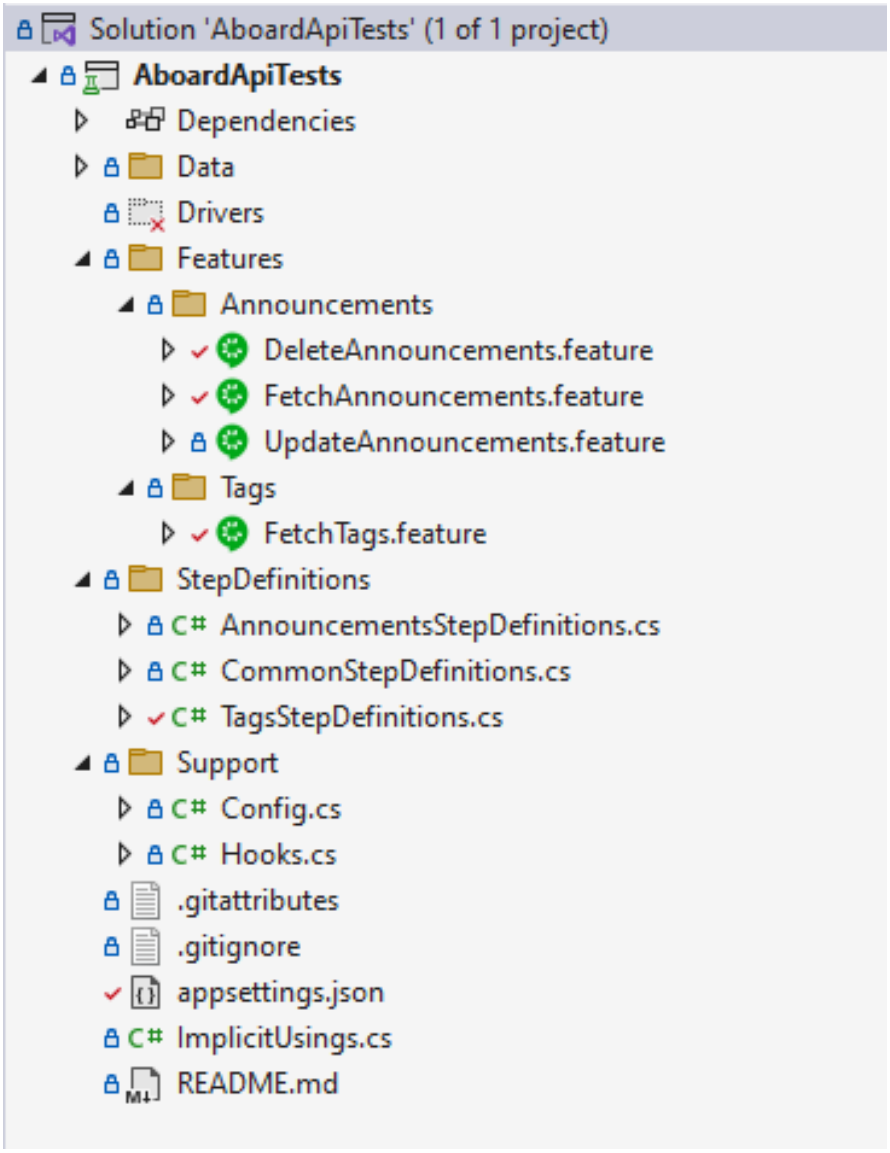
Η προσέγγιση αυτή παρέχει πολλαπλά οφέλη, μειώνει την ανάγκη για παγκόσμιες μεταβλητές, διατηρεί τον κώδικα πιο καθαρό και επαναχρησιμοποιήσιμο, ενώ παράλληλα διευκολύνει την προσαρμογή των σεναρίων όταν αλλάζουν οι απαιτήσεις ή τα δεδομένα εισόδου. Συμπερασματικά, η λογική παραμένει συγκεντρωμένη, επιτρέποντας την ευκολότερη συντήρηση και την αποφυγή ασυνεπειών μεταξύ των σεναρίων.

4.2.4.2 Τελική Δομή του Project

Η οργάνωση του project εξελίχθηκε έτσι ώστε να υποστηρίζει τη συντήρηση με ευκολία, την επαναχρησιμοποίηση του κώδικα και το σαφή διαχωρισμό αρμοδιοτήτων. Η τελική δομή περιλαμβάνει διακριτούς φακέλους και αρχεία, καθένας με συγκεκριμένο ρόλο:

- **Data**: Περιέχει τα αρχεία JSON που χρησιμοποιούνται για σύγκριση απαντήσεων API με αναμενόμενα δεδομένα. Ο διαχωρισμός των δεδομένων από τον κώδικα επιτρέπει την ενημέρωση των τιμών χωρίς αλλαγές στη λογική των tests.
- **Features**: Περιλαμβάνει τα .feature αρχεία, είναι οργανωμένα σε υποφακέλους (π.χ. Announcements, Tags) ανά κατηγορία API endpoints. Διευκολύνοντας με αυτόν τον τρόπο, τον εντοπισμό και την εκτέλεση σχετικών σεναρίων.
- **StepDefinitions**: Περιέχει τις υλοποιήσεις των βημάτων που χρησιμοποιούνται στα .feature αρχεία. Η ομαδοποίηση έγινε με βάση το αντικείμενο (π.χ. ανακοινώσεις, tags) ενώ τα κοινά βήματα βρίσκονται στο CommonStepDefinitions.cs, αποφεύγοντας την επανάληψη κώδικα.
- **Support**: Περιλαμβάνει βοηθητικές κλάσεις, όπως το Config.cs για την ανάγνωση παραμέτρων από το appsettings.json, και το Hooks.cs για ρυθμίσεις που εκτελούνται πριν ή μετά από σενάρια (π.χ. αρχικοποίηση πελάτη API).

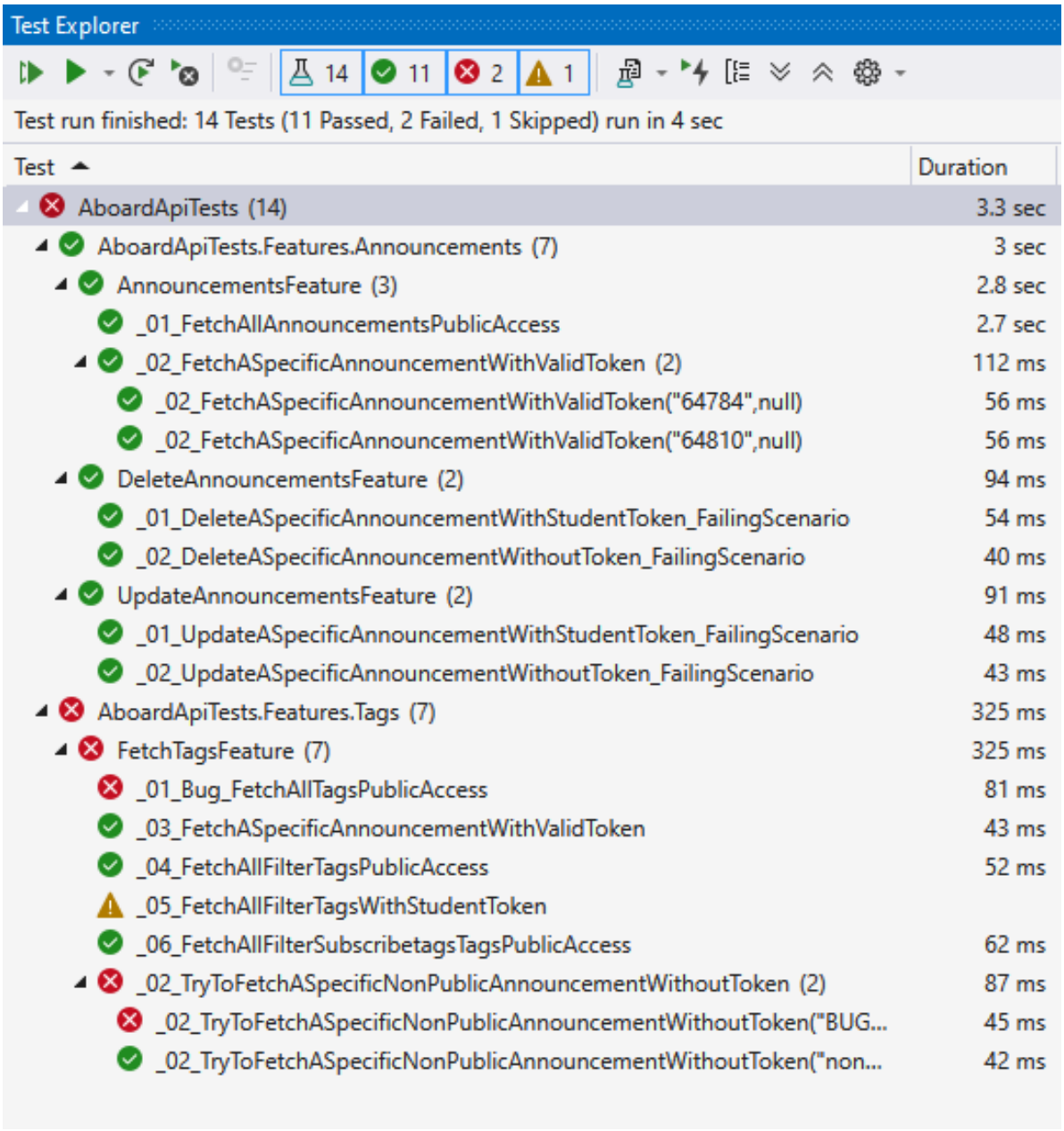
Η συγκεκριμένη δομή βελτιώνει τη διαφάνεια του κώδικα, την επεκτασιμότητα και την αποσύνδεση μεταξύ λογικής και δεδομένων. Κάθε νέα λειτουργικότητα μπορεί να προστεθεί εύκολα, χωρίς να απαιτούνται αλλαγές σε άσχετα σημεία του project.



Εικόνα 9: Τελική Δομή

4.2.5 Αποτελέσματα και παρατηρήσεις

Η εκτέλεση των 14 σεναρίων ολοκληρώθηκε σε περίπου 4 δευτερόλεπτα, επιβεβαιώνοντας την ταχύτητα και την αποδοτικότητα του API testing. Από τα σενάρια αυτά, 11 ολοκληρώθηκαν με επιτυχία, 2 απέτυχαν και 1 αγνοήθηκε σκοπίμως.



Εικόνα 10: Αποτελέσματα API

Οι δύο αποτυχίες σχετίζονται με συμπεριφορές του API που, ενώ δεν παραβιάζουν άμεσα τις προδιαγραφές, μπορούν να θεωρηθούν ασυνέπειες ή κενά (gaps) στο documentation και όχι στην υλοποίηση. Συγκεκριμένα:

- Στο σενάριο Fetch all tags (public access), η αρχική εκτίμηση του tester ήταν ότι όλα τα tags που επιστρέφονται δημόσια θα πρέπει να έχουν την ιδιότητα is_public με τιμή true, ενώ στην απάντηση επιστράφηκαν αποτελέσματα όπου η τιμή ήταν false. Ωστόσο, μετά από συζήτηση με τους υπεύθυνους ανάπτυξης διευκρινίστηκε ότι η ιδιότητα is_public δεν αφορά το ίδιο το tag, αλλά τις ανακοινώσεις που συνδέονται με αυτό. Συνεπώς, το ίδιο το tag μπορεί να εμφανιστεί, αλλά οι μη δημόσιες ανακοινώσεις που το χρησιμοποιούν δεν θα είναι ορατές χωρίς token.

- Αντίστοιχα, στο σενάριο Try to fetch a specific non-public tag without token, η αναμενόμενη συμπεριφορά θεωρήθηκε πως ήταν η επιστροφή 404 Not Found, ενώ στην πράξη το API απάντησε με 200 OK. Και σε αυτή την περίπτωση, η συμπεριφορά δεν παραβιάζει την πραγματική λογική του συστήματος, αλλά οφείλεται σε παρανόηση των προδιαγραφών.

appsettings.json | CommonStepDefinitions.cs | FetchTags.feature | TagsStepDefinitions.cs

```
1   Feature: FetchTags
2
3     This feature tests the Aboard API endpoints, tries to fetch tags.
4
5   # Tags
6
7   # failing scenario - bug
8   @public
9   Scenario: 01 - Bug - Fetch all tags (public access)
10      When I send a GET request to "/api/v2/tags"
11      Then server responds with 200 OK
12      And the tags list should not be empty
13      And the tags list has the property is_public equal to True
14
15  # failing scenario - bug
16  @public
17  Scenario: 02 - Try to Fetch a specific non public announcement without token
18      When I send a GET request to "/api/v2/tags/<tagId>"
19      Then server responds with 404 Not Found
20  Examples:
21  | scenario                        | tagId |
22  | BUG - existing id but not public | 25    |
23  | non existing id                  | 9999  |
```

Εικόνα 11: Τεστ που αποτυγχάνουν

Run | Debug

Test Detail Summary

_01_Bug_FetchAllTagsPublicAccess

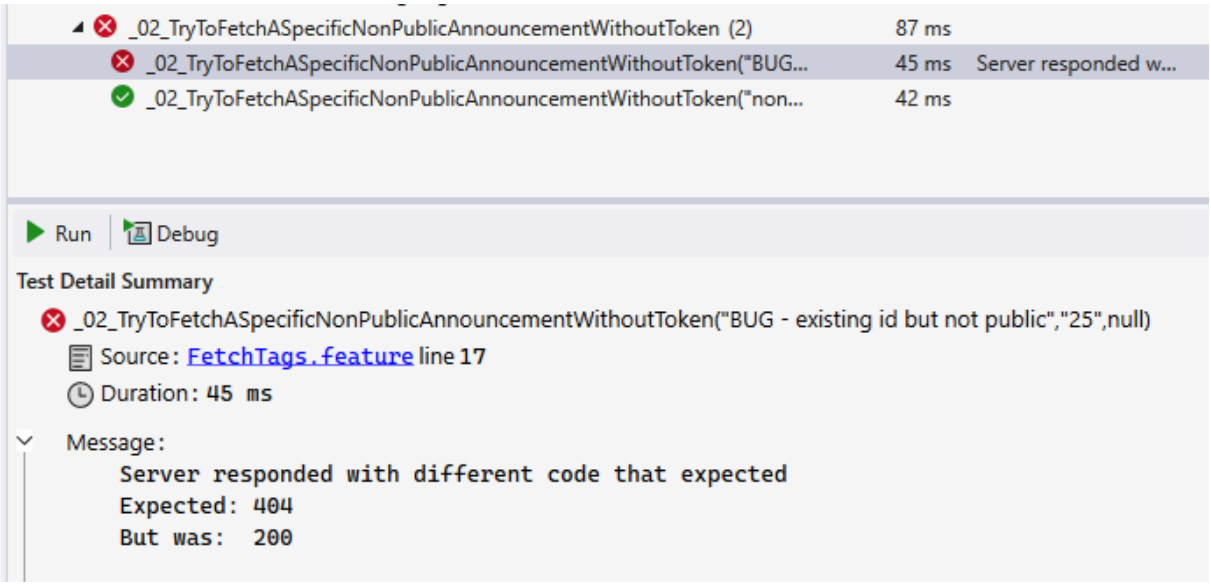
Source: FetchTags.feature line 9

Duration: 81 ms

Message:

```
is_public doesn't match. Expected: True, Actual: False
Expected string length 4 but was 5. Strings differ at index 0.
Expected: "True"
But was:  "False"
-----------^
```

Εικόνα 12: Μήνυμα αποτυχίας 1



Εικόνα 13: Μήνυμα αποτυχίας 2

Αυτά τα παραδείγματα δείχνουν ότι πέρα από τον εντοπισμό λειτουργικών σφαλμάτων, το testing συμβάλλει και στην ανάδειξη σημείων όπου το documentation χρειάζεται βελτίωση ώστε να μειωθούν τα λάθη κατανόησης και τα λανθασμένες υποθέσεις. Η καταγραφή αυτών των κενών είναι εξίσου σημαντική με την εύρεση πραγματικών bugs, καθώς ενισχύει τη διαφάνεια και την ακρίβεια στην επικοινωνία μεταξύ ομάδων ανάπτυξης και testing.

Σχετικά με το σενάριο Fetch all filter tags with student token, πήρε το tag ignore. Η λογική του βασίζεται σε σύγκριση της πλήρους απάντησης με ένα αποθηκευμένο τοπικό JSON αρχείο. Η μέθοδος αυτή λειτούργησε αρχικά, όμως η δυναμική φύση των δεδομένων (π.χ. προσθήκη νέων tags στην πραγματική βάση) οδήγησε σε αποκλίσεις που δεν σχετίζονται με σφάλματα του API, αλλά με την αλλαγή του περιεχομένου. Συμπερασματικά, το συγκεκριμένο σενάριο παραμένει αγνοημένο προς το παρόν και θεωρείται κατάλληλο για περιπτώσεις όπου τα δεδομένα είναι σταθερά ή ελεγχόμενα. Η χρήση του ignore σε σενάρια που βασίζονται σε δυναμικά δεδομένα είναι μια πρακτική προσέγγιση, καθώς αποτρέπει την καταγραφή ψευδώς αποτυχημένων tests που δεν ανακαλύπτουν πραγματικά σφάλματα. Μπορεί να ενεργοποιηθεί ξανά σε περιβάλλοντα staging ή σε περιπτώσεις όπου η βάση δεδομένων είναι σταθερή, ώστε να αξιοποιηθεί πλήρως η δυνατότητα σύγκρισης με τοπικά JSON αρχεία.

4.2.6 Συμπεράσματα

Το API testing στο Aboard ανέδειξε την αξία του ελέγχου backend όσον αφορά την επιβεβαίωση της ορθότητας και της συνέπειας των δεδομένων, αλλά ταυτόχρονα και για την έγκαιρη ανίχνευση πιθανών λειτουργικών κενών (gaps) που δεν καταγράφονται ρητά στο API documentation. Μέσα από την εξέταση θετικών και αρνητικών σεναρίων, προέκυψαν περιπτώσεις όπου η πραγματική συμπεριφορά του API δεν ταυτίστηκε με τις αρχικές προσδοκίες του tester. Για παράδειγμα, η επιστροφή tags με is_public με τιμή false ή η απόκριση 200 OK αντί για 404 Not Found σε συγκεκριμένα αιτήματα, αν και δεν αποτελούν σφάλματα, ανέδειξαν σημεία όπου το documentation θα μπορούσε να είναι πιο σαφές, ώστε να αποφεύγονται λανθασμένες υποθέσεις κατά τον σχεδιασμό των tests.

Η επιλογή της BDD προσέγγισης με το Reqnroll διευκολύνει τη σαφή απόδοση και την ευκολότερη κατανόηση των σεναρίων, ενώ η υλοποίηση δυναμικών βημάτων και η παραμετροποίηση μέσω appsettings.json αυξάνουν την επαναχρησιμοποίηση και τη μείωση της πολυπλοκότητας του κώδικα. Η

τελική δομή του project αποφέρει καθαρό διαχωρισμό ευθυνών, ευελιξία στην εκτέλεση (μέσω tagging όπως public, token, smoke, ignore) και ευκολότερη συντήρηση στο μέλλον.

Παρά τον περιορισμό στην πλήρη αυτοματοποίηση του authentication, η υλοποίηση πετυχαίνει υψηλή κάλυψη ελέγχων και παρέχει χρήσιμες πληροφορίες για τη σταθερότητα και τη συνέπεια του API. Στο μέλλον, η ενσωμάτωση μηχανισμών αυτόματης λήψης/ανανέωσης του token και η αξιοποίηση τεχνικών AI-driven testing μπορούν να αυξήσουν περαιτέρω το επίπεδο αυτοματοποίησης, ενώ η υιοθέτηση πιο σταθερών δεδομένων θα αποφέρει πιο αξιόπιστη σύγκριση απαντήσεων σε σενάρια που σήμερα επηρεάζονται από δυναμικά δεδομένα.

4.3 UI Testing - Selenium

4.3.1 Εισαγωγή

Ακολουθώντας τον έλεγχο του Aboard API στο προηγούμενο κεφάλαιο, η παρούσα ενότητα επικεντρώνεται στον έλεγχο της ίδιας πλατφόρμας, σε επίπεδο γραφικού περιβάλλοντος χρήστη (UI). Το UI testing αποτελεί αναπόσπαστο μέρος της διαδικασίας διασφάλισης ποιότητας, διότι ελέγχει την αλληλεπίδραση των χρηστών με το σύστημα, τη λειτουργικότητα των στοιχείων της διεπαφής και την ευχρηστία της εφαρμογής. Από τη μία πλευρά, το API testing προσφέρει ταχύτητα και ακρίβεια σε επίπεδο backend, και από την άλλη το UI testing λειτουργεί συμπληρωματικά με σκοπό την επιβεβαίωση ότι οι βασικές ροές χρήσης λειτουργούν ομαλά και ότι η εμπειρία του χρήστη παραμένει συνεπής και αξιόπιστη.

Ο έλεγχος του UI στην πλατφόρμα Aboard, έχει ιδιαίτερη σημασία επειδή η αλληλεπίδραση των χρηστών γίνεται κατά κύριο λόγο μέσω του φυλλομετρητή (browser). Η σελίδα ανακοινώσεων, που παρουσιάστηκε στο Κεφάλαιο 4, αποτελεί το κεντρικό σημείο πληροφόρησης και, επομένως, η σωστή λειτουργία των βασικών δυνατοτήτων της είναι κρίσιμη για την αξιοπιστία της πλατφόρμας. Τέτοιες λειτουργίες είναι η αναζήτηση, το φιλτράρισμα, η ταξινόμηση και η πλοήγηση.

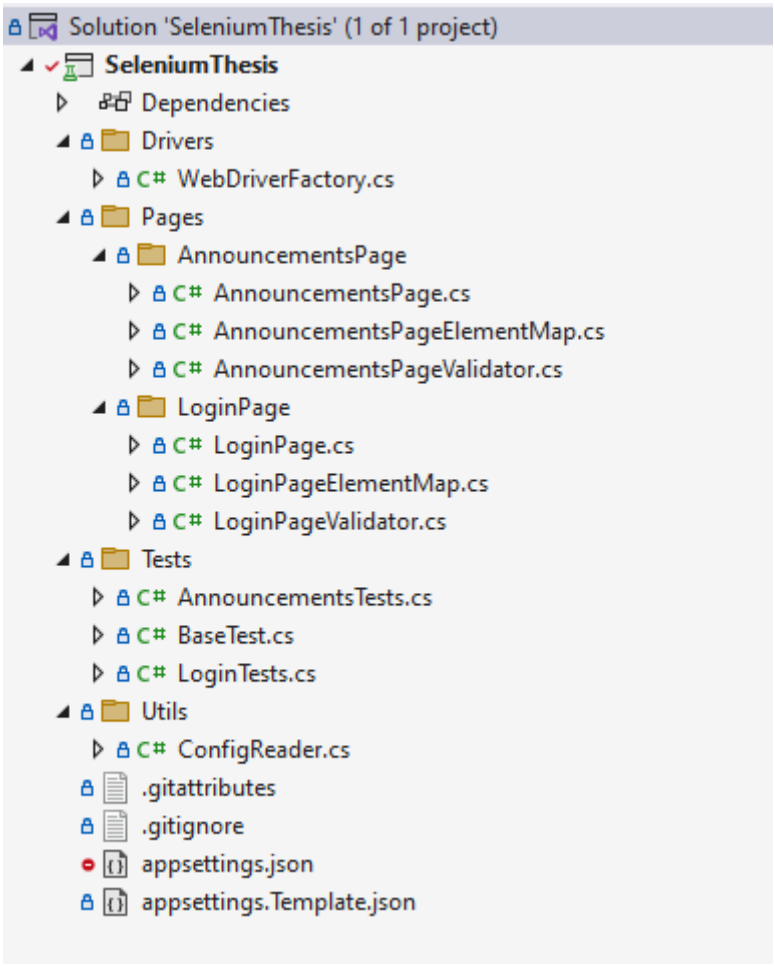
Για το UI testing, χρησιμοποιήθηκε το Selenium WebDriver 4, σε συνδυασμό με C# και .NET, ενώ η εκτέλεση και η οργάνωση των tests γίνεται με το framework NUnit. Η ανάπτυξη ακολουθεί τη σχεδιαστική αρχή του Page Object Model (POM), έχοντας σκοπό στη μείωση της πολυπλοκότητας και στη βελτίωση της συντηρησιμότητας του κώδικα. Στις επόμενες ενότητες θα παρουσιαστούν οι τεχνολογίες, η δομή του project, τα υλοποιημένα σενάρια και οι βασικές σχεδιαστικές αποφάσεις που καθόρισαν τη σταθερότητα και την αποτελεσματικότητα της σουίτας δοκιμών.

4.3.2 Σχεδιασμός και Τεχνολογίες

Όπως αναφέρθηκε, έγινε χρήση του Selenium WebDriver 4, ένα από τα πιο διαδεδομένα εργαλεία για την αυτοματοποίηση UI test, καθώς υποστηρίζει πολλούς διαφορετικούς browsers με ενιαίο τρόπο. Η επιλογή της C# αιτιολογείται αφενός από την καλή ενσωμάτωσή της με testing frameworks όπως το NUnit, αφετέρου από την εξοικείωσης της φοιτήτριας. Το NUnit είναι ένα testing framework ανοιχτού κώδικα και χρησιμοποιήθηκε ως test runner, για την εκτέλεση και τη διαχείριση των test.

Το σχεδιαστικό μοντέλο που εφαρμόστηκε είναι το Page Object Model (POM). Ακολουθώντας αυτή την προσέγγιση, ο διαχωρισμός του κώδικα έγινε σε τρία μέρη: τα *ElementMap.cs, τα *Pages.cs και τα *Validators.cs. Στο πρώτο, βρίσκονται οι locators που χρειάζονται για να αλληλεπιδράσει το

σύστημα με τις ιστοσελίδες. Τα Pages αντιστοιχούν στις σελίδες στις οποίες πλοηγείται ο χρήστης (για παράδειγμα η σελίδα LoginPage.cs και η βασική σελίδα AnnouncementsPage.cs), και περιέχουν μεθόδους για βασικές λειτουργίες όπως το πάτημα ενός κουμπιού. Στις validator κλάσεις, βρίσκονται οι απαραίτητες μέθοδοι για τους ελέγχους και επαληθεύσεις· λόγου χάρη, μετά την εφαρμογή ενός φίλτρου επιβεβαιώνεται ότι οι εμφανιζόμενες ανακοινώσεις ακολουθούν το κριτήριο.



Εικόνα 14: Δομή

Με την επιλογή αυτών των τεχνολογιών και αρχών σχεδίασης, η σουίτα των δοκιμών εξασφάλισε μεγαλύτερη σταθερότητα, ευκολότερη συντήρηση και δυνατότητα επέκτασης. Αυτό σημαίνει ότι μικρές αλλαγές στο UI δεν απαιτούν εκτεταμένες τροποποιήσεις στον κώδικα των tests, μειώνοντας τον χρόνο συντήρησης και τον κίνδυνο εμφάνισης σφαλμάτων.

4.3.3 Δομή του Project

Τα επιμέρους test βρίσκονται στις σελίδες AnnouncementsTests.cs και LoginTests.cs. Αυτές οι κλάσεις κληρονομούν από την BaseTest.cs, η οποία περιέχει τον κεντρικό κώδικα για την αρχικοποίηση (setup) του browser και του WebDriver, και στο τέλος, τον καθαρισμό (teardown). Μέσα σε αυτήν, υπάρχει μία ενότητα (Shared Steps) η οποία περιέχει βήματα που είναι κοινά και επαναλαμβανόμενα όπως η σύνδεση του χρήστη. Με τον τρόπο αυτό, αποφεύγεται ο κίνδυνος να υπάρξουν ασυνέπειες μεταξύ διαφορετικών tests, καθώς όλα κληρονομούν την ίδια βασική ρύθμιση και συμπεριφέρονται με ομοιογένεια.

Η μέθοδος LoginUser(), για παράδειγμα, τοποθετήθηκε μέσα στη BaseTest αντί να επαναλαμβάνεται σε κάθε test ξεχωριστά. Ο λόγος είναι ότι, σε πολλά σενάρια, η σύνδεση του χρήστη δεν αποτελεί το αντικείμενο της δοκιμής, αλλά ένα απαραίτητο προαπαιτούμενο. Αν τα βήματα του login επαναλαμβάνονταν σε κάθε test, θα υπήρχε μεγάλος βαθμός επαναληψιμότητας, αυξημένη ευθραυστότητα και οποιαδήποτε αλλαγή στη ροή σύνδεσης θα απαιτούσε τροποποίηση σε πολλά σημεία, που συνεπάγεται σε δυσανάγνωστα σενάρια. Ακολουθώντας την αρχή DRY (Don't Repeat Yourself), το project είναι ευανάγνωστο, εύκολα συντηρήσιμο και επεκτάσιμο.

```
public class BaseTest
    {

        protected IWebDriver _driver;

        [SetUp]
        public void SetUp()
        {
            _driver = WebDriverFactory.CreateChromeDriver();
        }

        [TearDown]
        public void TearDown()
        {
            _driver.Dispose();
        }

        // Shared Steps

        protected void LoginUser()
        {
            _driver.Navigate().GoToUrl(ConfigReader.BaseUrl);
            var loginPage = new LoginPage(_driver);
            loginPage.ClickSignInButton();
            loginPage.LoginAs(ConfigReader.ValidUsername,
                        ConfigReader.ValidPassword);
        }
    }
```

Εικόνα 15: BaseTest Κλάση

Η κλάση WebDriverFactory υλοποιεί το σχεδιαστικό μοτίβο Factory (Factory Pattern) και είναι υπεύθυνη για τη δημιουργία instances του επιλεγμένου browser (Chrome, Edge, Firefox κ.λπ.), τη ρύθμιση των διαθέσιμων επιλογών και τη διαχείριση της δημιουργίας των drivers.

Στον αυτόματο έλεγχο διεπαφής, είναι πολύ σημαντική η στρατηγική συγχρονισμού που θα χρησιμοποιηθεί. Πολλές σελίδες φορτώνουν τα αντικείμενα ασύγχρονα, προκειμένου να μην είναι διακριτή η καθυστέρηση στον χρήστη, αυτό όμως μπορεί να δημιουργήσει προβλήματα στον αυτόματο κώδικα, καθώς μια πρόωρη προσπάθεια αλληλεπίδρασης με ένα στοιχείο που δεν έχει φορτωθεί σωστά, θα οδηγήσει σε σφάλμα (exception). Στην συγκεκριμένη υλοποίηση, επιλέχθηκε το WebDriverWait αντί για Thread.Sleep. Με αυτόν τον τρόπο δίνεται ένα ευέλικτο περιθώριο μέσα στο οποίο η εφαρμογή περιμένει να ικανοποιηθεί μια συνθήκη, σε αντίθεση με τη ρητή καθυστέρηση σταθερού χρόνου. Με αυτή την προσέγγιση, το test γίνεται δυναμικό και λιγότερο επιρρεπές σε σφάλματα.

```
public AnnouncementsPage(IWebDriver driver)
{
     _driver = driver;
     _wait = new WebDriverWait(driver, TimeSpan.FromSeconds(8));
}
```

Εικόνα 16: Αρχικοποίηση του WebDriverWait στη σελίδα AnnouncementsPage.cs

Η αρχικοποίηση του WebDriverWait γίνεται όπως στην εικόνα παραπάνω. Δημιουργείται ένα αντικείμενο αναμονής με όνομα _wait (explicit wait), δέχεται δύο ορίσματα: τον driver και τον μέγιστο χρόνο μέχρι να πραγματοποιηθεί μία συνθήκη.

```
_wait.Until(e => e.FindElements(_by.PinnedAnnouncements).Count > 0);

var activePageNumber = _wait.Until(ExpectedConditions.ElementIsVisible(_by.
ActivePage)).Text;
```

Εικόνα 17: Παραδείγμα χρήσης του WebDriverWait στη σελίδα AnnouncementsPageValidator.cs

Στο πρώτο παράδειγμα, η συνθήκη περιμένει μέχρι να εντοπιστεί τουλάχιστον μία εμφάνιση του στοιχείο με όνομα PinnedAnnouncements. Στο δεύτερο, η συνθήκη περιμένει μέχρι το στοιχείο ActivePage να γίνει ορατό (visible) και έπειτα αποθηκεύει το κείμενο του σε μία μεταβλητή. Ο χρόνος αναμονής είναι έως 8 δευτερόλεπτα, όπως ορίστηκε προηγουμένως. Χρησιμοποιώντας αυτή την λειτουργικότητα, το τεστ είναι ταχύτερο (δεν περιμένει πάντα το πλήρες χρονικό περιθώριο αν η συνθήκη ικανοποιηθεί νωρίτερα), πιο αξιόπιστο και πιο ευέλικτο, ώστε να προσαρμόζεται στις αλλαγές της σελίδας σε πραγματικό χρόνο.

Η σωστή χρήση των waits συνδέεται άμεσα με τον τρόπο εντοπισμού των στοιχείων. Για να εξασφαλιστεί ότι τα tests παραμένουν σταθερά και επαναχρησιμοποιήσιμα, κρίθηκε αναγκαίο να ακολουθηθεί συγκεκριμένη στρατηγική για την οργάνωση των locators.

Ο εντοπισμός των στοιχείων στη σελίδα γίνεται με τη χρήση Locators τύπου By (π.χ. CSS/Xpath),οι οποίοι περιγράφουν πώς θα βρεθεί ένα στοιχείο και δίνονται ως ορίσματα στις μεθόδους εντοπισμού FindElement(s) και στα waits. Η προσέγγιση αυτή είναι σύμφωνη με την τεκμηρίωση του Selenium, που ορίζει τους locators ως τον ενδεδειγμένο τρόπο ταυτοποίησης στοιχείων και προτείνει να δηλώνονται ρητά και χωριστά από τον κώδικα που τα χρησιμοποιεί. Με αυτόν τον τρόπο, το framework μπορεί να επαναεντοπίζει τα στοιχεία, κάτι που ταιριάζει ιδανικά με τα explicit waits και μειώνει την εξάρτηση από παλαιές/δεσμευμένες αναφορές.

```
public By FilterButton => By.CssSelector("div.filters-actions > button");
public By SearchByTitleBox => By.Id("search-title");
public By SelectTagsTree => By.ClassName("tree-controller-container");
public By PerPageDropdown =>
        By.XPath("//label[contains(.,'Ανά σελίδα')]/following::div[contains
      (@class,'dropdown-indicator')]");
```

Εικόνα 18: Χρήση locators με By στη σελίδα AnnouncementsPageElementMap.cs

Σε αντίθεση με ένα IWebElement, που αποτελεί ήδη αναφορά σε στοιχείο του τρέχοντος DOM και μπορεί να καταστεί «stale» μετά από ανανέωση ή δυναμική αλλαγή της σελίδας (προκαλώντας το σφάλμα StaleElementReferenceException), η χρήση του By επιτρέπει την εκ νέου αναζήτηση του στοιχείου πριν από κάθε ενέργεια. Με αυτόν τον τρόπο, το framework μπορεί να βρίσκει ξανά τα

στοιχεία δυναμικά, γεγονός που ταιριάζει ιδανικά με την εφαρμογή explicit waits και βελτιώνει σημαντικά τη σταθερότητα και τη συντηρησιμότητα του κώδικα.

4.3.3.1 Φάκελος Utils και αρχείο appsettings.json

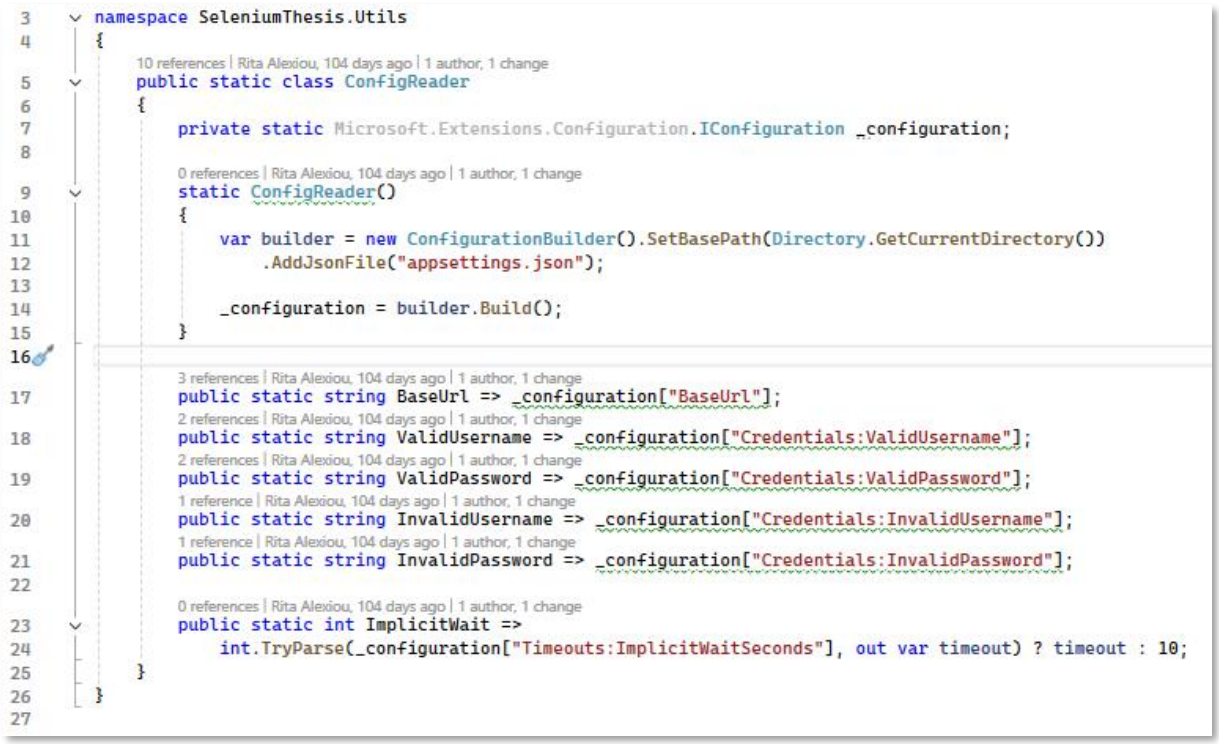
Ένα ακόμη στοιχείο που προστέθηκε στη δομή του project είναι ο φάκελος Utils, μαζί με τη χρήση του αρχείου appsettings.json. Η ανάγκη για αυτά τα δύο, προέκυψε από πρακτικά ζητήματα που εμφανίζονται συχνά σε αυτοματοποιημένα tests. Όταν τιμές όπως URLs, credentials ή χρόνοι αναμονής γράφονται απευθείας στον κώδικα (hard-coded), τα tests γίνονται εύθραυστα και δύσκολα στη συντήρηση. Ακόμη περισσότερο, διαφορετικά περιβάλλοντα εκτέλεσης (π.χ. τοπικός υπολογιστής ή CI server) μπορεί να απαιτούν διαφορετικές ρυθμίσεις, κάτι που θα ήταν πολύπλοκο αν έπρεπε να αλλάζει συνεχώς ο κώδικας.

Με τη χρήση του appsettings.json δημιουργήθηκε ένα ενιαίο, ευανάγνωστο σημείο στο οποίο αποθηκεύονται όλες οι βασικές ρυθμίσεις: το BaseUrl, οι χρόνοι αναμονής (timeouts), ο browser που θα χρησιμοποιηθεί, αλλά και τα credentials για δοκιμαστικούς λογαριασμούς. Για λόγους ασφάλειας, το πραγματικό αρχείο appsettings.json δεν παραμένει στο repository. Αντίθετα, καταχωρείται μόνο ένα appsettings.Template.json που λειτουργεί ως παράδειγμα σχήματος, ενώ το πραγματικό αρχείο αγνοείται μέσω .gitignore. Με αυτόν τον τρόπο, τα ευαίσθητα δεδομένα και τις μηχανικά εξαρτώμενες ρυθμίσεις, μένουν εκτός ελέγχου εκδόσεων, χωρίς όμως να χάνεται η τεκμηρίωση των απαιτούμενων πεδίων.

```
{
  "BaseUrl": "https://aboard.iee.ihu.gr/announcements",
  "Credentials": {
    "ValidUsername": "addvalidusername",
    "ValidPassword": "addvalidpassword",
    "InvalidUsername": "wrongusername",
    "InvalidPassword": "wrongpass"
  },
  "Browser": "Chrome",
  "Timeouts": {
    "ImplicitWaitSeconds": 10
  }
}
```

Εικόνα 19: Appsettings.Template.json

Η ανάγνωση των ρυθμίσεων υλοποιείται μέσω της .NET configuration stack (Microsoft.Extensions.Configuration). Ο ConfigReader, που βρίσκεται στον φάκελο Utils, αξιοποιεί έναν ConfigurationBuilder και εκθέτει strongly-typed getters όπως ConfigReader.BaseUrl, ConfigReader.ValidUsername, ConfigReader.ImplicitWait κ.ά. Έτσι, δεν υπάρχουν πλέον «μαγικές τιμές» στον κώδικα, μια αλλαγή στο appsettings.json αρκεί για να προσαρμοστεί όλη η σουίτα δοκιμών. Επιπλέον, η δομή αυτή δίνει τη δυνατότητα να προστεθούν αργότερα διαφορετικά αρχεία ρυθμίσεων (π.χ. appsettings.Local.json, appsettings.CI.json) και να επιλεγούν με βάση μεταβλητές περιβάλλοντος.



```
namespace SeleniumThesis.Utils
{
    10 references | Rita Alexiou, 104 days ago | 1 author, 1 change
    public static class ConfigReader
    {
        private static Microsoft.Extensions.Configuration.IConfiguration _configuration;

        0 references | Rita Alexiou, 104 days ago | 1 author, 1 change
        static ConfigReader()
        {
            var builder = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("appsettings.json");

            _configuration = builder.Build();
        }

        3 references | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static string BaseUrl => _configuration["BaseUrl"];
        2 references | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static string ValidUsername => _configuration["Credentials:ValidUsername"];
        2 references | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static string ValidPassword => _configuration["Credentials:ValidPassword"];
        1 reference | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static string InvalidUsername => _configuration["Credentials:InvalidUsername"];
        1 reference | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static string InvalidPassword => _configuration["Credentials:InvalidPassword"];

        0 references | Rita Alexiou, 104 days ago | 1 author, 1 change
        public static int ImplicitWait =>
            int.TryParse(_configuration["Timeouts:ImplicitWaitSeconds"], out var timeout) ? timeout : 10;
    }
}
```

Εικόνα 20: ConfigReader.cs

Ο φάκελος Utils γενικά φιλοξενεί βοηθητικές λειτουργίες που χρησιμοποιούνται από πολλά σημεία του κώδικα, αλλά δεν ανήκουν σε συγκεκριμένες σελίδες. Στην τρέχουσα μορφή περιέχει μόνο τον ConfigReader, αλλά μελλοντικά μπορεί να επεκταθεί με κοινές μεθόδους explicit waits, με μηχανισμό λήψης screenshots σε περίπτωση αποτυχίας ή με retry helpers για την αντιμετώπιση ασταθών συμπεριφορών του UI. Με αυτόν τον τρόπο, οι Page classes παραμένουν καθαρές και επικεντρωμένες στις ενέργειες του χρήστη, ενώ οι διασταυρούμενες λειτουργίες συγκεντρώνονται σε ένα κοινό σημείο.

Η προσέγγιση αυτή ενισχύει την αρχιτεκτονική του project: οι Pages οργανώνονται σε ElementMap, Page και Validator για να διαχειρίζονται selectors, αλληλεπιδράσεις και επαληθεύσεις αντίστοιχα. Τα Tests εκφράζουν απλώς σενάρια χωρίς να ασχολούνται με timeouts ή credentials, οι Drivers ασχολούνται με τη δημιουργία και ρύθμιση του WebDriver, και τέλος τα Utils αναλαμβάνουν τις ρυθμίσεις και τα κοινά helpers. Έτσι το project αποκτά καθαρό διαχωρισμό ευθυνών και μένει πιο ευέλικτο για μελλοντική επέκταση.

4.3.4 Σενάρια και Υλοποίηση

Η υλοποίηση των αυτοματοποιημένων σεναρίων επικεντρώθηκε στις δύο βασικές σελίδες: τη σελίδα με τον έλεγχο μηχανισμού σύνδεσης (login) και στον έλεγχο της λειτουργικότητας της σελίδας ανακοινώσεων (announcements).

4.3.4.1 Σενάρια Μηχανισμού Σύνδεσης

Η σελίδα αυτή είναι σημαντική καθώς διαχωρίζει τους χρήστες και τα δικαιώματά τους. Τα σενάρια που γίνονται με επιτυχή σύνδεση είναι με τα διαπιστευτήρια ενός φοιτητή. Δημιουργήθηκαν δύο βασικά σενάρια: στο ένα γίνεται επιβεβαίωση ότι ο χρήστης με έγκυρα οδηγείται στην αρχική σελίδα και εμφανίζεται το μενού λογαριασμού, στο δεύτερο ελέγχεται ότι η προσπάθεια σύνδεσης με λανθασμένα διαπιστευτήρια, δεν συνδέει τον χρήστη και του εμφανίζει κατάλληλο μήνυμα σφάλματος.

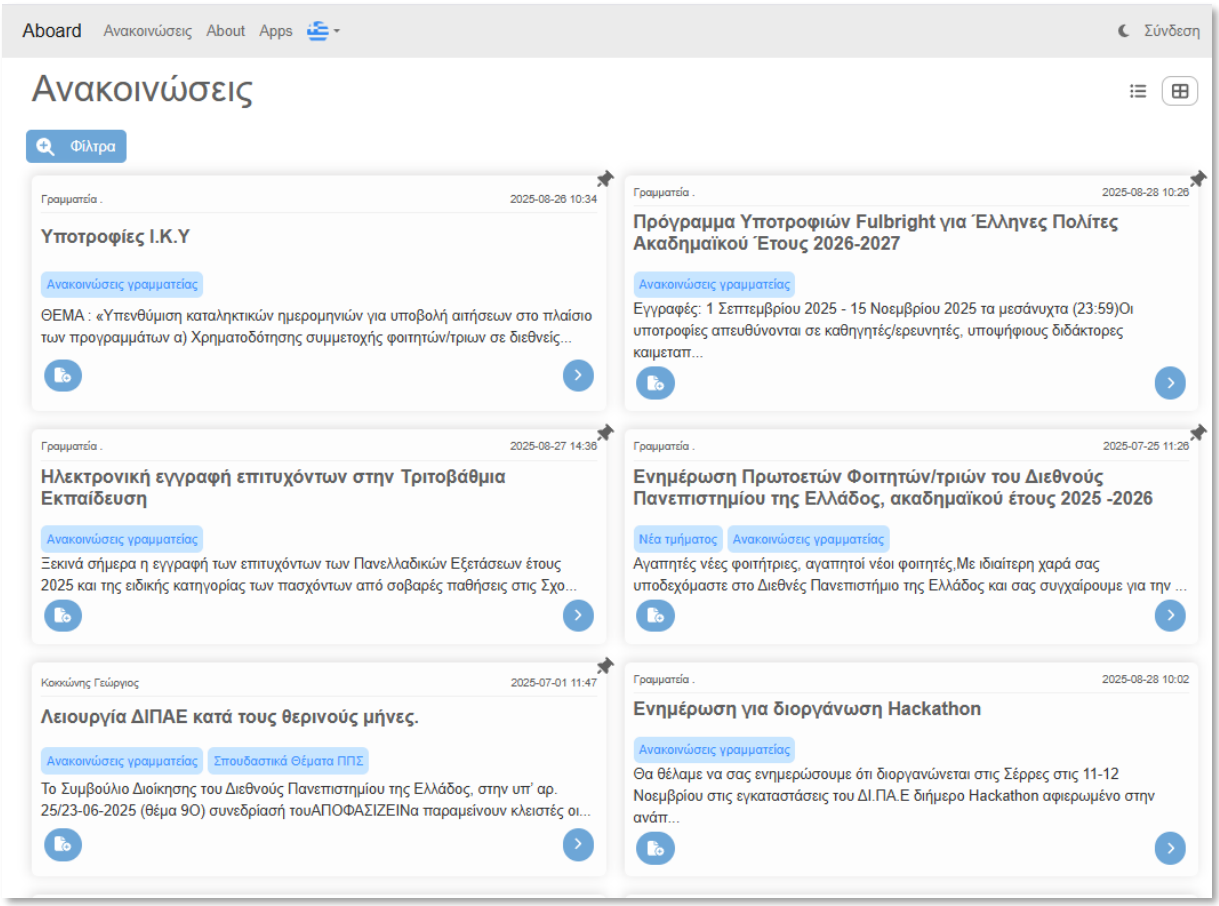
```
public class LoginTests : BaseTest
{
    /// <summary>
    /// Navigates to site, Logins with valid credentials and verifies that login was successful
    /// </summary>
    [Test]
    public void ValidCredentials_SuccessLogin()
    {
        _driver.Navigate().GoToUrl(ConfigReader.BaseUrl);

        var loginPage = new LoginPage(_driver);

        loginPage.ClickSignInButton();

        loginPage.LoginAs(ConfigReader.ValidUsername, ConfigReader.ValidPassword);

        var validator = new LoginPageValidator(_driver);

        validator.AssertUserIsLoggedIn();
    }

    /// <summary>
    /// Navigates to site, Logins with invalid credentials and verifies that the error is visible
    /// and that contains the expected error message
    /// </summary>
    [Test]
    public void InvalidCredentials_LoginError()
    {
        _driver.Navigate().GoToUrl(ConfigReader.BaseUrl);

        var loginPage = new LoginPage(_driver);

        loginPage.ClickSignInButton();

        loginPage.LoginAs(ConfigReader.InvalidUsername, ConfigReader.InvalidPassword);

        var validator = new LoginPageValidator(_driver);

        validator.AssertErrorMessageIsVisible();
    }

}
```

Εικόνα 21: Login Tests

Στα σενάρια αυτά, το αντικείμενο ελέγχου είναι η ροή αυθεντικοποίησης και τα βήματα σύνδεσης του χρήστη, επομένως παρακάμπτεται η χρήστη του κοινού βήματος LoginUser() που παρέχεται στη BaseTest.cs ώστε να επιβεβαιωθεί με ακρίβεια η λειτουργία του μηχανισμού.

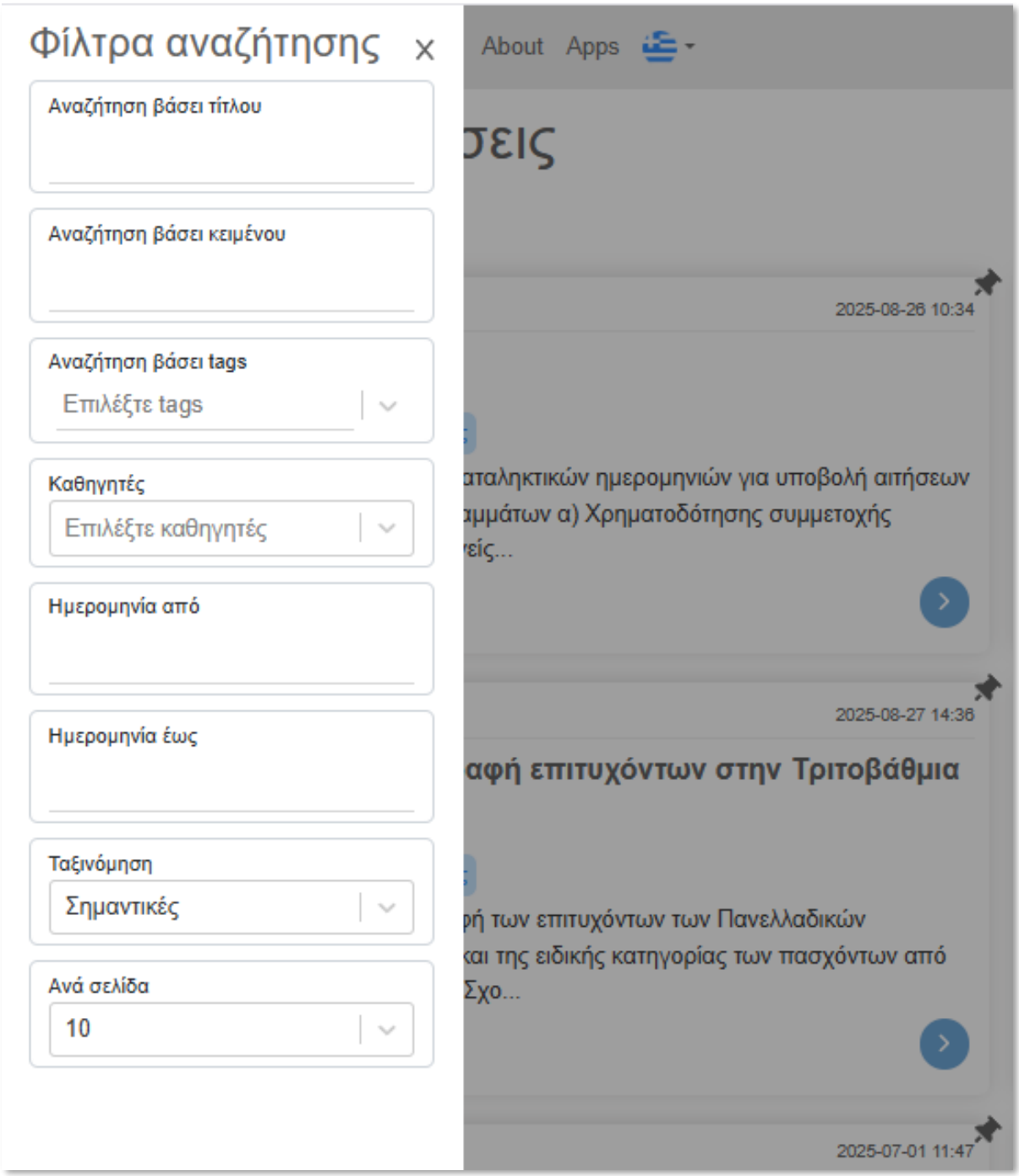
4.3.4.2 Σενάρια στη σελίδα Ανακοινώσεων

Η κεντρική σελίδα ανακοινώσεων έχει προεπιλεγμένη επιλογή εμφάνισης «τα σημαντικά πρώτα», δηλαδή οι ανακοινώσεις που έχουν καρφιτσωθεί είναι πάντα στην κορυφή, ακολουθούμενες από τις υπόλοιπες σε χρονολογική σειρά (από τις νεότερες προς τις παλαιότερες). Στα αριστερά της σελίδας, υπάρχει το κουμπί «Φίλτρα», το οποίο στο πάτημά του ανοίγει ένα μενού στο οποίο ο χρήστης μπορεί να ορίσει διαφορετικά φίλτρα ή και συνδυασμούς αυτών.



Εικόνα 22: Κεντρική σελίδα Ανακοινώσεων

Στα σενάρια αυτής της ενότητας γίνεται χρήση της κοινής μεθόδου LoginUser(), ώστε ο χρήστης να συνδέεται πριν την εκτέλεση των επιμέρους ενεργειών. Έμφαση δίνεται στη λειτουργικότητα των φίλτρων και στην ορθή εφαρμογή τους στην προβολή των ανακοινώσεων.



Εικόνα 23: Μενού Φίλτρων

Το πρώτο σενάριο: FilterByText()

- Ο χρήστης συνδέεται στο σύστημα.
- Πατάει το κουμπί «Φίλτρα».
- Στο πεδίο «Αναζήτηση βάσει κειμένου» πληκτρολογεί τον όρο «δηλώσεις μαθημάτων».
- Κλείνει το μενού φίλτρων

- Εκτελείται η μέθοδος VerifyAllTitlesContain, η οποία ελέγχει ότι όλοι οι τίτλοι των ανακοινώσεων περιέχουν το ζητούμενο κείμενο.
- Ο χρήστης μεταβαίνει στην επόμενη σελίδα πατώντας το κατάλληλο κουμπί
- Η ίδια μέθοδος καλείται ξανά, για να διασφαλιστεί ότι το φίλτρο παραμένει ενεργό και μετά την αλλαγή σελίδας.

```
5   public class AnnouncementsTests : BaseTest
6   {
7       /// <summary>
8       /// Opens Filters, inserts text and verifies that all announcement Titles
9       /// contain the input text, clicks next page and verifies that the filter remains
10      /// </summary>
11      [Test]
12      public void FilterByText()
13      {
14          LoginUser();
15
16          // apply filter
17          var announcementsPage = new AnnouncementsPage(_driver);
18          announcementsPage.ClickFiltersButton();
19          announcementsPage.SearchByTitle("δηλώσεις μαθημάτων");
20          announcementsPage.CloseFilters();
21
22          // verify results contain filter
23          var announcementsValidator = new AnnouncementsPageValidator(_driver);
24          announcementsValidator.VefiryAllTitlesContain("δηλώσεις μαθημάτων");
25
26          // go to next page and verify filter remains
27          announcementsPage.ClickNextPage();
28          announcementsValidator.VerifyActivePageHasTheNumber("2");
29          announcementsValidator.VefiryAllTitlesContain("δηλώσεις μαθημάτων");
30      }
31
```

Εικόνα 24: Σενάριο FilterByText

Η προσθήκη του τελευταίου βήματος με την πλοήγηση στη δεύτερη σελίδα είναι ιδιαίτερα σημαντική, καθώς επιβεβαιώνει τη διατήρηση φίλτρων κατά την εναλλαγή σελίδων και καθόλη την εμπειρία πλοήγησης του χρήστη.

Δεύτερο σενάριο: ChangePageSize()

Το σενάριο αυτό ελέγχει τη δυνατότητα αλλαγής του αριθμού ανακοινώσεων που εμφανίζονται ανά σελίδα. Αρχικά εφαρμόζεται μια τιμή (20), και στη συνέχεια μία δεύτερη (50), ώστε να διασφαλιστεί ότι το σύστημα ανταποκρίνεται σωστά σε διαδοχικές αλλαγές.

Τρίτο σενάριο: SortAnnouncements()

Εδώ ελέγχεται η ορθή εφαρμογή της ταξινόμησης. Ο χρήστης επιλέγει διαφορετικά κριτήρια (π.χ. ημερομηνία, σημασία) και το σύστημα πρέπει να εμφανίζει τις ανακοινώσεις στη σωστή σειρά. Κατά την εκτέλεση του σεναρίου εντοπίστηκε σφάλμα στον κώδικα, αν και στις περισσότερες περιπτώσεις οι ανακοινώσεις ταξινομούνται σωστά, δύο εγγραφές εμφανίστηκαν σε λανθασμένη σειρά όταν επιλέγεται η ταξινόμηση με βάση την ημερομηνία.

Τέταρτο σενάριο: SelectSpecificDates()

Το σενάριο αφορά τον ορισμό ενός εύρους ημερομηνιών και την επαλήθευση ότι οι ανακοινώσεις που εμφανίζονται ανήκουν αποκλειστικά σε αυτό το διάστημα. Ο έλεγχος βασίστηκε σε parsing των ημερομηνιών σε μορφή ISO (yyyy-MM-dd), ώστε να αποφευχθούν ασάφειες λόγω διαφορετικών τοπικών ρυθμίσεων. Την πρώτη φορά συμπληρώνεται μόνο το φίλτρο «Ημερομηνία από», ενώ στη συνέχεια ορίζεται ημερομηνία μεταξύ ορίων «Ημερομηνία από» και «Ημερομηνία έως».

4.3.5 Αποτελέσματα και παρατηρήσεις

Αξίζει να σημειωθεί ότι κάθε σενάριο υλοποιήθηκε ως ξεχωριστό και μικρό test case, χωρίς να συνδυάζονται πολλά βήματα σε ένα ενιαίο τεστ. Αυτή η μεθοδολογία ακολουθεί την αρχή της μοναδικής ευθύνης (Single Responsibility Principle), σύμφωνα με την οποία κάθε test επικεντρώνεται σε ένα συγκεκριμένο κομμάτι λειτουργικότητας. Με τον τρόπο αυτό:

- Τα tests παραμένουν μικρά, καθαρά και ευανάγνωστα.
- Όταν αποτυγχάνει ένα σενάριο, είναι σαφές ποια ακριβώς λειτουργία έχει πρόβλημα.
- Αποφεύγεται η δημιουργία «εύθραυστων» σεναρίων που αποτυγχάνουν λόγω άσχετων ή πολλών βημάτων.
- Διευκολύνεται η συντήρηση, αφού κάθε αλλαγή στο UI επηρεάζει τοπικά μόνο το αντίστοιχο test.

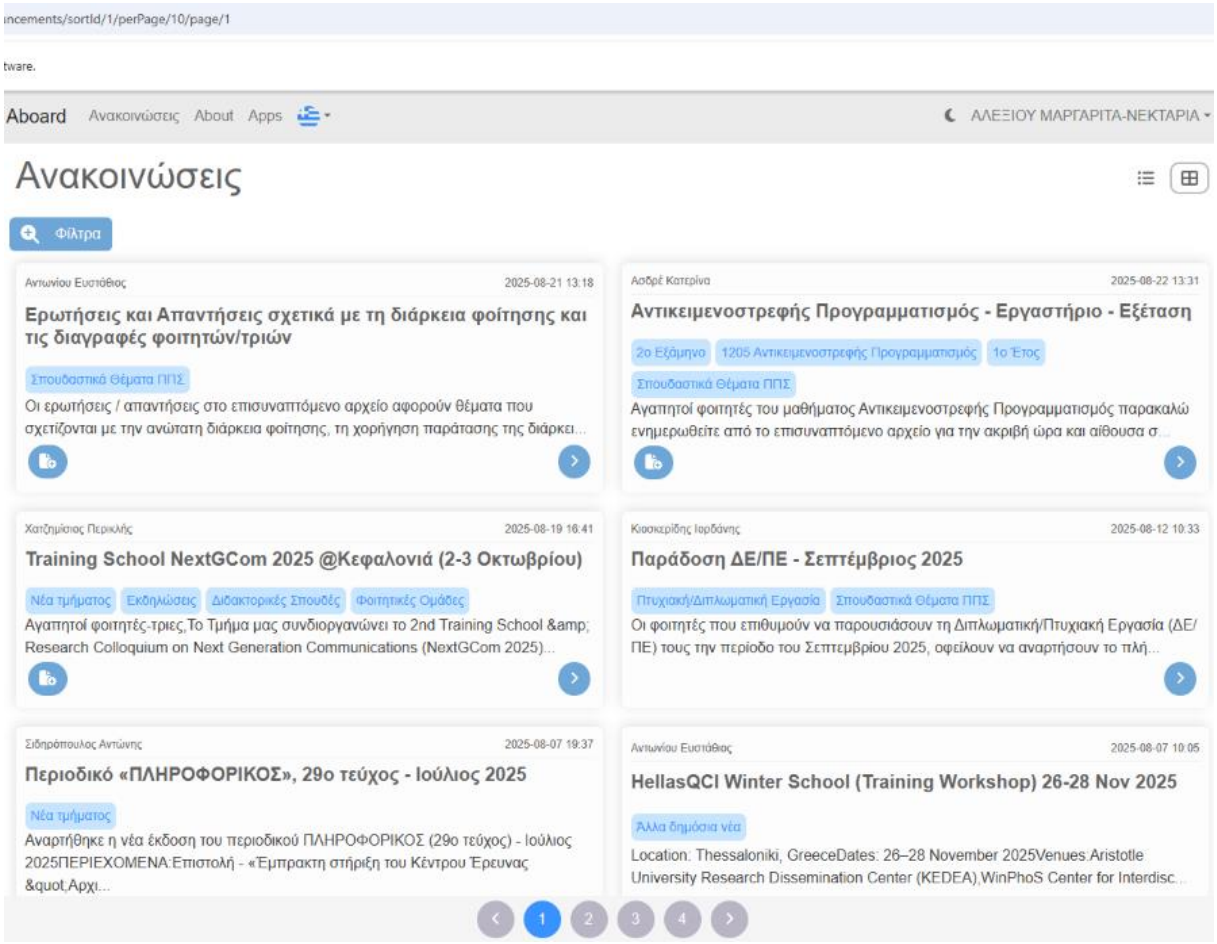
Αυτή η πρακτική αντανακλά μια ευρέως υιοθετημένη στρατηγική στο automation testing: προτιμάται ένας μεγαλύτερος αριθμός από απλά, εστιασμένα tests αντί για λίγα και περίπλοκα σενάρια.

Η αρχή του Single Responsibility Principle (SRP), λειτουργεί συμπληρωματικά προς το regression testing. Μία αλλαγή στον κώδικα, σε ένα συγκεκριμένο σημείο λειτουργικότητας, απαιτεί την εκτέλεση των σεναρίων που επικεντρώνονται σε αυτό, αντί της εκτέλεσης ολόκληρης της σουίτας. Η εστίαση αυτή, μειώνει την ακτίνα επίδρασης, περιορίζει το υπολογιστικό κόστος και τον χρόνο εκτέλεσης στους κύκλους CI/CD, επιτρέποντας ταχύτερη και πιο ασφαλή, την προώθηση των αλλαγών στην παραγωγή. Η καθαρή διάκριση ευθυνών του SRP κάνει τα σενάρια πιο προβλέψιμα και εύκολα στη συντήρηση.

[31]

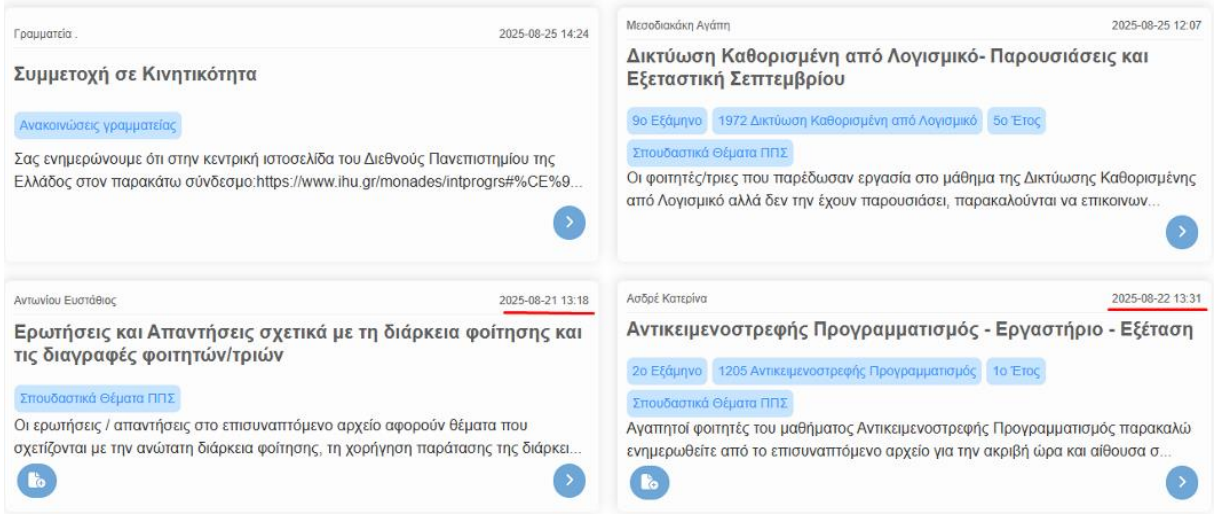
4.3.5.1 Εντοπισμός Σφάλματος

Όπως αναφέρθηκε ήδη, στο τρίτο σενάριο που ελέγχεται η ταξινόμηση, βρέθηκε σφάλμα. Η ανίχνευσή του ήταν άμεση, καθώς το τεστ είναι απομονωμένο και επικεντρωμένο αποκλειστικά στη λειτουργία ταξινόμησης.



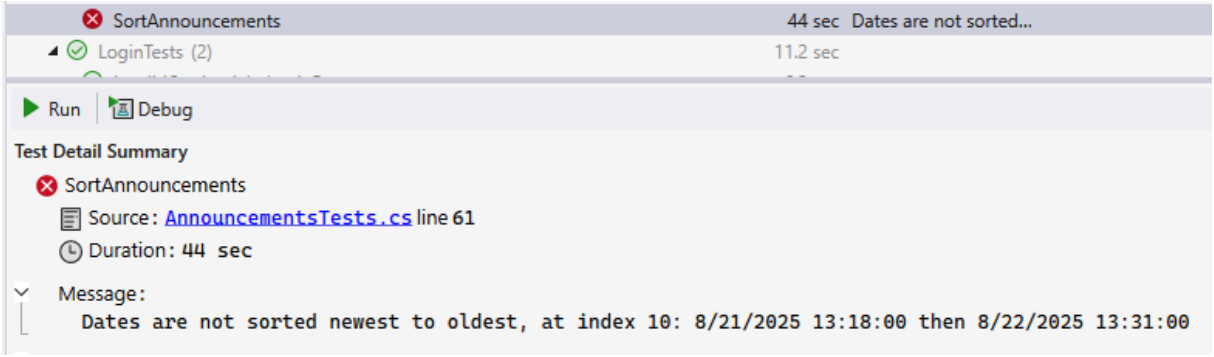
Εικόνα 25: Σφάλμα σε ταξινόμηση 1

Στην εικόνα 6.12, φαίνεται ένα στιγμιότυπο από το τεστ όταν απέτυχε, στη διεύθυνση σελίδας φαίνεται ότι έχει γίνει εφαρμογή της ταξινόμησης. Το σφάλμα παρουσιάζεται στις 2 πρώτες δημοσιεύσεις, με ημερομηνία 2025-08-21, να προηγείται της 2025-08-22, ενώ οι επόμενες φαίνεται να έχουν σωστή ταξινόμηση.



Εικόνα 26: Σφάλμα σε ταξινόμηση 2

Στην εικόνα 6.13, οι πρώτες ακολουθούν σωστά την επιλεγμένη ταξινόμηση, και το πρόβλημα παρατηρείται στις επόμενες (9, 10), με ημερομηνίες ξανά την 2025-08-21, να προηγείται της 2025-08-22. Με περαιτέρω διερεύνηση, σε μεγαλύτερο δείγμα ανακοινώσεων, προς το παρόν δεν φαίνεται να εμφανίζεται σε άλλες δημοσιεύσεις και ημερομηνίες. Πιθανό σενάριο είναι πώς μία από τις ανακοινώσεις έχει υποστεί επεξεργασία, επομένως συμπεραίνεται ότι ο μηχανισμός ταξινόμησης στο backend εφαρμόζεται βάσει του updated_at (τελευταία τροποποίηση), αντί του created_at (ημερομηνία δημιουργίας). Η ασυμφωνία αυτή εξηγεί γιατί οι δύο συγκεκριμένες εγγραφές αντιστρέφονται, ενώ οι υπόλοιπες, που δεν έχουν τροποποιηθεί, παραμένουν σωστά ταξινομημένες. Μέχρι την τελική διόρθωση, η συγκεκριμένη γραμμή στον κώδικα που επαληθεύει ότι η ταξινόμηση γίνεται σωστά έχει μπει σαν σχόλιο, προκειμένου το σενάριο να μην σταματάει και να συνεχίσει στον επόμενο έλεγχο που είναι η ταξινόμηση βάση σημαντικότητας.



Εικόνα 27: Λεπτομέρειες αποτυχίας του σεναρίου

```
51
52  public void VerifySortingMostRecent()
53  {
54      _wait.Until(e => e.FindElements(_by.AnnouncementItems).Count > 0);
55
56      ReadOnlyCollection<IWebElement> dateElements = _driver.FindElements(_by.AnnouncementDate);
57      string format = "yyyy-MM-dd HH:mm";
58
59      List<DateTime> dates = dateElements.Select(e => e.Text?.Trim())
60                              .Where(s => !string.IsNullOrWhiteSpace(s))
61                              .Select(s => DateTime.ParseExact(s, format, CultureInfo.InvariantCulture))
62                              .ToList();
63
64      for (int i = 1; i < dates.Count; i++)
65      {
66          if (dates[i] > dates[i - 1])
67              Assert.Fail($"Dates are not sorted newest to oldest, at index {i}: {dates[i - 1]} then {dates[i]}");
68      }
69  }
70
```

Εικόνα 28: Μέθοδος ελέγχου ταξινόμησης

Στην εικόνα 6.15, παρουσιάζεται η υλοποίηση της μεθόδου VerifySortingMostRecent(). Η μέθοδος αρχικά περιμένει (με explicit wait) έως ότου εμφανιστούν τα στοιχεία που περιέχουν τις ημερομηνίες των ανακοινώσεων. Στη συνέχεια, συλλέγονται όλα τα στοιχεία ημερομηνίας από το DOM και μετατρέπονται σε αντικείμενα τύπου DateTime, χρησιμοποιώντας προκαθορισμένο format (yyyy-MM-dd HH:mm). Η λίστα που προκύπτει περιλαμβάνει όλες τις ημερομηνίες με τη σωστή χρονική ακρίβεια. Έπειτα, πραγματοποιείται έλεγχος με βρόχο επανάληψης, κάθε ημερομηνία συγκρίνεται με την αμέσως προηγούμενή της για να διαπιστωθεί αν είναι μικρότερη ή ίση, εξασφαλίζοντας ότι η σειρά είναι από τη νεότερη προς την παλαιότερη. Σε περίπτωση που βρεθεί ημερομηνία τοποθετημένη λανθασμένα, το τεστ αποτυγχάνει και εμφανίζει κατάλληλο μήνυμα σφάλματος, στο οποίο καταγράφεται η ακριβής θέση του προβλήματος καθώς και οι δύο ημερομηνίες που παραβιάζουν τη σωστή σειρά.

4.3.5.2 Αντιμετώπιση δυσκολιών

Κατά την ανάπτυξη των αυτοματοποιημένων δοκιμών, ένα από τα κύρια ζητήματα που προέκυψαν σχετιζόταν με τον εντοπισμό στοιχείων (locators) στη σελίδα. Σε αρκετές περιπτώσεις η διαδικασία ήταν απλή, καθώς τα στοιχεία διέθεταν μοναδικά Id ή χαρακτηριστικά κλάσεων (class names), γεγονός που επέτρεπε τη χρήση απλών CSS selectors (εικόνα 6.16). Ωστόσο, υπήρξαν και περιπτώσεις όπου ο εντοπισμός ήταν πιο σύνθετος, απαιτώντας XPath εκφράσεις ή συνδυασμούς selectors για να προσδιοριστεί με ακρίβεια το επιθυμητό στοιχείο (εικόνα 6.17).

```
// Filters
public By FilterButton => By.CssSelector("div.filters-actions > button");
public By SearchByTitleBox => By.Id("search-title");
public By SearchByTextBox => By.Id("search-body");
public By SelectTagsTree => By.ClassName("tree-controller-container");
public By CloseXButton => By.CssSelector("div.search-header > svg");
```

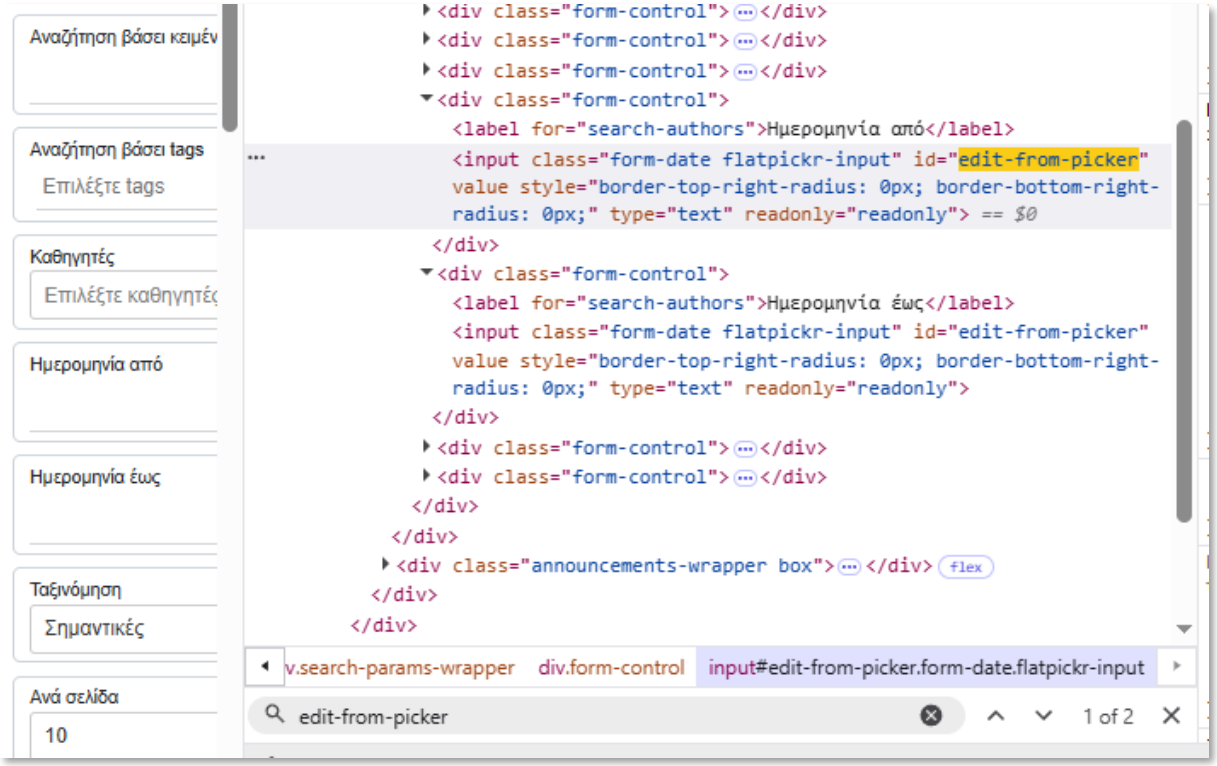
Εικόνα 29: Στοιχεία με εύκολο εντοπισμό

```
// Sorting
public By SortingDropdown => By.XPath("//label[contains(.,'Ταξινόμηση')]/following::div[contains(@class,'dropdown-indicator')]");
public By SortingMenu => By.CssSelector(".my-react-select__menu");
public By SortingOption(string n) =>
    By.XPath($"//div[contains(@class,'my-react-select__option') and normalize-space()='{n}']");

// Per page
public By PerPageDropdown => By.XPath("//label[contains(.,'Ανά σελίδα')]/following::div[contains(@class,'dropdown-indicator')]");
public By PerPageMenu => By.CssSelector(".my-react-select__menu");
public By PerPageOption(string n) =>
    By.XPath($"//div[contains(@class,'my-react-select__option') and normalize-space()='{n}']");
```

Εικόνα 30: Στοιχεία με περίπλοκο εντοπισμό

Ένα χαρακτηριστικό παράδειγμα προέκυψε στα πεδία ημερομηνιών του φίλτρου. Στον κώδικα της σελίδας βρέθηκαν δύο στοιχεία με το ίδιο Id (edit-from-picker, εικόνα 6.16), κάτι που δεν αποτελεί καλή πρακτική, καθώς το Id πρέπει να είναι μοναδικό σε κάθε HTML έγγραφο. Η ύπαρξη διπλού Id οδήγησε σε ασάφεια κατά τον εντοπισμό του στοιχείου και, στην πράξη, σε σφάλματα κατά την εκτέλεση των tests. Για την αντιμετώπιση του προβλήματος επιλέχθηκε η χρήση XPath που βασίζεται στο κείμενο της ετικέτας (label) και εντοπίζει το αντίστοιχο input πεδίο, παρακάμπτοντας την εσφαλμένη επαναχρησιμοποίηση του Id.

Η εμπειρία αυτή ανέδειξε δύο σημαντικές παρατηρήσεις: αφενός, ότι η ποιότητα του frontend κώδικα (π.χ. σωστή χρήση μοναδικών Id) επηρεάζει άμεσα την ευκολία και την αξιοπιστία του automated testing, αφετέρου, ότι η ύπαρξη ευέλικτων locators μέσω XPath ή σύνθετων CSS selectors αποτελεί απαραίτητο εργαλείο για την επιτυχή κάλυψη σεναρίων σε περιβάλλοντα όπου δεν τηρούνται πάντα οι βέλτιστες πρακτικές ανάπτυξης.



Εικόνα 31: Διπλή χρήση id σε στοιχείο της σελίδας

Ένα ακόμη ζήτημα που παρουσιάστηκε αφορούσε την αναζήτηση και σύγκριση κειμένου στην ελληνική γλώσσα. Οι τίτλοι των ανακοινώσεων μπορεί να εμφανίζονται άλλοτε με τόνους και άλλοτε χωρίς, καθώς και με διαφορετική χρήση πεζών ή κεφαλαίων γραμμάτων. Αυτό δημιουργεί δυσκολίες σε μια απλή σύγκριση string, καθώς η μέθοδος Contains() σε .NET δεν λαμβάνει υπόψη τέτοιες γλωσσικές παραλλαγές.

Για την αντιμετώπιση του προβλήματος αξιοποιήθηκε η κλάση CultureInfo με ρυθμίσεις για τα ελληνικά (el-GR), σε συνδυασμό με το CompareOptions.IgnoreCase (παράβλεψη πεζών/κεφαλαίων) και το CompareOptions.IgnoreNonSpace (παράβλεψη τόνων). Με αυτόν τον τρόπο, η μέθοδος VerifyAllTitlesContain ελέγχει με αξιοπιστία αν οι τίτλοι περιέχουν το αναμενόμενο κείμενο, ανεξάρτητα από διαφορές στην ορθογραφική απεικόνιση.

Η λύση αυτή αναδεικνύει μια γενικότερη δυσκολία στον αυτοματοποιημένο έλεγχο, καθώς η αξιοπιστία τους εξαρτάται από την ακριβή αντιστοίχιση δεδομένων, η οποία σε γλώσσες με διακριτικά (όπως τα ελληνικά) απαιτεί προσαρμογές που δεν είναι αναγκαίες σε γλώσσες χωρίς τόνους (π.χ. αγγλικά).

```
public void VefiryAllTitlesContain(string expectedTitle)
{
    // helpers for comparison in Greek Language
    var compare = CultureInfo.GetCultureInfo("el-GR").CompareInfo;
    var options = CompareOptions.IgnoreCase | CompareOptions.IgnoreNonSpace;

    _wait.IgnoreExceptionTypes(typeof(NoSuchElementException), typeof(StaleElementReferenceException));

    _wait.Until(e => e.FindElements(_by.AnnouncementItems).Count > 0);

    IReadOnlyCollection<IWebElement> headers = _wait.Until(driver =>
    {
        var elements = driver.FindElements(_by.AnnouncementTitleHeaders);
        if (elements.Count == 0) return null;

        // Ensure each header has non-empty text
        return elements.All(e => !string.IsNullOrWhiteSpace(e.Text)) ? elements : null;
    });

    foreach (var item in headers)
    {
        var title = item.Text?.Trim() ?? string.Empty;

        if (compare.IndexOf(title, expectedTitle, options) < 0)
        {
            Assert.Fail($"Title: {title}, does not contain '{expectedTitle}'");
        }

    }
}
```

Εικόνα 32: Έλεγχος κειμένου Τίτλων

4.3.6 Συμπεράσματα

Ο έλεγχος της σελίδας Aboard με Selenium ανέδειξε τη σημασία του ελέγχου διεπαφής χρήστη (frontend) συμπληρωματικά, στον συνολικό ποιοτικό έλεγχο. Ενώ το API testing επικεντρώνεται στη συνέπεια των δεδομένων, τα σενάρια UI κατέδειξαν πώς βιώνουν οι χρήστες την εφαρμογή και αν οι βασικές λειτουργίες (login, φιλτράρισμα, ταξινόμηση, πλοήγηση) εκτελούνται όπως αναμένεται.

Η επιλογή της αρχιτεκτονικής Page Object Model (POM) προσέφερε σαφή διαχωρισμό ευθυνών μεταξύ locators, ενεργειών και ελέγχων, κάνοντας τον κώδικα σημαντικά πιο εύκολο στη συντήρηση. Η χρήση explicit waits ενίσχυσε τη σταθερότητα των tests, αποτρέποντας αποτυχίες που οφείλονται σε ασύγχρονες εμφανίσεις στοιχείων. Παράλληλα, η παραμετροποίηση μέσω appsettings.json και η αξιοποίηση του ConfigReader επέτρεψαν την αποφυγή «μαγικών τιμών» στον κώδικα και έκαναν τη σουίτα εύκολα προσαρμόσιμη σε διαφορετικά περιβάλλοντα.

Κατά την ανάπτυξη των σεναρίων εντοπίστηκαν πρακτικές δυσκολίες που συχνά συναντώνται σε πραγματικές εφαρμογές, όπως η ύπαρξη διπλών Id στον HTML κώδικα ή η ανάγκη σύγκρισης κειμένου στα ελληνικά. Η ύπαρξη μικρών και εστιασμένων tests διευκόλυνε τον εντοπισμό συγκεκριμένων σφαλμάτων, όπως το bug που προκαλεί λανθασμένη ταξινόμηση σε ορισμένες ανακοινώσεις, αποδεικνύοντας στην πράξη την αξία της μεθοδολογίας αυτής.

Συνολικά, η υλοποίηση απέδειξε ότι ο έλεγχος του UI με Selenium, όταν συνδυάζεται με καλή αρχιτεκτονική και σαφείς πρακτικές συγχρονισμού, παρέχει αξιόπιστη κάλυψη των κρίσιμων σεναρίων χρήσης. Παράλληλα, εμφανίστηκαν προκλήσεις που σχετίζονται με την εξάρτηση από τη δομή του frontend και αναδείχτηκε η σημασία της συνεργασίας μεταξύ ομάδων ανάπτυξης και του testing, για τη διασφάλιση ότι οι εφαρμογές είναι εύκολο να υποβληθούν σε διαδικασία ελέγχου με αυτόματο τρόπο.

Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης

5.1 Εισαγωγή

Η εργασία αυτή, συνδύασε θεωρητική ανάλυση και πρακτική υλοποίηση με σκοπό να αποτυπώσει τον ρόλο και την αξία των σύγχρονων τεχνικών automation testing. Στα πρώτα κεφάλαια έγινε αναφορά στα θεμέλια του ελέγχου και στην ποιότητας λογισμικού, στις κατηγορίες και στις προσεγγίσεις testing, καθώς και στα μοντέλα και τις μεθοδολογίες ανάπτυξης (Agile, TDD, BDD, CI/CD, DevSecOps). Στη συνέχεια, μέσα από τη μελέτη περίπτωσης της πλατφόρμας Aboard, υλοποιήθηκε μια ολοκληρωμένη σουίτα ελέγχων τόσο σε επίπεδο API όσο και σε επίπεδο UI με Selenium, επιτρέποντας τη συσχέτιση της θεωρίας με την πρακτική εφαρμογή τους.

Η βιβλιογραφική διερεύνηση ανέδειξε τρεις κομβικές θέσεις. Πρώτον, ότι η ποιότητα δεν είναι τελικό στάδιο αλλά διαρκής δραστηριότητα που διαπερνά όλο τον κύκλο ζωής του λογισμικού, επομένως, πρακτικές όπως το shift-left και το continuous testing δεν είναι «προαιρετικές» αλλά προϋπόθεση για προβλέψιμα αποτελέσματα. Δεύτερον, τόσο ο διαχωρισμός σε λειτουργικές, όσο και σε μη λειτουργικές δοκιμές παραμένει χρήσιμος μόνο όταν συνδέεται με σαφή κριτήρια αποδοχής και μετρικές (π.χ. αξιοπιστία, ασφάλεια, απόδοση). Τρίτον, πως οι σύγχρονες μεθοδολογίες ανάπτυξης (Agile/DevOps) ευνοούν την αυτοματοποίηση, όσο μικρότερα τα βήματα ολοκλήρωσης, τόσο περισσότερο αξίζει μία υποδομή γρήγορων, επαναλήψιμων και αξιόπιστων tests που εκτελούνται συνεχώς και επιστρέφουν έγκαιρη ανατροφοδότηση.

5.2 Συμπεράσματα από τη μελέτη περίπτωσης (API και UI)

Η μελέτη περίπτωσης στο Aboard επέτρεψε να φανεί στην πράξη το κέρδος από τον συνδυασμό API και UI testing. Σε επίπεδο API, η προσέγγιση BDD με Reqnroll και η οργάνωση με NUnit παρήγαγαν ευανάγνωστα σενάρια και επαναχρησιμοποιήσιμα βήματα. Η χρήση RestSharp και Newtonsoft.Json διευκόλυνε τόσο στοχευμένους ελέγχους πεδίων όσο και συγκρίσεις πλήρους JSON. Παρά τον περιορισμό ότι το authentication token δεν εκδίδεται από δημόσιο endpoint αλλά μέσω frontend (sessionStorage), η σουίτα κάλυψε επαρκώς δημόσια και προστατευμένα endpoints, ανέδειξε αποκλίσεις όπου υπήρχαν και τεκμηρίωσε συμπεριφορές με σαφή assertions. Η κατηγοριοποίηση με tags τύπου public, token, smoke, επέτρεψε στοχευμένες εκτελέσεις.

Σημαντική διαπίστωση είναι ότι η κάλυψη μόνο με ρόλο φοιτητή αφήνει κενά σε λειτουργίες που διαθέτουν περισσότερα δικαιώματα (π.χ. ανακοινώσεις από γραμματεία ή από καθηγητές). Η υλοποίηση κεντρικών σεναρίων βασισμένα στους ρόλους (role-based tests) με λογαριασμούς γραμματείας και καθηγητή, θα επιτρέψει τον έλεγχο ολόκληρης της υλοποίησης δικαιωμάτων (create/read/update/delete) και κατά συνέπεια, την αξιολόγηση endpoints που σήμερα δεν είναι προσβάσιμα. Ειδικά τα σενάρια POST/PUT/DELETE, τα οποία προς το παρόν δεν εκτελούνται λόγω περιορισμένων δικαιωμάτων, είναι καίρια για να ελέγξουν κανόνες επικύρωσης, εξαρτήσεις, και τους σωστούς κωδικούς σφαλμάτων. Η προσθήκη τέτοιων σεναρίων χρειάζεται να συνοδευτεί από ασφαλή διαχείριση test data, σε ειδικό περιβάλλον για testing, ώστε οι δυναμικές εγγραφές να μην επηρεάζουν την παραγωγή ή άλλα tests.

Σε επίπεδο UI, η αρχιτεκτονική Page Object Model (ElementMap / Page / Validator), ο κεντρικός χειρισμός του κύκλου ζωής του WebDriver (BaseTest) και τα explicit waits (WebDriverWait) αύξησαν τη σταθερότητα. Η συνειδητή επιλογή χρήσης By locators αντί αποθηκευμένων IWebElement αναφορών προστάτευσε τα tests από StaleElementReferenceException, ενώ η παραμετροποίηση μέσω

appsettings.json και ConfigReader έκανε τη σουίτα φορητή ανάμεσα σε μηχανές και περιβάλλοντα. Τα σενάρια, στη σελίδα ανακοινώσεων, σχεδιάστηκαν μικρά και εστιασμένα (single-responsibility), γεγονός που επιτρέπει τη γρήγορη διάγνωση προβλημάτων. Έτσι εντοπίστηκε bug στην ταξινόμηση (δύο ημερομηνίες σε λανθασμένη σειρά, λόγω update), ενώ κατά τον σχεδιασμό των locators παρατηρήθηκε στην HTML η χρήση διπλού id, το οποίο αντιμετωπίστηκε με τη χρήση XPath. Τέλος, οι συγκρίσεις κειμένου προσαρμόστηκαν στις ιδιαιτερότητες της ελληνικής γλώσσας (πεζά, κεφαλαία και τονισμός), αυξάνοντας την αξιοπιστία των επαληθεύσεων.

Συνολικά, ο συνδυασμός ελέγχου σε API και UI, παρέχει σφαιρική εικόνα ποιότητας: τα API tests επιβεβαιώνουν επιχειρησιακή λογική και συνέπεια δεδομένων, ενώ τα UI tests διασφαλίζουν ότι η εμπειρία του χρήστη παραμένει λειτουργική, προβλέψιμη και σταθερή. Όταν τρέχουν παράλληλα και τροφοδοτούνται με κοινά δεδομένα/αναμενόμενα αποτελέσματα, μειώνεται δραστικά ο χρόνος διάγνωσης σφαλμάτων.

5.3 Προτάσεις βελτίωσης και μελλοντικής επέκτασης

Με βάση τα αποτελέσματα και συμπεράσματα, προτείνονται οι ακόλουθες κατευθύνσεις, δίνοντας βαρύτητα στις ανάγκες που αναδείχθηκαν:

Εμβάθυνση κάλυψης API με ρόλους και έλεγχος των υπολοίπων endpoints

Η δημιουργία σεναρίων με ρόλους γραμματείας και καθηγητή θα επιτρέψει τον έλεγχο πολιτικών πρόσβασης και επιχειρησιακών ροών που σήμερα δεν έχουν ελεγθεί. Παράλληλα, η προσθήκη σεναρίων POST/PUT/DELETE είναι κρίσιμη για να ελεγχθούν επικυρώσεις και μηνύματα σφαλμάτων. Για να είναι ασφαλής η εκτέλεση, απαιτείται ο έλεγχος να πραγματοποιηθεί σε ελεγχόμενο περιβάλλον για testing και όχι στο παραγωγικό περιβάλλον.

Ενίσχυση του Selenium για CI/CD

Η εκτέλεση των τεστ με τη λειτουργία headless, δηλαδή χωρίς τη χρήση του browser να είναι ορατή, μειώνει το χρόνο και τους πόρους. Είναι ιδανική για pipelines και επιτρέπει ευκολότερο παράλληλο τρέξιμο. Ακόμη, η προσθήκη λήψης screenshot στο σημείο αποτυχίας, μειώνει το χρόνο που θα χρειαστεί για να βρεθεί η αιτία, κάνοντας πιο εύκολο για τον tester να καταλάβει αν πρόκειται για πραγματικό σφάλμα, ή αν πρόκειται για αστοχία. Έχοντας την εικόνα στο σημείο που το τεστ σταμάτησε/απέτυχε, γίνεται πιο εύκολη η διαδικασία της αναπαραγωγής για περαιτέρω διερεύνηση.

5.4 Επίλογος

Η εμπειρία από την παρούσα εργασία επιβεβαίωσε ότι το automation testing είναι κάτι περισσότερο από εργαλεία και script για γρήγορη εκτέλεση εντολών. Είναι ένας τρόπος σκέψης που ευνοεί την απλότητα, τον καθαρό διαχωρισμό ευθυνών και τη γρήγορη ανατροφοδότηση. Η σχεδίαση μικρών, εστιασμένων test, η επένδυση σε αρχιτεκτονική (POM, κοινές βιβλιοθήκες, παραμετροποίηση) και η καλή συνεργασία με την ανάπτυξη, αποδίδουν απτά αποτελέσματα. Συγκεκριμένα, γρηγορότερη ανίχνευση σφαλμάτων, ταχύτερες διορθώσεις και μεγαλύτερη εμπιστοσύνη στις εκδόσεις.

Σε προσωπικό επίπεδο, η μελέτη περίπτωσης αποτέλεσε ευκαιρία να συνδεθούν οι θεωρητικές αρχές με τις πραγματικές αποφάσεις σχεδιασμού και τα καθημερινά εμπόδια ενός μηχανικού δοκιμών. Το σημαντικότερο συμπέρασμα είναι ότι η ποιότητα χτίζεται σταδιακά, με μικρά, σταθερά βήματα, μετρήσιμες πρακτικές και συνεχή βελτίωση. Αυτό η νοοτροπία σε συνδυασμό με τα τεχνικά θεμέλια που αναπτύχθηκαν, αποτελεί τη βάση για τα επόμενα βήματα επέκτασης και ωρίμανσης της σουίτας ελέγχων.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] G. J. Myers, C. Badgett, and C. Sandler, The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 2011.

[2] IEEE Standard for Software and System Test Documentation, IEEE Std. 829-2008, 2008.

[3] W. S. Humphrey, Managing the Software Process. Reading, MA, USA: Addison-Wesley, 1989. [Online]. Available: https://www.scribd.com/document/471352752/282899986-pdf. [Accessed: 10-Aug-2025].

[4] M. Fowler, "Continuous Integration," ThoughtWorks, May 1, 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Accessed: 10-Aug-2025].

[5] ISO/IEC 25010:2011, Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, ISO/IEC, 2011.

[6] IBM, "What is Shift-Left Testing?," IBM Think, 2024. [Online]. Available: https://www.ibm.com/think/topics/shift-left-testing. [Accessed: 10-Aug-2025].

[7] B. Beizer, Software Testing Techniques, 2nd ed. New York, NY, USA: Van Nostrand Reinhold, 1990.

[8] C. Kaner, J. Bach, and B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach. Hoboken, NJ, USA: Wiley, 2002. [Online]. Available: https://www.scribd.com/doc/127131803/Wiley-Lessons-Learned-in-Software-Testing. [Accessed: 10-Aug-2025].

[9] ZAPTEST, "What is Sanity Testing? Deep dive into types, process, approaches, tools & more," ZAPTEST, 2025. [Online]. Available: https://www.zaptest.com/what-is-sanity-testing-deep-dive-into-types-process-approaches-tools-more. [Accessed: 10-Aug-2025].

[10] International Software Testing Qualifications Board (ISTQB), Certified Tester Foundation Level Syllabus, 4.0 ed., 2021.

[11] Testlio, "Test Automation Challenges: Common Issues and How to Overcome Them," Testlio, 2025. [Online]. Available: https://testlio.com/blog/test-automation-challenges/. [Accessed: 10-Aug-2025].

[12] Software Testing Magazine, "Comparison of Selenium, Cypress, and Playwright as Automation Tools," Software Testing Magazine, Jun. 2025.

[13] J. Langr, A. Hunt, and D. Thomas, Modern Unit Testing with Java and Python. Raleigh, NC, USA: Pragmatic Bookshelf, 2023.

[14] JustAcademy, "How to do Performance Test using Appium for Mobile App," JustAcademy.co, May 2025.

[15] BrowserStack, "Top Performance Testing Tools in 2025: JMeter Overview," BrowserStack Guide, Apr. 2025. [Online]. Available: https://www.browserstack.com/guide/performance-testing-tools. [Accessed: 10-Aug-2025].

[16] GeeksforGeeks, “Waterfall Model – Software Engineering,” GeeksforGeeks, Jul. 11, 2025. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/waterfall-model/. [Accessed: 10-Aug-2025].

[17] Wikipedia contributors, “Agile software development,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Agile_software_development. [Accessed: 10-Aug-2025].

[18] R. S. Pressman, Software Engineering: A Practitioner’s Approach, 8th ed. New York, NY, USA: McGraw-Hill, 2014.

[19] Wikipedia contributors, “Agile software development,” Wikipedia, The Free Encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Agile_software_development. [Accessed: 10-Aug-2025].

[20] PMColumn, “Agile vs Waterfall: A Side-by-Side Comparison,” PMColumn, Feb. 11, 2023. [Online]. Available: https://www.pmcolumn.com/agile-vs-waterfall/. [Accessed: 10-Aug-2025].

[21] K. Beck, Test-Driven Development: By Example. Boston, MA, USA: Addison-Wesley, 2003. [Online]. Available: https://archive.org/details/est-driven-development-by-example/test-driven-development-by-example/page/n23/mode/2up. [Accessed: 10-Aug-2025].

[22] N. Nagappan, T. Bhat, and E. M. Maximilien, “Realizing Quality Improvement Through Test-Driven Development: Results and Experiences of Four Industrial Teams,” Empirical Software Engineering, vol. 13, no. 3, pp. 289-302, 2008.

[23] Software Testing Magazine, “The Role of AI & Machine Learning in Modern Software Testing,” Software Testing Magazine, Jun. 21, 2024. [Online]. Available: https://www.softwaretestingmagazine.com/knowledge/the-role-of-ai-and-machine-learning-in-modern-software-testing/. [Accessed: 10-Aug-2025].

[24] RTSLabs, “Top 10 Benefits of Partnering with DevOps Consulting Firms,” RTSLabs, Feb. 19, 2025. [Online]. Available: https://rtslabs.com/partnering-with-devops-consulting-firms. [Accessed: 10-Aug-2025].

[25] S. Spiro, “120 DevOps statistics: Principles, success, & the future endeavors,” Hutte.io, Mar. 6, 2024. [Online]. Available: https://hutte.io/trails/devops-statistics/. [Accessed: 10-Aug-2025].

[26] S. W. Ambler, “Going Beyond Scrum: Disciplined Agile Delivery,” 2013. [Online]. Available: https://www.scribd.com/doc/252000509/Going-Beyond-Scrum-Scott-Ambler. [Accessed: 10-Aug-2025].

[27] M. Postman, *The Complete Guide to API Testing*. New York, NY: API Academy, 2020.

[28] G. Smart, *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Shelter Island, NY: Manning Publications, 2014.

[29] NUnit.org, “NUnit 3 Documentation,” 2023. [Online]. Available: https://docs.nunit.org/

[30] J. Newton-King, “Json.NET Documentation,” Newtonsoft, 2023. [Online]. Available: https://www.newtonsoft.com/json

[31] GeeksforGeeks, “Single Responsibility in SOLID Design Principle,” GeeksforGeeks, Jul. 23 2025. [Online]. Available: https://www.geeksforgeeks.org/system-design/single-responsibility-in-solid-design-principle/. [Accessed: 31-Aug-2025]