ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Σύστημα διευκόλυνσης του ελέγχου και της ανάλυσης των υπό διερεύνηση tests κατά τη διάρκεια του κύκλου ζωής ενός λογισμικού»

Του φοιτητή
Καρακασίδη Δημήτριου
Αρ. Μητρώου: 164672

Επιβλέπων
Τζέκης Παναγιώτης
Καθηγητής

Ιανουάριος 2025

Τίτλος Π.Ε. Σύστημα διευκόλυνσης του ελέγχου και της ανάλυσης των υπό διερεύνηση tests κατά τη διάρκεια του κύκλου ζωής ενός λογισμικού

Κωδικός Δ.Ε. 24164

Ονοματεπώνυμο φοιτητή Καρακασίδης Δημήτριος

Ονοματεπώνυμο εισηγητή Τζέκης Παναγιώτης

Ημερομηνία ανάληψης Δ.Ε. 22-03-2024

Ημερομηνία περάτωσης Δ.Ε. 26-01-2025

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

Πρόλογος

Το θέμα της πτυχιακής προέκυψε μετά από χρόνια καθημερινής, επαγγελματικής χρήσης του Azure DevOps και κυρίως του εργαλείου Test Plans, για τη διασφάλιση της ποιότητας του λογισμικού. Ως test automation engineer, καλούμαι να δημιουργώ αυτοματοποιημένα tests, να τα εκτελώ και να επιβλέπω τα αποτελέσματά τους μετά από κάθε εκτέλεση σε κάθε νέα έκδοση του λογισμικού. Τα εργαλεία και οι υπηρεσίες που προσφέρονται μέσω του Azure DevOps, αν και εξαιρετικά χρήσιμα, πολλές φορές καταλήγουν να εμποδίζουν την παραγωγικότητα, καθώς συνυπάρχουν με αμέτρητες άλλες λειτουργικότητες και επιλογές, οι οποίες στην πραγματικότητα αποτελούν τροχοπέδη για το μεγαλύτερο μέρος της διάρκειας του κύκλου ζωής ανάπτυξης του λογισμικού. Σε μια προσπάθεια εξυγίανης του εργαλείου Test Plans του Azure DevOps, απομακρύνοντας όλες τις παθογένειες, διατηρώντας, όμως, παράλληλα όλα τα προτερήματα του, διαπίστωσα ότι πέρα από εφικτό, είναι στην πραγματικότητα ακόμα πιο αποτελεσματικό. Η προσπάθεια αυτή, πραγματώθηκε μέσα από την εκπόνηση της παρούσας πτυχιακής.

Περίληψη

Στην παρακάτω πτυχιακή εργασία παρουσιάζεται η υλοποίηση μιας διαδικτυακής εφαρμογής, που στοχεύει στην εξομάλυνση της διαδικασίας της ανάλυσης των αποτυχημένων test κατά τη διάρκεια του κύκλου ζωής ανάπτυξης ενός λογισμικού. Προσφέρει στους testers τη δυνατότητα να ελέγξουν τα αποτελέσματα των test pipelines του Azure DevOps, παρέχοντας γενικές πληροφορίες για την κατάσταση του κάθε pipeline, αλλά και να περιηγηθούν μέσα σε αυτά, ώστε να έχουν μια πρώτη, άμεση, εικόνα της εκτέλεσης των tests. Επιπλέον, παρουσιάζει σε μια σελίδα, το κάθε test που εκτελέστηκε είτε επιτυχώς είτε όχι, με τις μέγιστες πιθανές πληροφορίες του test, την εκτέλεσή του, αλλά και το πρόσφατο ιστορικό του. Αυτή η συγκεντρωτική παρουσίαση, διευκολύνει τον tester στην ανάλυση των tests, καθώς δεν χρειάζεται πια να απευθύνεται σε διαφορετικές σελίδες και εφαρμογές. Με αυτόν τον τρόπο, ο χρήστης αποκτά όλες τις πληροφορίες που χρειάζεται με σκοπό να διεκπεραιώσει την ανάλυσή του. Αρχικά, γίνεται αναφορά στον ρόλο και τις αρμοδιότητες του test automation engineer στον κύκλο ζωής του λογισμικού. Έπειτα, αναλύονται η αρχιτεκτονική και οι τεχνολογίες που χρησιμοποιήθηκαν, τόσο για την ανάπτυξη της εφαρμογής, όσο και για την σχεδίαση και ανάλυση της. Τέλος, παραθέτονται παραδείγματα κώδικα των βασικών σημείων της υλοποίησης και οι αντίστοιχες σελίδες της.

«System to facilitate the analysis of tests under investigation during the software life cycle»

«Dimitrios Karakasidis»

Abstract

The following thesis presents the implementation of a web application, which aims to streamline the process of analyzing the failed tests during the software development life cycle. It offers to the testers the ability to review the results of Azure DevOps test pipelines, providing general information about the status of each pipeline, but also to navigate through them, in order to have an immediate view of the execution of the tests. In addition, it presents on a single page, each test that was executed either successfully or not, with the maximum possible information about the test, its execution, and its recent history. This concentrated presentation facilitates the testers in analyzing the tests, as they no longer need to reach out to different pages and applications. In this way, the users obtain all the information they needs in order to carry out their analysis. First, is being mentioned the role and responsibilities of the test automation engineer in the software development life cycle. Then, are being analyzed the architecture and technologies used for the development of the application, as well as for its design and analysis. Finally, code examples of the key points of the implementation and its corresponding pages are being provided.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω αρχικά τους γονείς μου, για τη στήριξη όλων αυτών των χρόνων. Στη συνέχεια, την Αντιγόνη, για την αγάπη, την υποστήριξή και την υπομονή της, καθώς και για τη φιλολογική επιμέλεια της πτυχιακής εργασίας. Δεν θα μπορούσα να παραλείψω τον Ερμή και τον Μίκη για τον χώρο που καταλαμβάνουν στο σπίτι και στην καρδιά μου. Τέλος, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της πτυχιακής εργασίας, κύριο Παναγιώτη Τζέκη, για την εμπιστοσύνη του στο θέμα της πτυχιακής.

Περιεχόμενα

Πρόλογος iv
Περίληψη v
Abstract vi
Ευχαριστίες vii
Περιεχόμενα viii
Κατάλογος Εικόνων x
Συντομογραφίες xii
Κεφάλαιο 1ο: Εισαγωγή 1
1.1 Εισαγωγή 1
1.2 Ο κύκλος ζωής ανάπτυξης του λογισμικού (SDLC). 1
1.3 Ανάλυση του προβλήματος 2
1.4 Επίλυση του προβλήματος 3
1.5 Δομή εργασίας 4
1.6 Επίλογος 5
Κεφάλαιο 2ο: Τεχνολογίες 6
2.1 Εισαγωγή 6
2.2 Backend 6
2.2.1 C# και .NET 8 6
2.2.2 Entity Framework Core 6
2.3 Frontend 7
2.3.1 Blazor 7
2.3.2 MudBlazor 8
2.4 Επικοινωνία με Azure DevOps 9
2.4.1 Azure DevOps REST API 9
2.4.2 Postman 9
2.5 Building και deploying με χρήση Azure DevOps pipelines 9
2.6 Δημιουργία UI test cases 10
2.6.1 Selenium 10
2.6.2 NUnit 10
2.7 Επίλογος 10
Κεφάλαιο 3ο: Αρχιτεκτονική και σχεδιαστικά πρότυπα 12
3.1 Εισαγωγή 12

3.2 Controller Service Repository pattern 12
3.2.1 Controller layer 12
3.2.2 Service layer 13
3.2.3 Repository layer 13
3.2.4 Οφέλη του προτύπου CSR 13
3.3 Dependency injection 14
3.4 Επίλογος 15
Κεφάλαιο 4ο: Υλοποίηση της εφαρμογής 16
4.1 Εισαγωγή 16
4.2 Υλοποίηση του Blazor project 17
4.3 Υλοποίηση του backend 34
4.4 Επίλογος 54
Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης 55
ΒΙΒΛΙΟΓΡΑΦΙΑ 56

Κατάλογος Εικόνων

Εικόνα 1.1: Κύκλος ζωής ανάπτυξης λογισμικού 2
Εικόνα 2.1: Blazor και SingalR για αμφίδρομη επικοινωνία μεταξύ server και client 8
Εικόνα 3.1: Τα επίπεδα του προτύπου Controller-Service-Repository και οι αμφίδρομες αλληλεπιδράσεις του 14
Εικόνα 4.1: Ορισμός προφίλ έναρξης τοπικού development 17
Εικόνα 4.2: Ορισμός των σελίδων Home και Test runs στο πλαϊνό μενού 18
Εικόνα 4.3: Το πλαϊνό μενού 18
Εικόνα 4.4: Υλοποίηση του πίνακα των test runs με MudBlazor components 19
Εικόνα 4.5: Υλοποίηση της ανάκτησης των test runs κατά τη φόρτωση της σελίδας 19
Εικόνα 4.6: Η σελίδα test runs 20
Εικόνα 4.7: Απόκρυψη κειμένου σε περίπτωση που αυτό ξεπερνά ένα όριο 21
Εικόνα 4.8: Υλοποίηση της ανάκτησης των test results κατά τη φόρτωση της σελίδας 21
Εικόνα 4.9: Η σελίδα test results 21
Εικόνα 4.10: Το panel test result με αριθμημένες ενότητες 22
Εικόνα 4.11: Υλοποίηση της ενότητας Analysis 23
Εικόνα 4.12: Τα παραγόμενα αρχεία στους αναδιπλούμενους πίνακες 24
Εικόνα 4.13: Υλοποίηση του αναδιπλούμενου πίνακα General attachments 24
Εικόνα 4.14: Υλοποίηση της αποθήκευσης της ανάλυσης 25
Εικόνα 4.15: Υλοποίηση αντιστροφής της κατάστασης ενός αναδιπλούμενου πίνακα 25
Εικόνα 4.16: Υλοποίηση της ανάκτησης των δεδομένων για τη σελίδα test result 26
Εικόνα 4.17: Το panel test case με αριθμημένες ενότητες 26
Εικόνα 4.18: Το pop up παράθυρο των shared steps 27
Εικόνα 4.19: Η υλοποίηση του component των shared steps 28
Εικόνα 4.20: Η υλοποίηση της μεθόδου για την εμφάνιση του shared steps 28
Εικόνα 4.21: Υλοποίηση της ανάκτησης των test runs μέσω του service 29
Εικόνα 4.22: Υλοποίηση της patch κλήσης για την οριστικοποίηση της ανάλυσης 29
Εικόνα 4.23: Υλοποήση της μεθόδου CreateHttpClient 30
Εικόνα 4.24: Κλήση της CreateHttpClient στον constructor της κλάσης TestResultsService 30
Εικόνα 4.25: Το αντικείμενο builder και το configuration των bindings 31
Εικόνα 4.26: Η εγγραφή των http clients στον DI container 31
Εικόνα 4.27: Η εγγραφή των services στον DI container 31
Εικόνα 4.28: Η εγγραφή των repositories στον DI container 31
Εικόνα 4.29: Η εγγραφή των Controllers στον DI container 32
Εικόνα 4.30: Η εγγραφή του MudBlazor στον DI Container 32
Εικόνα 4.31: Η εγγραφή της SQL Server βάσης στον DI Container 32
Εικόνα 4.32: Το χτίσιμο του DI container και η προσαρμογή το middleware 32
Εικόνα 4.33: Διαχείριση σφαλμάτων κατά την έναρξη της εφαρμογή και logging στη βάση 33
Εικόνα 4.34: Το αρχείο appsettings.Development.json 34
Εικόνα 4.35: Το αρχείο NLog.config 34
Εικόνα 4.36: Ορισμός πινάκων Testers και Logs με EF Core 35
Εικόνα 4.37: Η διαμόρφωση των οντοτήτων Log και Tester 36
Εικόνα 4.38: Η εκτέλεση των EF Core εντολών 36
Εικόνα 4.39: Υλοποίηση του interface test runs service 37
Εικόνα 4.40: Υλοποίηση της κλάσης test runs service 37

Εικόνα 4.41: Ο constructor του test runs controller 38
Εικόνα 4.42: Υλοποίηση του get test runs endpoint 38
Εικόνα 4.43: Η μορφή του test runs DTO 39
Εικόνα 4.44: Η μορφή του γενικού wrapper των μοντέλων 39
Εικόνα 4.45: Υλοποίηση του interface test results repository 40
Εικόνα 4.46: Ο constructor του test result repository 40
Εικόνα 4.47: Υλοποίηση των μεθόδων του test result repository 41
Εικόνα 4.48: Υλοποίηση του interface test result service 42
Εικόνα 4.49: Ο constructor του test result service 42
Εικόνα 4.50: Υλοποίηση της μεθόδου για ενημέρωση της αναλύσεων ενός test result 43
Εικόνα 4.51: Υλοποίηση της μεθόδου για ανάκτηση TAEs 43
Εικόνα 4.52: Υλοποίηση του update test result endpoint 44
Εικόνα 4.53: Η μορφή του test result DTO 45
Εικόνα 4.54: Υλοποίηση της μεθόδου για την ομαδοποίηση των test attachments 47
Εικόνα 4.55: Μεθόδος για την ανάκτηση των test attachments και χρήση της μεθόδου ομαδοποίησης 47
Εικόνα 4.56: Η μορφή του test attachment DTO 48
Εικόνα 4.57: Υλοποίηση της μεθόδου για την εξαγωγή των work item steps 50
Εικόνα 4.58: Υλοποίηση της μεθόδου για την εξαγωγή των work item data 51
Εικόνα 4.59: Υλοποίηση της μεθόδου για την εξαγωγή των work item parameters 51
Εικόνα 4.60: Η μορφή του μοντέλου work item fields raw DTO 53
Εικόνα 4.61: Η μορφή του μοντέλου work item fields DTO 54

Συντομογραφίες

Δ.Ε.	Διπλωματική Εργασία
ΔΙΠΑΕ	Διεθνές Πανεπιστήμιο Ελλάδος
Π.Ε.	Πτυχιακή Εργασία
T. A. E	Test Automation Engineer
S.D.L.C.	Software Development Life Cycle
CI/CD	Continuous Integration and Continuous Delivery/Deployment
Q.A.	Quality Assurance
C.S.R.	Controller Service Repository
D.I.	Dependency Injection
U.I.	User Interface
CRUD	Create Read Update Delete

Κεφάλαιο 1ο: Εισαγωγή

1.1 Εισαγωγή

Στην εποχή του ψηφιακού μετασχηματισμού, οι σύγχρονες εφαρμογές λογισμικού έχουν γίνει αναπόσπαστο κομμάτι σχεδόν κάθε πτυχής της προσωπικής και επαγγελματικής ζωής. Από πλατφόρμες e-commerce και υπηρεσίες χρηματοοικονομικών, μέχρι συστήματα υγειονομικής περίθαλψης και εφαρμογές ψυχαγωγίας, η ζήτηση για επεκτάσιμες, αξιόπιστες και ασφαλείς λύσεις λογισμικού βρίσκεται σε εξαιρετικά υψηλά επίπεδα. Αυτές οι εφαρμογές χαρακτηρίζονται από την πολυπλοκότητά τους, την εξάρτηση σε κατανεμημένα συστήματα και τις συχνές ενημερώσεις για την ικανοποίηση των αναγκών των χρηστών και των επιχειρήσεων. Καθώς οι οργανισμοί εκπαιδεύονται και υιοθετούν πρακτικές agile και συστήνουν τμήματα DevOps, ο ρυθμός ανάπτυξης λογισμικού έχει επιταχυνθεί, με CI/CD pipelines να πραγματοποιούν την ταχεία κυκλοφορία νέων λειτουργιών και την επίλυση των σφαλμάτων.

Με αυτήν την εξέλιξη στην ανάπτυξη του λογισμικού, προκύπτει η κρίσιμη ανάγκη για αξιόπιστες και αποδοτικές πρακτικές testing. Οι σύγχρονες εφαρμογές πρέπει να πληρούν αυστηρά πρότυπα για τη λειτουργικοτητά, την απόδοση, τη χρηστικότητα και την ασφάλεια, συχνά κάτω από δύσκολες συνθήκες, όπως την επεξεργασία δεδομένων σε πραγματικό χρόνο. Η αποτυχία κάλυψης αυτών των απαιτήσεων, μπορεί να οδηγήσει σε κρίσιμα σφάλματα, ευπάθειες ασφαλείας ή ακόμα και διακοπές συστημάτων, προκαλώντας πιθανώς σημαντικές ζημίες και οικονομικές απώλειες [1].

Το testing σύγχρονων εφαρμογών περιπλέκεται περαιτέρω από την ποικιλομορφία της αρχιτεκτονικής τους, η οποία περιλαμβάνει cloud-native εφαρμογές, microservices και APIs. Αυτά τα συστήματα απαιτούν όχι μόνο functional testing αλλά και performance, integration και security testing, για να διασφαλιστεί ότι λειτουργούν ομαλά σε διάφορα περιβάλλοντα. Επιπλέον, η αυξανόμενη εξάρτηση από την αυτοματοποίηση έχει αλλάξει το τρόπο που πραγματοποιείται το testing. Το test automation επιτρέπει ταχύτερους κύκλους feedback, υποστηρίζει CI/CD και διασφαλίζει την πλήρη κάλυψη με tests (test coverage), καθιστώντας το αναπόσπαστο στοιχείο της σύγχρονης ανάπτυξης λογισμικού [2].

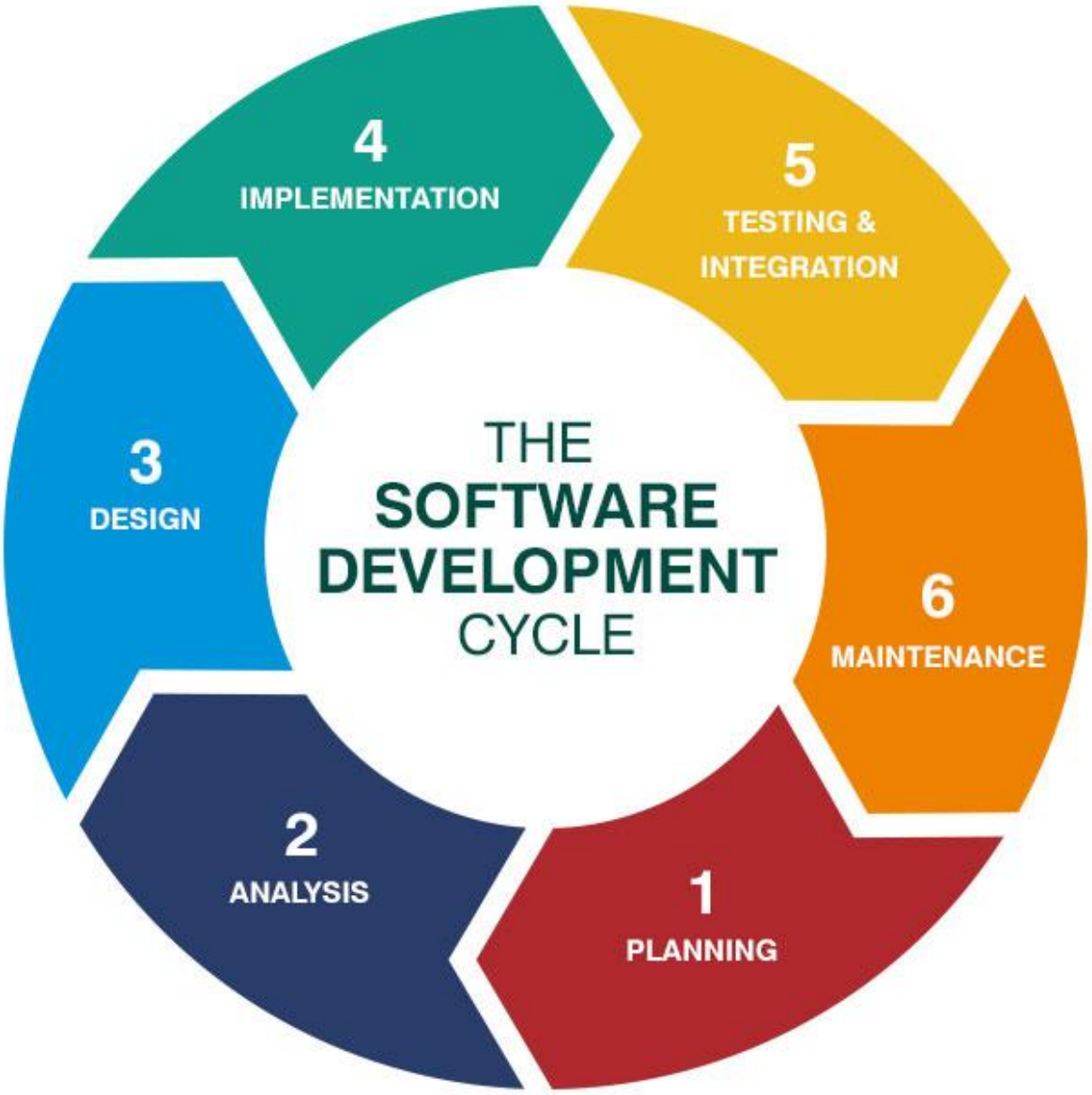
1.2 Ο κύκλος ζωής ανάπτυξης του λογισμικού (SDLC).

Ο κύκλος ζωής ανάπτυξης του λογισμικού (Software Development Life Cycle - SDLC) λειτουργεί ως το θεμελιώδες πλαίσιο για τη δημιουργία λογισμικού υψηλής ποιότητας, με δομημένο και αποδοτικό τρόπο. Περιγράφει μια σειρά από διακριτά στάδια, καθένα με συγκεκριμένους στόχους και παραδοτέα, που καθοδηγούν τις ομάδες από την αρχική ιδέα μέχρι την ανάπτυξη και τη συντήρηση [3]. Στην σημερινή ταχύρυθμη και ανταγωνιστική βιομηχανία λογισμικού, η υιοθέτηση μιας ισχυρής μεθοδολογίας SDLC διασφαλίζει ότι οι διαδικασίες ανάπτυξης είναι οργανωμένες και ικανές να παράγουν αξιόπιστες και επεκτάσιμες λύσεις λογισμικού.

Ο SDLC δίνει έμφαση στη σημασία των σαφώς καθορισμένων σταδίων, όπως ο σχεδιασμός, η ανάλυση, η σχεδίαση, η υλοποίηση, οι δοκιμές, η ανάπτυξη και η συντήρηση. Κάθε φάση είναι αλληλοεξαρτώμενη και συμβάλλει στον τελικό στόχο την παράδοσης λογισμικού που ικανοποιεί τις απαιτήσεις των χρηστών (Εικόνα 1.1).

Το testing και η διασφάλιση ποιότητας (Quality Assurance) αποτελούν αναπόσπαστα στοιχεία του SDLC, διασφαλίζοντας ότι πιθανά σφάλματα εντοπίζονται και διορθώνονται νωρίς στη διαδικασία. Αυτό είναι ιδιαίτερα κρίσιμο στη σύγχρονη ανάπτυξη λογισμικού, όπου οι πρακτικές CI/CD απαιτούν

άμεσο feedback και τη δυνατότητα δυναμικής προσαρμογής στις αλλαγές [4]. Η ενσωμάτωση της αυτοματοποίησης δοκιμών στο πλαίσιο του SDLC έχει μετατρέψει τον τρόπο με τον οποίο οι ομάδες προσεγγίζουν την διασφάλιση της ποιότητας, επιτρέποντας ταχύτερους κύκλους, ενώ διατηρείται η λεπτομερής επικύρωση της λειτουργικότητας και της απόδοσης του λογισμικού. [5]



Εικόνα 1.1: Κύκλος ζωής ανάπτυξης λογισμικού

1.3 Ανάλυση του προβλήματος

Η διαχείριση των αποτελεσμάτων των tests και η ανάλυση των αποτυχημένων test cases σε σύνθετα pipelines μπορεί να είναι μια εξαιρετικά χρονοβόρα και επιρρεπής σε σφάλματα διαδικασία. Μια σημαντική πρόκληση που αντιμετωπίζουν οι ομάδες που χρησιμοποιούν εργαλεία όπως το Azure DevOps, είναι η αναποτελεσματικότητα που προκαλείται από την πλοήγηση μεταξύ πολλαπλών pipelines, αποτελεσμάτων των tests και σχετικών work items κατά τη διάρκεια των διερευνήσεων των tests.

Όταν ένα test αποτυγχάνει στο Azure DevOps, οι test automation engineers(TAEs) συνήθως χρειάζεται να πλοηγηθούν στο αποτυχημένο test case μέσα στο test run, για να συλλέξουν πληροφορίες σχετικά με την αποτυχία. Αυτό μπορεί να περιλαμβάνει την ανασκόπηση των παραχθέντων logs, των μηνυμάτων σφαλμάτων και λεπτομέρειες σχετικά με την εκτέλεση των tests. Ωστόσο, το αναλυτικό πλαίσιο, όπως το ιστορικό του test case, τα σχετικά work items ή προηγούμενες εκτελέσεις, αποθηκεύονται σε διαφορετικά τμήματα του Azure DevOps, απαιτώντας από τους χρήστες να εναλλάσσονται συνεχώς μεταξύ pipelines, αποτελεσμάτων των tests και των σελίδων των test cases. Αυτή η συνεχής εναλλαγή αυξάνει τον φόρτο εργασίας στους TAEs, επιβραδύνει σημαντικά τη διαδικασία διερεύνησης και ενέχει τον κίνδυνο να παραλειφθούν κρίσιμες πληροφορίες που θα μπορούσαν να βοηθήσουν στον εντοπισμό και την επίληση της βασικής αιτίας (root cause) ενός bug.

Επιπλέον, σε περιβάλλοντα με εκτεταμένες test suites, με εκατοντάδες tests και καθημερινές εκτελέσεις tests, αυτή η κατακερματισμένη ροή εργασίας(fragmented workflow) επιδεινώνει τις αναποτελεσματικότητες. Οι προγραμματιστές και οι TAEs μπορεί να αφιερώνουν υπερβολικό χρόνο στη συλλογή δεδομένων αντί να αντιμετωπίζουν τα προβλήματα, μειώνοντας τη συνολική παραγωγικότητα της ομάδας. Επιπρόσθετα, η έλλειψη μιας κεντρικοποιημένης σελίδας, που να συσχετίζει τα αποτελέσματα δοκιμών με σχετικά work items και ιστορικές τάσεις, μπορεί να εμποδίσει τη λήψη αποφάσεων και να καθυστερήσει την επίλυση προβλημάτων, ιδιαίτερα σε CI/CD pipelines που απαιτούν άμεσες ενέργειες.

Αυτές οι αναποτελεσματικότητες αναδεικνύουν την ανάγκη για ένα κεντρικοποιημένο σύστημα που να συγκεντρώνει όλες τις σχετικές πληροφορίες για τα αποτυχημένα test cases σε ένα μέρος. Ένα τέτοιο σύστημα θα επέτρεπε τους TAEs και τους προγραμματιστές να αναλύουν τις αποτυχίες πιο αποδοτικά, εξοικονομώντας χρόνο, μειώνοντας τα σφάλματα και βελτιώνοντας τη συνολική διαδικασία διασφάλισης ποιότητας. Η αντιμετώπιση αυτού του προβλήματος είναι ζωτικής σημασίας για την αύξηση της παραγωγικότητας των TAEs και την εξεσφάλιση της απρόσκοπτης παράδοσης προϊόντων λογισμικού υψηλής ποιότητας.

1.4 Επίλυση του προβλήματος

Η διαδικτυακή εφαρμογή που αναπτύχθηκε στο πλαίσιο αυτής της πτυχιακής εργασίας παρουσιάζει σε ένα κεντρικό σημείο τα κρίσιμα δεδομένα των tests από τον Azure DevOps, προσφέροντας έναν απλοποιημένο και αποδοτικό τρόπο διαχείρισης των test runs, των αποτελεσμάτων των tests και των work items.

Στον πυρήνα του συστήματος βρίσκεται η δυνατότητα του να εμφανίζει όλα τα test runs σε μια σελίδα, παρουσιάζοντας συνοπτικές λεπτομέρειες, όπως ο τίτλος του test run, η κατάσταση της εκτέλεσής του και γενικές πληροφορίες για την κατάσταση των tests που έτρεξαν σε αυτό. Οι TAEs μπορούν να εμβαθύνουν σε επιμέρους test runs για να αποκτήσουν πρόσβαση σε πιο συγκεκριμένες πληροφορίες, σχετικά με όλα τα τεστ που έτρεξαν στο εκάστοτε test run. Οι πληροφορίες αυτές περιέχουν την έκβαση του κάθε τεστ, το μήνυμα σφάλματος που οδήγησε στην αποτυχία του και αναλυτικές πληροφορίες σχετικά με τον λόγο της αποτυχίας αυτής. Οι TAEs μπορούν να διεισδύσουν ακόμα περισσότερο, μέσα σε κάθε test, για να έχουν στη διάθεσή τους όλες τις πληροφορίες του τρέχοντως test, όπως τα βήματα του test, τις επαναλήψεις του και logs σφαλμάτων. Οι πληροφορίες που σχετίζονται με συγκεκριμένες επαναλήψεις είναι οργανωμενες με φιλικό προς το χρήστη τρόπο, επιτρέποντας στους TAEs να εντοπίζουν προβλήματα αποδοτικά. Επιπλέον, η εφαρμογή ενσωματώνει λειτουργίες για τη διαχείριση συνημμένων, όπως logs, screenshots και άλλα αρχεία εντοπισμού σφαλμάτων. Αυτά τα αρχεία κατηγοριοποιούνται ανά επανάληψη, διασφαλίζοντας ότι όλα τα δεδομένα είναι εύκολα προσβάσιμα.

Η εφαρμογή βελτιώνει επίσης τη διαχείριση των work items συνδέοντας tasks, test cases και shared steps απευθείας με τα αντίστοιχα αποτελέσματα των tests. Τα shared steps, που συχνά είναι δύσκολο να εντοπιστούν μέσα στο Azure DevOps, γίνονται εύκολα προσβάσιμα, επιτρέποντας στους χρήστες να προβάλλουν τις λεπτομέρειές τους, σε οποιοδήποτε test case αυτά χρησιμοποιούνται. Αυτή η ενσωμάτωση εξαλείφει την ανάγκη εναλλαγής μεταξύ διαφορετικών σελίδων ή pipelines και βελτιώνει την ορατότητα.

Με την ενοποίηση των δεδομένων σε μια σελίδα, η εφαρμογή αυτή όχι μόνο εξοικονομεί σημαντικό χρόνο, αλλά και ενισχύει την παραγωγικότητα των TAEs. Τους επιτρέπει να επικεντρωθούν στην επίλυση των προβλημάτων και στη βελτίωση της ποιότητας του λογισμικού, αντί να πλοηγούνται μεταξύ pipelines.

Αυτή η διαδικτυακή εφαρμογή αποδεικνύει πως η ενοποίηση των ροών εργασίας μπορεί να αυξήσει την αποδοτικότητα, να μειώσει τα ποσοστά σφαλμάτων και να απλοποιήσει τον κύκλο ζωής ανάπτυξης λογισμικού, αντιμετωπίζοντας ένα κρίσιμο κενό στα υπάρχοντα εργαλεία δοκιμών.

1.5 Δομή εργασίας

Η πτυχιακή εργασία είναι δομημένη σε πέντε κεφάλαια. Το καθένα από τα οποία βασίζεται στα προηγούμενα για να προσδώσει μια ολοκληρωμένη επισκόπηση του σχεδιασμού και της υλοποίησης της διαδικτυακής εφαρμογής βασισμένης στο Blazor. Η δομή ακολουθεί μια λογική ροή, ξεκινώντας από τον εντοπισμό του προβλήματος και καταλήγοντας στα αποτελέσματα και τις ευκαιρίες για περαιτέρω ανάπτυξη.

Το 1ο κεφάλαιο, εισαγωγικό κεφάλαιο, θέτει τη βάση εξετάζοντας τις προκλήσεις που προκύπτουν κατά τον κύκλο ζωής ανάπτυξης λογισμικού, ιδιαίτερα από την οπτική ενός TAE. Εστιάζει στις πολυπλοκότητες της διαχείρισης test runs, test results, test attachments και work items στο Azure Devops. Αναδεικνύει τις αναποτελεσματικότητες και τις κατακερματισμένες ροές εργασίας που αντιμετωπίζουν συχνά οι testers. Το κεφάλαιο εντοπίζει αυτά τα προβλήματα και προτείνει μια εφαρμογή σχεδιασμένη να συγκεντρώνει και να απλοποιεί όλα τα δεδομένα που σχετίζονται με το QA. Η προτεινόμενη λύση στοχεύει στη βελτίωση της προσβασιμότητας και της αποδοτικότητας.

Το 2ο κεφάλαιο περιέχει μια λεπτομερή ανάλυση των τεχνολογιών και των frameworks που χρησιμοποιήθηκαν στο έργο. Για το backend, εξετάζεται η επιλογή της C# και του .NET 8, μαζί με το Entity Framework Core για πρόσβαση και αλληλεπίδραση με τη βάση δεδομένων. Στο frontend αναλύεται η χρήση του Blazor για τη δημιουργία μια σύγχρονης εφαρμογής και του MudBlazor για τη δημιουργία ενός responsive UI.

Η αρχιτεκτονική βάση της εφαρμογής αναλύεται στο 3ο κεφάλαιο. Εξηγείται η χρήση του προτύπου Controller-Service-Repository για τη διασφάλιση καθαρού διαχωρισμού αρμοδιοτήτων και συντηρήσιμου κώδικα. Συζητείται, επίσης, το Dependency Injection ως βασικός μηχανισμός για τη διαχείριση των εξαρτήσεων. Αυτό το κεφάλαιο παρέχει πληροφορίες σχετικά με το τρόπο που αυτά τα πρότυπα συνέβαλαν σε μια επεκτάσιμη λύση.

Το 4ο κεφάλαιο, αυτό της υλοποίησης χωρίζεται σε δύο ενότητες, frontend και backend. Η ενότητα του frontend περιγράφει τα Blazor components, την πλοήγηση και τη δυναμική παρουσίαση των δεδομένων που ανακτώνται από το backend. Η ενότητα του backend καλύπτει την ανάπτυξη των controllers, των services, των repositories και των models. Δίνει έμφαση στη διαχείριση των ενσωματώσεων με το Azure DevOps API. Κάθε λειτουργικότητα, συμπεριλαμβανομένων των test runs, test results, test attachments

και work items περιγράφεται λεπτομερώς. Παρουσιάζονται επίσης οι τρόποι με τους οποίους οι λειτουργίες της εφαρμογής αντιμετωπίζουν τις προκλήσεις που παρουσιάστηκαν κατά την ανάπτυξη.

Το 5ο και τελευταίο κεφάλαιο αναφέρεται στα αποτελέσματα της εφαρμογής, τονίζοντας το οφέλη που προσφέρει στους TAEs. Επίσης, περιγράφει πιθανές περιοχές για μελλοντική βελτίωση, όπως καλύτερες ενημερώσεις σε πραγματικό χρόνο, βελτιωμένες δυνατότητες logging και exception handling, με στόχο την περαιτέρω ενίσχυση της λειτουργικότητας και της χρηστικότητας της εφαρμογής.

1.6 Επίλογος

Η σύγχρονη ανάπτυξη λογισμικού απαιτεί αποδοτικές λύσεις για τη διαχείριση της πολυπλοκότητας και του ρυθμού των testing workflows. Αυτή η πτυχιακή εργασία αντιμετωπίζει αυτές τις προκλήσεις μέσω της ανάπτυξης μιας κεντρικοποιημένης διαδικτυακής εφαρμογής, που ενοποιεί δεδομένα από test runs του Azure DevOps. Ενοποιώντας pipelines, test runs, test cases και work items σε μια σελίδα, η εφαρμογή απλοποιεί τις διερευνήσεις, μειώνει τα σφάλματα και ενισχύει την παραγωγικότητα των TAEs.

Κεφάλαιο 2ο: Τεχνολογίες

2.1 Εισαγωγή

Η επιτυχία οποιουδήποτε έργου λογισμικού επηρεάζεται άμεσα από την επιλογή των τεχνολογιών, καθώς αυτές διαμορφώνουν τη διαδικασία ανάπτυξης, την απόδοση και την εμπειρία χρήστη. Αυτό το κεφάλαιο εξετάζει τις διάφορες τεχνολογίες που χρησιμοποιήθηκαν σε αυτό το έργο, κάθε μια από τις οποίες επιλέχθηκε για να αντιμετωπίσει συγκεκριμένες ανάγκες και προκλήσεις. Ο στόχος ήταν η δημιουργία μιας σύγχρονης, αποδοτικής και φιλικής προς τον χρήστη εφαρμογής, ικανής να ανακτά και να παρουσιάζει δεδομένα test runs από τον Azure DevOps.

Αυτό το κεφάλαιο αναλύει τον ρόλο αυτών των τεχνολογιών στο παρόν έργο, υπογραμμίζοντας τα χαρακτηριστικά, τα οφέλη και τη συμβολή τους στην επίτευξη των στόχων του συστήματος. Αξιοποιώντας αυτά τα εργαλεία, η εφαρμογή προσφέρει μια επεκτάσιμη και συντηρήσιμη λύση, διασφαλίζοντας την αποδοτικότητα και την αξιοπιστία στην ανάλυση των αποτελεσμάτων.

2.2 Backend

2.2.1 C# και .NET 8

Η C# είναι μια ευέλικτη και σύχρονη αντικειμενοστραφής γλώσσα προγραμματισμού, κατάλληλη για τη δημιουργία υψηλής απόδοσης εφαρμογών. Στην πτυχιακή αυτή εργασία, το .NET 8, η τελευταία σταθερή και long-term support έκδοση της πλατφόρμας ανάπτυξης της Microsoft χρησιμοποιείται ως ο πυρήνας της Blazor Server εφαρμογής. Το .NET 8 παρέχει χαρακτηριστικά όπως βελτιωμένη απόδοση, καλύτερη διεχείριση μνήμης και άμεση ενσωμάτωση με cloud-native αρχιτεκτονικές. Η υποστήριξη για dependency injection και οι ενσωματωμένες βιβλιοθήκες του διευκολύνουν τη δημιουργία επεκτάσιμων εφαρμογών, μειώνοντας τον χρόνο ανάπτυξης. Η επιλογή του .NET 8 διασφαλίζει τη συμβατότητα με σύγχρονες πρακτικές και εργαλεία, συμπεριλαμβανομένης της ενσωμάτωσης με το Azure DevOps REST API.

2.2.2 Entity Framework Core

Το Entity Framework Core (EF Core) είναι ένα ανοιχτού κώδικα object-relational mapping (ORM) framework για το .NET, που απλοποιεί τις αλληλεπιδράσεις με βάσεις δεδομένων. Το EF Core γεφυρώνει το χάσμα μεταξύ της εφαρμογής και της υποκείμενης βάσης δεδομένων, παρέχοντας ένα ισχυρά τυποποιημένο data model που επιτρέπει στους προγραμματιστές να εκτελούν create, read, update, delete (CRUD) λειτουργίες, χωρίς να γράφουν SQL queries. Με την αυτοματοποίηση της δημιουργίας σχημάτων και την υποστήριξη για LINQ queries, το EF Core απλοποιεί τη διαχείριση των αποτελεσμάτων test results, test runs, test cases, work items και των σχετικών δεδομένων στο τρέχον έργο. Η δυνατότητα του να συνεργάζεται με πολλούς παρόχους βάσεων δεδομένων το καθιστά μια ευέλικτη επιλογή για μικρά και μεγάλα συστήματα.

2.3 Frontend

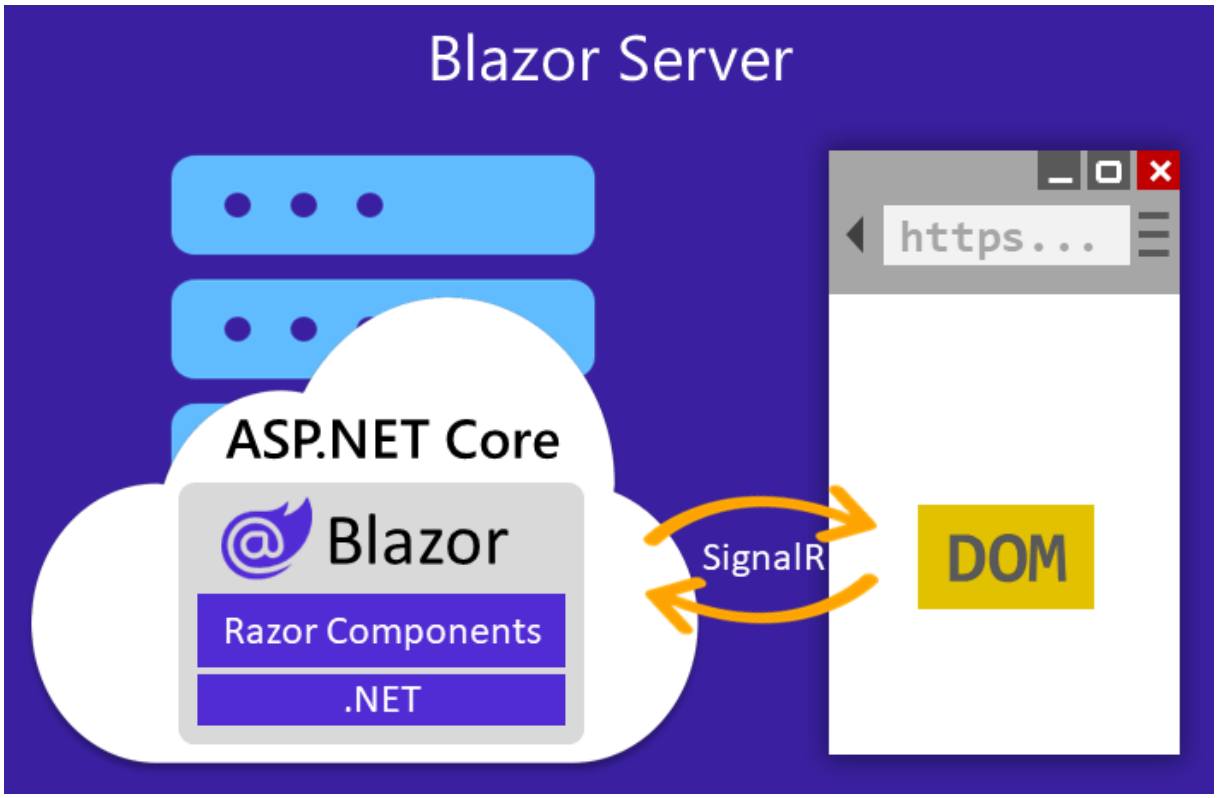
2.3.1 Blazor

Το Blazor είναι framework που αναπτύχθηκε από τη Microsoft για τη δημιουργία διαδραστικών διαδικτυακών εφαρμογών χρησιμοποιώντας το .NET. Ως μέρος του οικοσυστήματος ASP.NET Core, το Blazor επιτρέπει στους προγραμματιστές να δημιουργούν δυναμικές και σύγχρονες διαδικτυακές εφαρμογές με C#, χωρίς να βασίζονται σε Javascript [6]. Το framework υποστηρίζει τόσο server-side όσο και client-side hosting μοντέλα, μέσω του Blazor Server και του Blazor WebAssembly, παρέχοντας στους χρήστες την ευελιξία να επιλέγουν την προσέγγιση που ταιριάζει καλύτερα στις ανάγκες της εφαρμογής τους [7].

Στην τελευταία του έκδοση, ως μέρος του .NET 8, το Blazor εισάγει σημαντικές βελτιώσεις απόδοσης, ενισχυμένο component rendering και καλύτερη ενσωμάτωση με σύγχρονες πρακτικές ανάπτυξης [8] .Χρησιμοποιεί σύνταξη Razor για τον ορισμό των UI components, επιτρέποντας έναν απρόσκοπτο συνδυασμό HTML και C#. Το Blazor Server, που χρησιμοποιήθηκε σε αυτό το έργο, επεξεργάζεται τις αλληλεπιδράσεις του UI στον server και επικοινωνεί τις αλλαγές στον client, μέσω μιας σύνδεσης SignalR (Εικόνα 2.1). Αυτή η προσέγγιση διασφαλίζει ταχείες ενημερώσεις και μικρό μέγεθος λήψης, καθιστώντας το ιδανικό για εφαρμογές που απαιτούν υψηλή απόδοση και αλληλεπιδράσεις σε πραγματικό χρόνο.

Το πλούσιο οικοσύστημα του Blazor περιλαμβάνει υποστήριξη για dependency injection, routing και άμεση ενσωμάτωση με άλλες βιβλιοθήκες του .NET, επιτρέποντας στους χρήστες να δημιουργούν επεκτάσιμες και συντηρήσιμες εφαρμογές. Επιπλέον, η συμβατότητά του με βιβλιοθήκες όπως το MudBlazor, παρέχει ένα εκτεταμένο σύνολο UI components που ακολουθούν τις αρχές του Material Design, διευκολύνοντας την ανάπτυξη καλαίσθητων και responsive σελίδων [9].

Ένα από τα ξεχωριστά χαρακτηριστικά του Blazor είναι η δυνατότητα κοινής χρήσης κώδικα μεταξύ frontend και backend, μειώνοντας την επανάληψη και απλοποιώντας της ανάπτυξη. Με την υιοθέτηση του Blazor σε αυτήν την πτυχιακή εργασία, η εφαρμογή επωφελείται από τις σύγχρονες δυνατότητες ανάπτυξης διαδικτυακών εφαρμογών, διατηρώντας παράλληλα τις ισχυρές επιδόσεις και τα εγγενή χαρακτηριστικά ασφαλείας του NET [8].



Εικόνα 2.1: Blazor και SingalR για αμφίδρομη επικοινωνία μεταξύ server και client

2.3.2 MudBlazor

Το MudBlazor είναι μια ολοκληρωμένη βιβλιοθήκη UI Components, ειδικά σχεδιασμένη για εφαρμογές Blazor, που ακολουθεί τις αρχές του Material Design. Παρέχει στους προγραμματιστές ένα πλούσιο σύνολο προκατασκευασμένων components για τη δημιουργία σύγχρονων εφαρμογών, με ελάχιστη χρήση custom CSS ή Javascript [9]. Ως βιβλιοθήκη ανοιχτού κώδικα, το MudBlazor έχει ευρέως υιοθετηθεί χάρη στην απλότητα, την ευελιξία και την ενεργή υποστήριξη της κοινότητάς του.

Ως βιβλιοθήκη βασισμένη στην αρχιτεκτονική του Blazor, επιτρέπει την απρόσκοπτη ενσωμάτωση τόσο σε εφαρμογές Blazor Server όσο και σε WebAssembly. Το εκτεταμένο σύνολο από UI components περιλαμβάνει μεταξύ άλλων κουμπιά, πίνακες, grids, dialogs και στοιχεία πλοήγησης, καθιστώντας το κατάλληλο για την ανάπτυξη σύνθετων εφαρμογών με συνεπή σχεδιαστική γλώσσα [10]. Επιπλέον, η βιβλιοθήκη προσφέρει προηγμένα χαρακτηριστικά όπως data binding, διαχείριση κατάστασης (state management) και ενημερώσεις σε πραγματικό χρόνο, διευκολύνοντας την ανάπτυξη διαδραστικών σελίδων.

Ένα από τα βασικά πλεονεκτήματα του MudBlazor είναι η δυνατότητα προσαρμογής του. Οι προγραμματιστές μπορούν εύκολα να τροποποιήσουν θέματα, τυπογραφία και χρωματικές παλέτες για να επιτύχουν τα επιθυμητά αποτελέσματα. Επιπλέον, τα components του MudBlazor είναι βελτιστοποιημένα για προσβασιμότητα και αποκριτικότητα, εξασφαλίζοντας μια ομαλή εμπειρία χρήστη σε διάφορες συσκευές και μεγέθη οθόνης [11]. Χρησιμοποιώντας το MudBlazor σε αυτήν την πτυχιακή εργασία, η εφαρμογή αποκτά μια καλοσχεδιασμένη διεπαφή, ενισχύοντας σημαντικά την παραγωγικότητα και την ικανοποίηση των χρηστών.

2.4 Επικοινωνία με Azure DevOps

2.4.1 Azure DevOps REST API

Το REST API του Azure DevOps είναι ένα εργαλείο για την ενσωμάτωση και την αυτοματοποίηση εργασιών μέσα στο Azure DevOps. Παρέχει endpoints για πρόσβαση σε πόρους, όπως test runs, test results, pipelines και work items. Σε αυτήν την πτυχιακή, το REST API χρησιμοποιείται για την ανάκτηση δεδομένων σε πραγματικό χρόνο σχετικά με τα test cases και τα αποτελέσματά τους, επιτρέποντας στο σύστημα να εμφανίζει σχετικές πληροφορίες χωρίς να απαιτείται η χειροκίνητη πλοήγηση στο Azure DevOps. Αυτή η ενσωμάτωση διασφαλίζει ότι η εφαρμογή παραμένει δυναμική και ενημερωμένη, βελτιώνοντας την αποδοτικότητα της ανάλυσης των αποτυχημένων tests.

2.4.2 Postman

Το Postman είναι μια φιλική προς το χρήστη πλατφόρμα για το testing APIs, την αυτοματοποίηση workflows και την ενσωμάτωση με διάφορες υπηρεσίες. Η ευκολία στη χρήση του και τα εκτεταμένα χαρακτηριστικά του, το καθιστούν βασικό εργαλείο για προγραμματιστές και TAEs για την αλληλεπίδραση με APIs, την επικύρωση της λειτουργικότητάς τους και την ανάλυση των αποκρίσεων τους. Στο πλαίσιο του Azure DevOps, το Postman μπορεί να χρησιμοποιηθεί αποτελεσματικά για την αλληλεπίδραση με το Azure DevOps REST API, ώστε να ανακτηθούν και να αναλυθούν δεδομένα που σχετίζονται με test runs, test results και work items [12].

Μέσω του Postman, οι χρήστες μπορούν να στέλνουν HTTP αιτήματα προς τα endpoints του Azure DevOps REST API για την ανάκτηση ή την ενημέρωση πόρων. Για παράδειγμα, τα test runs μπορούν να ανακτηθούν μέσω αποστολής αιτημάτων GET στο αντίστοιχο endpoint, ενώ τα λεπτομερή αποτελέσματα των test, μπορούν να προσεγγιστούν με ερωτήματα μέσω συγκεκριμένου result ID [13]. Επιπλέον, τα work items που σχετίζονται με αποτυχημένα test cases, μπορούν να ανακτηθούν χρησιμοποιώντας τα μοναδικά IDs τους, παρέχοντας μια ολοκληρωμένη εικόνα των αποτυχιών.

2.5 Building και deploying με χρήση Azure DevOps pipelines

Τα build pipelines του Azure DevOps αποτελούν βασικό στειχείο της πλατφόρμας, επιτρέποντας το CI/CD έργων λογισμικού. Τα build pipelines αυτοματοποιούν τη διαδικασία ανάκτησης κώδικα από ένα repository, τη μεταγλώττιση, την εκτέλεση tests και τη διάθεση της εφαρμογής στο προορισμένο περιβάλλον. Αυτά τα pipelines προσφέρουν μια απλοποιημένη, επαναλαμβανόμενη διαδικασία που διασφαλίζει τη συνέπεια και την αξιοπιστία στην παράδοση λογισμικού [14].

Γι’ αυτό το έργο, χρησιμοποίησα ένα build pipeline για τη δημιουργία και τη διάθεση της Blazor εφαρμογής. Η διαδικασία ξεκινάει με την ενσωμάτωση του pipeline με ένα Git repository που φιλοξενεί τον πηγαίο κώδικα της εφαρμογής. Κάθε commit στο origin master branch, ενεργοποιεί το pipeline, διαφαλίζοντας ότι οι νέες αλλαγές προετοιμάζονται αυτόματα για την διάθεση στον hosting server. Το pipeline έχει διαμορφωθεί χρησιμοποιώντας YAML, όπου ορίζονται τα βήματα για το build της εφαρμογής, το publish του σχήματος της βάσης, τη διαχείριση των application pools και του IIS στον hosting server και τέλος την αναβάθμιση της διαδικτυακής εφαρμογής και την επαναφορά της σε λειτουργία.

Αξιοποιώντας τα pipelines του Azure DevOps, πέτυχα μια αξιόπιστη διαδικασία CI/CD που μειώνει τον χρόνο ανάπτυξης και διάθεσης της εφαρμογής, ελαχιστοποιεί τα ανθρώπινα λάθη και διασφαλίζει ότι η Blazor εφαρμογή είναι πάντα ενημερωμένη και λειτουργική.

2.6 Δημιουργία UI test cases

2.6.1 Selenium

Το Selenium είναι ένα ευρέως χρησιμοποιούμενο framework ανοιχτού κώδικα για την αυτοματοποίηση αλληλεπιδράσεων με web browsers, επιτρέποντας στους προγραμματιστές και στους TAEs να εκτελούν functional, regression και end-to-end tests σε διαδικτυακές εφαρμογές. Υποστηρίζει διάφορες γλώσσες προγραμματισμού, όπως C#, Java, Python και JavaScript, και είναι συμβατό με όλους του διαδεδομένους web browsers [15].

Η σουίτα του Selenium αποτελείται από διάφορα πακέτα, όπως το Selenium WebDriver, το Selenium IDE και το Selenium Grid. Το Selenium WebDriver είναι το πιο συχνό πακέτο, προσφέροντας άμεσο έλεγχο στους browsers μέσω μιας προγραμματιστικής διεπαφής για την προσομοίωση ενεργειών, όπως clicks, πληκτρολόγηση και πλοήγηση.

Σε αυτήν την πτυχιακή, το Selenium χρησιμοποιείται για τη συγγραφή αυτοματοποιημένων test case, δημιουργώντας δεδομένα όπως test runs, test results και work items, που αξιοποιούνται αργότερα από την Blazor εφαρμογή.

2.6.2 NUnit

Το NUnit είναι ένα ευρέως διαδεδομένο, testing framework ανοιχτού κώδικα, σχεδιασμένο ειδικά για εφαρμογές .NET. Παρέχει ένα πλούσιο σύνολο χαρακτηριστικών για τη συγγραφή και την εκτέλεση unit tests, αποτελώντας βασικό εργαλείο για τη διασφάλιση της αξιοπιστίας και της λειτουργικότητας των .NET projects. Έχει εξελιχθεί για να υποστηρίζει τις τελευταίες εκδόσεις του .NET, προσφέροντας στους προγραμματιστές τα εργαλεία που χρειάζονται για τη δημιουργία συντηρήσιμου κώδικα [16].

Τα χαρακτηριστικά του περιλαμβάνουν assertions για την επικύρωση των αναμενόμενων αποτελεσμάτων, test fixtures για την οργάνωση των test cases και attributes για τον έλεγχο της συμπεριφοράς, όπως SetUp, TearDown και TestCase. Αυτά τα χαρακτηριστικά απλοποιούν τη διαδικασία δημιουργίας επαναχρησιμοποιούμενων test cases, διασφαλίζοντας πλήρη κάλυψη σε διαφορετικά σημεία της εφαρμογής. Επιπλέον, ενσωματώνεται με μεγάλη ευκολία σε εργαλεία CI/CD, όπως το Azure DevOps, καθιστώντας το ιδανική επιλογή για αυτοματοποιημένα workflows [17].

Το NUnit χρησιμοποιήθηκε για τη συγγραφή και τη διαχείριση αυτοματοποιημένων test cases μέσω του Selenium.

2.7 Επίλογος

Η επιλογή των τεχνολογιών σε αυτό το έργο ήταν καθοριστικής σημασίας για την επίτευξη των στόχων της αποδοτικότητας, της επεκτασιμότητας και του σχεδιασμού με επίκεντρο τον χρήστη. Η C# και το .NET παρέχουν μια ισχυρή και ευέλικτη βάση, επιτρέποντας την απρόσκοπτη ενσωμάτωση με άλλα εργαλεία και υηρεσίες, ενώ προσφέρουν υψηλές επιδόσεις. Το Entity Framework Core απλοποιεί τις αλληλεπιδράσεις με τη βάση δεδομένων, μειώνοντας την πολυπλοκότητα της διαχείρισης δεδομένων. Το MudBlazor βελτιώνει την εμπειρία του χρήστη με ένα πλούσιο σύνολο από UI components. Το Postman λειτουργεί ως σημαντικό εργαλείο για τη δοκιμή και την ανάλυση του Azure DevOps REST API, επιτρέποντας την ομαλή ενσωμάτωση εξωτερικών πηγών δεδομένων. Τέλος, το Selenium και το NUnit έχουν κρίσιμο ρόλο στη δημιουργία των δεδομένων, που απαιτούνται για την λειτουργία της εφαρμογής. Συνδυαστικά, αυτές οι τεχνολογίες δίνουν τη δυνατότητα στην εφαρμογή να ανακτά και να

παρουσιάζει κρίσιμα δεδομένα αποδοτικά, διασφαλίζοντας παράλληλα μια ομαλή εμπειρία για τον χρήστη.

Κεφάλαιο 3ο: Αρχιτεκτονική και σχεδιαστικά πρότυπα

3.1 Εισαγωγή

Μια καλά καθορισμένη αρχιτεκτονική αποτελεί τον θεμέλιο λίθο κάθε επιτυχημένης εφαρμογής λογισμικού, παρέχοντας το δομικό πλαίσιο που διασφαλίζει την επεκτασιμότητα, τη συντηρησιμότητα και την αποδοτικότητα. Αυτό το κεφάλαιο εξετάζει τις αρχιτεκτονικές επιλογές και τα πρότυπα που υιοθετήθηκαν κατά τον σχεδιασμό και την ανάπτυξη αυτού του έργου. Με την εφαρμογή μιας συμπαγούς αρχιτεκτονικής, η εφαρμογή ανακτά, επεξεργάζεται και εμφανίζει αποδοτικά τα δεδομένα από τον Azure DevOps, ενώ διατηρεί έναν καθαρό και επαναχρησιμοποιούμενο κώδικα.

Το κεφάλαιο επικεντρώνεται στο πρότυπο Controller-Service-Repository (CSR), ένα ευρέως γνωστό και χρησιμοποιούμενο αρχιτεκτονικό πρότυπο, που προωθεί έναν σαφή διαχωρισμό ευθυνών (separation of concerns). Κάθε επίπεδο σε αυτό το πρότυπο εξυπηρετεί έναν συγκεκριμένο σκοπό: οι controllers διαχειρίζονται τις εισερχόμενες αιτήσεις, τα services περιλαμβάνουν την επιχειρησιακή λογική της εφαρμογής, και τα repositories ασχολούνται με την πρόσβαση στα δεδομένα. Αυτή η προσέγγιση εξασφαλίζει ότι το σύστημα είναι προσαρμόσιμο σε μελλοντικές αλλαγές, ενώ διευκολύνει το testing και το debugging.

Ακόμη, το κεφάλαιο αναλύει τη χρήση του dependency injection (DI), μια βασική αρχή σχεδιασμού που ενισχύει την ευελιξία και τη δυνατότητα του testing της εφαρμογής. Το DI απλοποιεί την αλληλεπίδραση μεταξύ των components με τον αποσυζευγμένο (decoupled) χειρισμό των εξαρτήσεων, οδηγώντας σε έναν καθαρότερο και πιο συντηρήσιμο κώδικα.

Σε συνδυασμό, αυτά τα αρχιτεκτονικά πρότυπα και οι αρχές παρέχουν μια γερή βάση στην εφαρμογή, διασφαλίζοντας την ικανότητά της να ανταποκρίνεται στις απαιτήσεις της σύγχρονης ανάπτυξης λογισμικού. Αυτό το κεφάλαιο προσφέρει μια λεπτομερή ανάλυση αυτών των προσεγγίσεων, υπογραμμίζοντας την εφαρμογή τους και τα οφέλη που προσφέρουν στο σύστημα.

3.2 Controller Service Repository pattern

Το πρότυπο Controller Service Repository αποτελεί μια από τις πιο δημοφιλείς επιλογές για την ανάπτυξη λογισμικού και κυρίως για διαδικτυακές εφαρμογές. Σχεδιάστηκε για να βελτιώνει την αρθρωτότητα (modularity), επεκτασιμότητα και τη συντηρησιμότητα μιας εφαρμογής, οργανώνοντας τον κώδικα σε διακριτά επίπεδα, το καθένα με συγκεκριμένη ευθύνη. Αυτός ο διαχωρισμός αρμοδιοτήτων επιτυγχάνει να παραμένει καθαρή η λογική της εφαρμογής, ευπρόσδεκτη σε testing και προσαρμόσιμη σε μελλοντικές αλλαγές (Εικόνα 3.1).

3.2.1 Controller layer

Ο controller λειτουργεί ως σημείο εισόδου για τα αιτήματα από κάποια client εφαρμογή και διαχειρίζεται αιτήματα και αποκρίσεις HTTP σε μια διαδικτυακή εφαρμογή. Στο πλαίσιο αυτού του έργου, η Blazor εφαρμογή χρησιμοποιεί controllers για την υποδοχή αιτημάτων από το UI και την προώθησή τους στο service layer. Οι controllers δεν περιέχουν καμία πληροφορία σχετικά με την επιχειρησιακή λογική της εφαρμογής (business logic), λειτουργούν ως ενδιάμεσοι, βεβαιώνοντας ότι η επεξεργασία των δεδομένων γίνεται από το service layer πριν επιστραφούν στην client εφαρμογή. Αυτός ο διαχωρισμός μειώνει την πολυπλοκότητα και βελτιώνει τη σαφήνεια της ροής της εφαρμογής [18].

3.2.2 Service layer

Το service layer περιλαμβάνει την επιχειρησιακή λογική της εφαρμογής. Επεξεργάζεται αιτήματα που λαμβάνει από το controller layer, εφαρμόζει επιχειρησιακούς κανόνες και εκτελεί απαραίτητες λειτουργίες, όπως ανάκτηση δεδομένων από το repository layer, μορφοποίηση και τροποποίηση δεδομένων. Σε αυτό το έργο, το service layer αλληλεπιδρά με το Azure DevOps REST API για την ανάκτηση δεδομένων, όπως test runs, test results και work items. Με την απομόνωση της λογικής στο service layer, το σύστημα γίνεται πιο αρθρωτό, επιτρέποντας ευκολότερες δοκιμές και μελλοντικές επεκτάσεις χωρίς να επηρεάζονται άλλα μέρη της εφαρμογής.

3.2.3 Repository layer

Το repository layer είναι υπεύθυνο για την πρόσβαση στα δεδομένα και την αλληλεπίδραση με τη βάση δεδομένων ή με τις εξωτερικές υπηρεσίες. Απομονώνει τη λογική της πρόσβασης στα δεδομένα, εξασφαλίζοντας ότι το υπόλοιπο σύστημα δεν εξαρτάται από συγκεκριμένες υλοποιήσεις βάσεων δεδομένων ή APIs. Σε αυτό το έργο, το repository layer διαχειρίζεται τις αλληλεπιδράσεις με το service layer, ανακτώντας ή και αποθηκεύοντας δεδομένα στη βάση, ανάλογα με τις ανάγκες της εφαρμογής. Αυτό το επίπεδο βελτιώνει τη συντηρησιμότητα με την κεντρικοποίηση της λογικής πρόσβασης στα δεδομένα, διευκολύνοντας τις ενημερώσεις σε περίπτωση αλλαγής της πηγής δεδομένων [19].

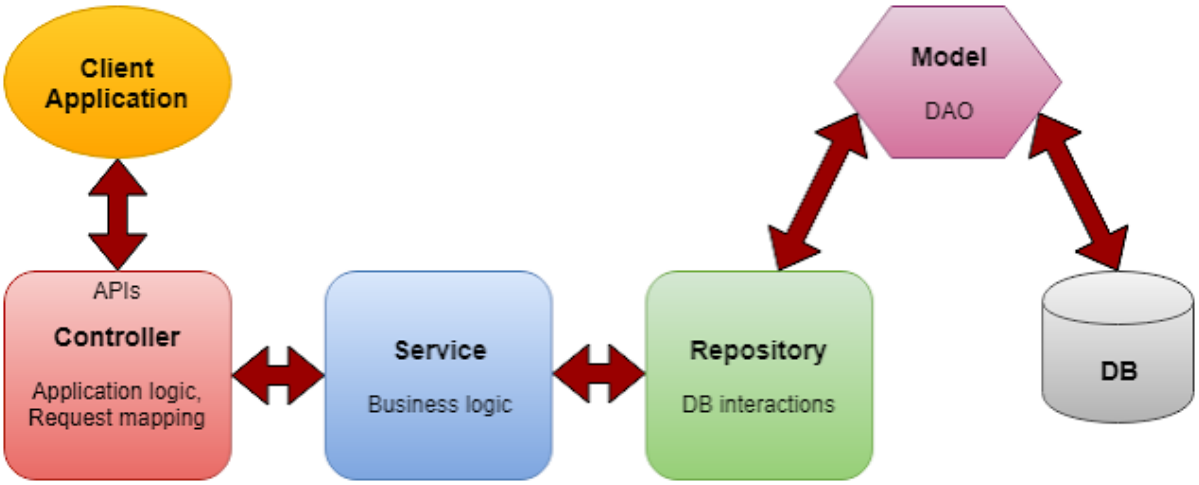
3.2.4 Οφέλη του προτύπου CSR

Με την οργάνωση της εφαρμογής σε αυτά τα επίπεδα, το πρότυπο CSR έχει τα κάτωθι προτερήματα:

Διαχωρισμός Ευθυνών (Separation of concerns): Κάθε επίπεδο επικεντρώνεται σε συγκεκριμένες ευθύνες και αρμοδιότητες, μειώνοντας την επανάληψη κώδικα και βελτιώνοντας την αναγνωσιμότητα.

Δοκιμαστικότητα (Testability): Τα απομονωμένα επίπεδα επιτρέπουν ανεξάρτητο unit testing

Επεκτασιμότητα (Scalability): Η αρθρωτή σχεδίαση διευκολύνει την ενσωμάτωση νέων λειτουργιών ή APIs χωρίς να επηρεάζεται η συνολική αρχιτεκτονική



Εικόνα 3.1: Τα επίπεδα του προτύπου Controller-Service-Repository και οι αμφίδρομες αλληλεπιδράσεις του

3.3 Dependency injection

Το dependency injection είναι μια αρχή σχεδιασμού λογισμικού που προωθεί την αποσύνδεση μεταξύ κλάσεων, εισάγοντας εξαρτήσεις, αντί να τις δημιουργεί απευθείας μέσα στις κλάσεις. Αποτελεί βασικό χαρακτηριστικό σε σύγχρονα frameworks όπως το .NET, όπου χρησιμοποιείται συχνά για τη διαχείριση του κύκλου ζωής των αντικειμένων και την απλοποίηση της ανάπτυξης αρθρωτών, testable και συντηρήσιμων εφαρμογών. Αντί μια κλάση να δημιουργεί τις απαιτούμενες εξαρτήσεις της, ένας DI container αποφασίζει ποιες εξαρτήσεις απαιτούνται και τις παρέχει στην κλάση κατά την εκτέλεση.

Σε αυτό το έργο, το DI χρησιμοποιείται εκτενώς για την αποσύνδεση των components και την απλοποίηση των αλληλεπιδράσεων μεταξύ διαφόρων μερών της εφαρμογής. Το πρότυπο CSR επωφελείται σημαντικά από το DI, καθώς κάθε επίπεδο βασίζεται σε services ή repositories που εισάγονται κατά την εκτέλεση, διασφαλίζοντας ότι τα components παραμένουν ανεξάρτητα και επαναχρησιμοποιήσιμα. Για παράδειγμα, οι controllers εξαρτώνται από services για τη διαχείριση της επιχειρησιακής λογικής, ενώ τα services εξαρτώνται από repositories για την ανάκτηση ή την αποθήκευση δεδομένων. Με την εισαγωγή (injection) αυτών των εξαρτήσεων μέσω του constructor ή των παραμέτρων της μεθόδου, η εφαρμογή αποφεύγει τις hardcoded αναφορές, διευκολύνοντας την αντικατάσταση ή την προσομοίωση (mocking) των components κατά το testing.

Το ενσωματωμένο DI container του .NET αξιοποιείται για τη διαμόρφωση και τη διαχείριση των εξαρτήσεων. Κατά την εκκίνηση της εφαρμογής, τα services, τα repositories και άλλα components εγγράφονται (registered) στο DI container, χρησιμοποιώντας μεθόδους όπως AddScoped, AddTransient ή AddSingleton, ανάλογα με τις απαιτήσεις του κύκλου ζωής τους. Για παράδειγμα, ο Azure DevOps REST API client και τα services εγγράφονται και εισάγονται στα services του container που ανακτούν test runs, test results και work items.

Με την υιοθέτηση του DI, το έργο επιτυγχάνει βελτιωμένο testability, καθώς οι mock υλοποιήσεις μπορούν εύκολα να εισαχθούν κατά τη διάρκεια των unit tests. Επιπλέον, το DI ενισχύει τη συντηρησιμότητα, καθώς οι αλλαγές σε ένα component δεν απαιτούν τροποποιήσεις στις εξαρτώμενες κλάσεις. Αυτή η προσέγγιση εξασφαλίζει έναν καθαρό και επεκτάσιμο σχεδιασμό, ευθυγραμμισμένο με τις σύγχρονες βέλτιστες πρακτικές ανάπτυξης λογισμικού [20].

3.4 Επίλογος

Οι αρχιτεκτονικές επιλογές που έγιναν σε αυτό το έργο αποτελούν τη βάση για τη δημιουργία μιας επεκτάσιμης και συντηρήσιμης εφαρμογής. Με την υιοθέτηση του προτύπου Controller-Service-Repository, το σύστημα επιτυγχάνει έναν σαφή διαχωρισμό ευθυνών, πετυχαίνοντας το κάθε επίπεδο να επικεντρώνεται στις δικές του διακριτές αρμοδιότητες. Αυτή η αρθρωτή προσέγγιση ενισχύει το testability, την αναγνωσιμότητα και την ικανότητα προσαρμογής σε μελλοντικές αλλαγές, χαρακτηριστικά που είναι κρίσιμα για σύγχρονες εφαρμογές λογισμικού.

Το Dependency Injection συμπληρώνει περαιτέρω αυτήν την αρχιτεκτονική, αποσυνδέοντας τα components και διαχειρίζοντας αποτελεσματικά τον κύκλο ζωής τους. Μέσω της χρήσης του DI, το έργο αποφεύγει τις hardcoded εξαρτήσεις, επιτρέποντας απρόσκοπτο testing και ευκολότερη ενσωμάτωση νέων λειτουργιών. Ο συνδυασμός αυτών των αρχιτεκτονικών αρχών παρέχει έναν καθαρό και αποδοτικό σχεδιασμό, όπου η βασική λογική, η πρόσβαση στα δεδομένα και τα επίπεδα αλληλεπίδρασης με τον χρήστη λειτουργούν ανεξάρτητα, αλλά αρμονικά.

Μαζί, το πρότυπο CSR και το Di παρέχουν ένα δομημένο πλαίσιο που ευθυγραμμίζεται με τις βέλτιστες πρακτικές στην ανάπτυξη λογισμικού. Αυτή η αρχιτεκτονική δεν υποστηρίζει μόνο τον τρέχοντα στόχο της εφαρμογής, για την ανάκτηση και παρουσίαση των δεδομένων από το Azure DevOps, αλλά θέτει και τις βάσεις για την επέκταση της λειτουργικότητας της στο μέλλον. Ακολουθώντας αυτές τις αρχές, το έργο αποτελεί παράδειγμα για το πως οι καλά μελετημένες αρχιτεκτονικές αποφάσεις, μπορούν να βελτιώσουν σημαντικά την ποιότητα και τη διάρκεια ζωής ενός συστήματος λογισμικού.

Κεφάλαιο 4ο: Υλοποίηση της εφαρμογής

4.1 Εισαγωγή

Η υλοποίηση αυτής της διαδικτυακής εφαρμογής επικεντρώνεται στη δημιουργία ενός κεντρικού συστήματος για τη διαχείριση των test runs, test results και των work items του Azure DevOps. Το C# solution είναι οργανωμένο σε τρία διαφορετικά projects. Ένα Blazor server για το frontend, ένα class library project για το backend και ένα ακόμα class library για τους κοινούς πόρους, μεταξύ του frontend και backend. Αυτό το κεφάλαιο περιγράφει τις τεχνικές πτυχές κάθε επιπέδου, την ενσωμάτωσή τους και τον τρόπο που συνεργάζονται για να υλοποιήσουν τις λειτουργίες της εφαρμογής.

Το Blazor project αποτελεί το σημείο εισόδου της εφαρμογής. Είναι υπεύθυνο για την εμφάνιση του UI, την διαχείριση των αλληλεπιδράσεων και την επικοινωνία με το backend μέσω αιτημάτων HTTP. Η σελίδα με τη λίστα των Test Runs παρέχει μια συνολική εικόνα όλων των εκτελεσμένων test runs, και την συνολική κατάσταση των tests που περιέχει το κάθε run. Μεταβαίνοντας σε κάποιο run, φορτώνει η λίστα των tests του συγκεκριμένου run. Ο χρήστης μπορεί να έχει μια πιο συγκεκριμένη άποψη για αυτά, καθώς εμφανίζονται πληροφορίες σχετικά με το μήνυμα σφάλματος, το μήνυμα της ανάλυσης του test και ο τύπος του σφάλματος. Ο χρήστης έπειτα, μπορεί να πλοηγηθεί σε όποιο test επιθυμεί μέσα από αυτή τη λίστα, για να φτάσει στη σελίδα της λεπτομερούς ανάλυσης του test. Εκεί εμφανίζονται όλες οι πληροφορίες και οι αναλύσεις του, καθώς και τα παραχθέντα αρχεία του test run. Διαδραστικά στοιχεία, όπως αναδιπλούμενα panels και πίνακες για την παρουσίαση των δεδομένων, βελτιώνουν τη χρηστικότητα και δευκολύνουν την διερεύνηση των λεπτομερών πληροφοριών. Το Blazor project χρησιμοποιεί services που επικοινωνούν με τα endpoints της backend εφαρμογής για την ανάκτηση ή και την ενημέρωση των δεδομένων, διασφαλίζοντας τον διαχωρισμό των αρμοδιοτήτων και την ευκολία της συντήρησης.

Το backend project εφαρμόζει το πρότυπο Controller-Service-Repository για την επεξεργασία, την αποθήκευση και την ανάκτηση δεδομένων. Οι controllers παρέχουν APIs για το frontend, ενώ τα services περιλαμβάνουν τη λογική της εφαρμογής, όπως την ανάλυση των αποτελεσμάτων ή την ομαδοποίηση των log αρχείων του κάθε test. Τα repositories διαχειρίζονται την αλληλεπίδραση με τη βάση δεδομένων, διασφαλίζοντας αποδοτική αποθήκευση και ανάκτηση δεδομένων. Αυτή η αρχιτεκτονική εφαρμόζει έναν αυστηρό διαχωρισμό μεταξύ της εξυπηρέτησης των HTTP αιτημάτων, της επιχειρησιακής λογικής της εφαρμογής και της πρόσβασης στα δεδομένα. Το backend project είναι δομημένο σε διαφορετικούς τομείς, ανάλογα με το πεδίο που χειρίζονται. Έτσι, υπάρχουν διαφορετικά σύνολα αρχείων, για τα test runs, τα test results, work items και test attachments, κάθε ένα από τα οποία έχει δική του υλοποίηση με βάση το CSR.

Το Shared project λειτουργεί ως σύνδεσμος μεταξύ frontend και backend, παρέχοντας Data Transfer Objects (DTOs), enums και βοηθητικές κλάσεις. Τα DTOs διασφαλίζουν τη συνεπή ανταλλαγή δεδομένων, μειώνοντας την επανάληψη και βελτιώνοντας την επικοινωνία μεταξύ των projects. Οι βοηθητικές κλάσεις παρέχουν επεκτάσεις πάνω σε συστημικές μεθόδους, για τη διευκόλυνση της χρήσης τους και την παραμετροποίησή τους βάσει των αναγκών της εφαρμογής.

Μαζί, αυτά τα τρία projects δημιουργούν ένα στιβαρό και αρθρωτό σύστημα, παρέχοντας στους TAEs ένα φιλικό UI για την αποτελεσματική ανάλυση των test cases.

4.2 Υλοποίηση του Blazor project

Το frontend project της διαδικτυακής εφαρμογής που βασίζεται στο Blazor παίζει καθοριστικό ρόλο στην παροχή μια διαδραστικής και φιλικής προς τον χρήστη εφαρμογής. Αυτή η ενότητα περιγράφει τη δομή, τα components και την υλοποίηση του project.

Φάκελος Properties

Περιέχει το αρχείο launchSettings.json, το οποίο προσδιορίζει τα προφίλ έναρξης της εφαρμογής για τοπικό development.

Εικόνα 4.1: Ορισμός προφίλ έναρξης τοπικού development

Φάκελος wwwroot

Φιλοξενεί CSS αρχεία και στατικά assets όπως το bootstrap.

Φάκελος Components

Περιέχει τις blazor σελίδες και τα blazor components. Είναι μοιρασμένος με βάσει τους λειτουργικούς τομείς της εφαρμογής, για αναγνωσιμότητα και ευκολία στην ανάπτυξη.

Οι υποφάκελοι του είναι οι εξής:

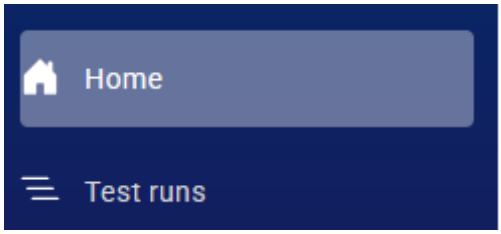
Υποφάκελος Layout

MainLayout.razor: Ορίζει τη γενική διάταξη της εφαρμογής, ενσωματώνοντας το σύστημα διάταξης του MudBlazor. Περιέχει βασικούς providers όπως το MudDialogProvider, MudPopoverProvider και MudSnackbarProvider, για τη διασφάλιση της συνοχής του θέματος και την καθολική χρήση των προσφερόμενων components του MudBlazor.

NavMenu.razor: Υλοποιεί το πλαϊνό μενού πλοήγησης (Εικόνα 4.2), συνδέοντας σελίδες όπως οι Test runs και η Home με το κύριο σώμα του UI (Εικόνα 4.3).

```
<div class="nav-scrollable" onclick="document.querySelector('.navbar-toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu" aria-hidden="true"></span> Home
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="testruns">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Test runs
            </NavLink>
        </div>
```

Εικόνα 4.2: Ορισμός των σελίδων Home και Test runs στο πλαϊνό μενού



Εικόνα 4.3: Το πλαϊνό μενού

Υποφάκελος Pages

Περιέχει τις κύριες σελίδες της εφαρμογής.

TestRuns.razor: Αποτελείται από ένα πίνακα MudTable. Παραθέτει όλα τα test runs που συλλέχθηκαν από τον Azure DevOps. Οι πληροφορίες που παρουσιάζει είναι το ID του test run, ο τίτλος του, με γαλάζιο χρώμα τα συνολικά test που έτρεξαν σε κάθε run, με πράσινο όσα πέρασαν, με κίτρινο όσα δεν έχουν ολοκληρωθεί, με κόκκινο όσα απέτυχαν και τέλος όσα δεν έχουν περάσει από ανάλυση ενός TAE παρουσιάζονται με μοβ χρώμα (Εικόνα 4.4).

Ο χρήστης κάνοντας διπλό κλικ στο επιθυμητό test run μεταφέρεται στην αντίστοιχη σελίδα.

Για την ανάκτηση των test runs, καλείται κατά την φόρτωση της σελίδας με override η συστημική μέθοδος OnInitializedAsync(). Αυτή με τη σειρά της καλεί την μέθοδο GetTestRunAsync(), για να φορτώσει τη λίστα _testRunsList τύπου TestRunDTO με τις επιθυμητές τιμές. Η λίστα _testRunsList, πριν χρησιμοποιηθεί στον πίνακα για να τον γεμίσει με τα αποτελέσματα πρέπει να μετατραπεί σε IEnumerable<TestRunDTO>. Τέλος, υλοποιείται η μέθοδος NavigateToTestResultsPage(), για την πλοήγηση στη σελίδα testResults με διπλό κλικ σε κάποιο test run του πίνακα. (Εικόνα 4.5)

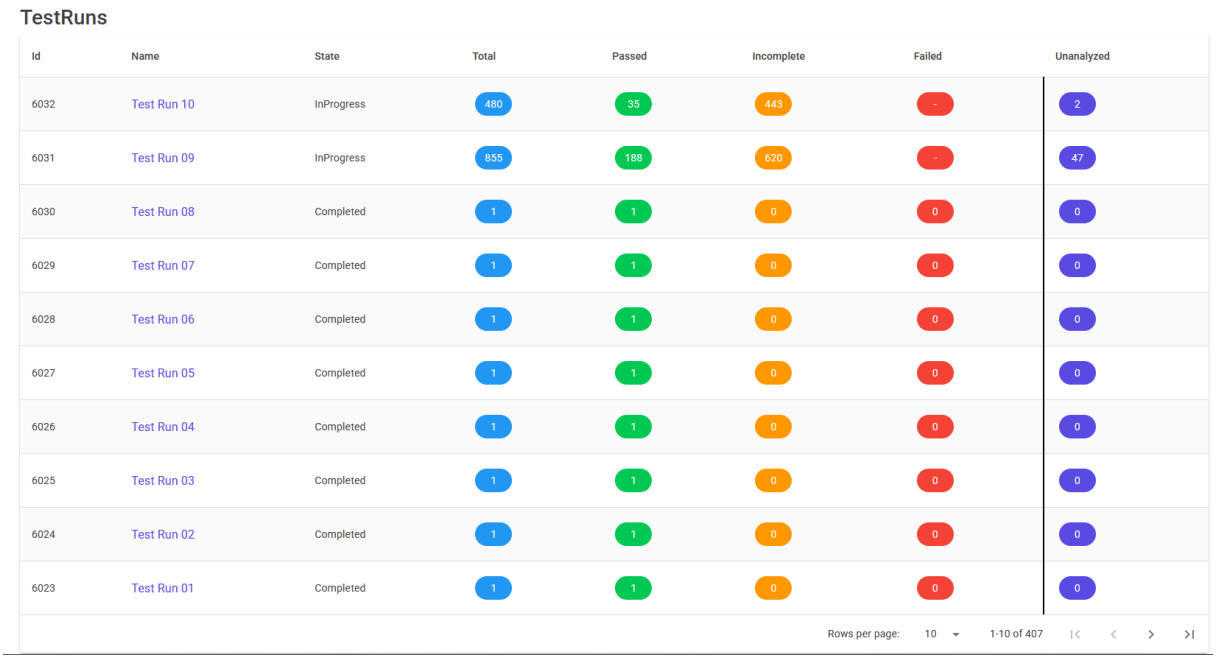
Το τελικό αποτέλεσμα αποτελεί μια καθαρή απεικόνιση των test runs με όσες πληροφορίες χρειάζονται, για την άμεση εικόνα της κατάστασης του λογισμικού (Εικόνα 4.6).

```
@if (_testRuns is not null && _testRuns.Any())
{
    <MudTable T="TestRunDTO" Items="_testRuns" Hover="true" Striped="true" RowsPerPage="10" OnRowClick="@NavigateToTestResultsPage">
        <HeaderContent>
            <MudTh>Id</MudTh>
            <MudTh>Name</MudTh>
            <MudTh>State</MudTh>
            <MudTh>Total</MudTh>
            <MudTh>Passed</MudTh>
            <MudTh>Incomplete</MudTh>
            <MudTh>Failed</MudTh>
            <MudTh>Unanalyzed</MudTh>
        </HeaderContent>
        <RowTemplate Context="testRun">
            <MudTd>@testRun.ID</MudTd>
            <MudTd>
                <MudLink Href="@testRun.WebAccessUrl">@testRun.Name</MudLink>
            </MudTd>
            <MudTd>@testRun.State</MudTd>
            <MudTd>
                <MudChip T="string" Color="Color.Info" Style="width:50px">
                    @testRun.TotalTests
                </MudChip>
            </MudTd>
            <MudTd>
                <MudChip T="string" Color="Color.Success" Style="width:50px">
                    @testRun.PassedTests
                </MudChip>
            </MudTd>
            <MudTd>
                <MudChip T="string" Color="Color.Warning" Style="width:50px">
                    @testRun.IncompleteTests
                </MudChip>
            </MudTd>
            <MudTd>
                <MudChip T="string" Color="Color.Error" Style="width:50px">
                    @(!@testRun.State.Equals("Completed") ? "-" : @testRun.TotalTests - @testRun.PassedTests)
                </MudChip>
            </MudTd>
            <MudTd Style="border-left: 2px solid  black">
                <MudChip T="string" Color="Color.Primary" Style="width:50px">
                    @testRun.UnanalyzedTests
                </MudChip>
            </MudTd>
        </RowTemplate>
        <PagerContent>
            <MudTablePager />
        </PagerContent>
    </MudTable>
}
```

Εικόνα 4.4: Υλοποίηση του πίνακα των test runs με MudBlazor components

```
@code {
    private GenericListModel<TestRunDTO>? _testRunsList;

    private IEnumerable<TestRunDTO> _testRuns => _testRunsList?.Value ?? Enumerable.Empty<TestRunDTO>();

    protected override async Task OnInitializedAsync()
    {
        _testRunsList = await testRunService.GetTestRunAsync();
    }

    private void NavigateToTestResultsPage(TableRowClickEventArgs<TestRunDTO> testRunRow)
    {
        if (testRunRow.MouseEventArgs.Detail == 2)
            navigation.NavigateTo($"testresults/{testRunRow.Item?.ID}", true);
    }
}
```

Εικόνα 4.5: Υλοποίηση της ανάκτησης των test runs κατά τη φόρτωση της σελίδας



Εικόνα 4.6: Η σελίδα test runs

TestResults.razor: Παραθέτει σε έναν πίνακα γενικές πληροφορίες όλων των tests στο τρέχον run. Οι πληροφορίες αυτές είναι το ID του test case, το μήνυμα σφάλματος, το κείμενο της ανάλυσης του TAE, τα οποία αν ξεπερνούν ένα όριο χαρακτήρων, αποκρύπτουν το κείμενο μέχρι εκείνο το όριο και το εμφανίζουν μέσω ενός ToolTip με πέρασμα του cursor (Εικόνα 4.7). Επίσης, παραθέτει το αποτέλεσμα του test Failed με κόκκινο χρώμα ή Passed με πράσινο, τον τύπο της αποτυχίας και την κατηγορία της ανάλυσης. Ακόμα εμφανίζει την ημερομηνία της πρώτης αποτυχίας του κάθε αποτυχημένου test, μετά από την τελευταία επιτυχημένη του εκτέλεση. Ο χρήστης με διπλό κλικ στο επιθυμητό test result, μεταφέρεται στην αντίστοιχη σελίδα

Η ανάκτηση των test results λειτουργεί όπως ακριβώς και η ανάκτηση των test runs, με τις διαφορές ότι καλείται η GetTestResultsAsync(RunId) δίνοντας ως παράμετρο το ID του επιθυμητού test run, η λίστα είναι τύπου TestResultDTO και τέλος, η μέθοδος πλοήγησης στο επιθυμητό test result NavigateToTestResultPage() χρησιμοποιεί το url της Test Result σελίδας (Εικόνα 4.8).

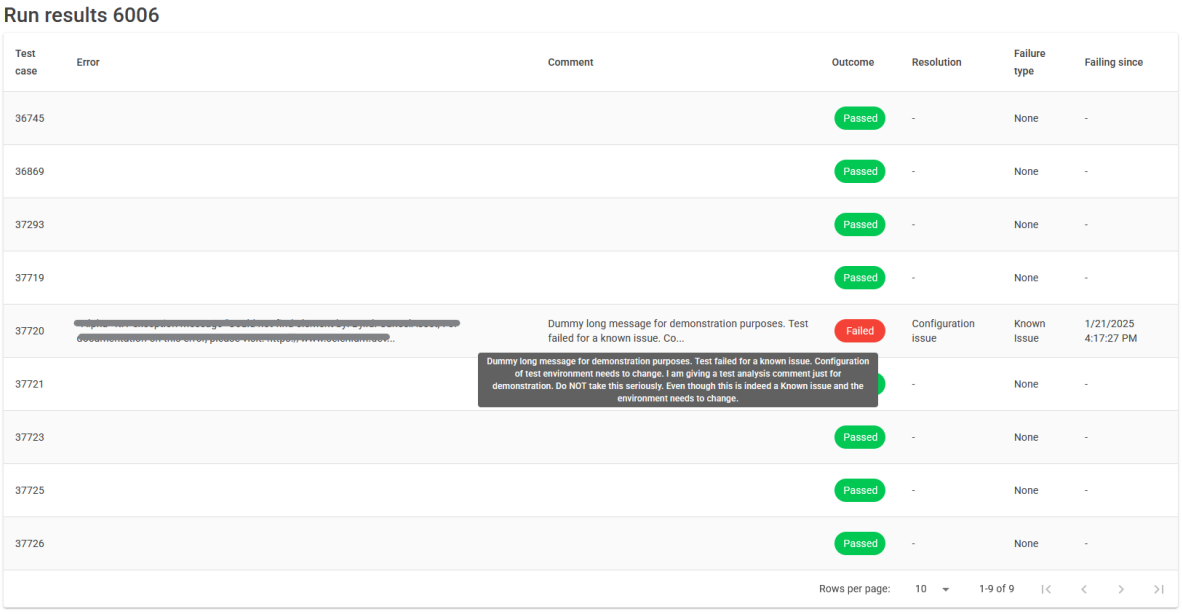
Το τελικό αποτέλεσμα παρουσιάζει σε μια σελίδα τις γενικές πληροφορίες των αποτελεσμάτων του εκάστοτε test run (Εικόνα 4.9).

```
<MudTd>
    @if (string.IsNullOrEmpty(testResult.ErrorMessage))
    {
    }
    else if (testResult.ErrorMessage.Length > 150) {
        <MudTooltip Text="@testResult.ErrorMessage" Style="width:550px">
            @testResult.ErrorMessage.Substring(0, 150)...
        </MudTooltip>
    }
    else
    {
        @testResult.ErrorMessage
    }
</MudTd>
```

Εικόνα 4.7: Απόκρυψη κειμένου σε περίπτωση που αυτό ξεπερνά ένα όριο

```
@code {
    [Parameter]
    public int RunId { get; set; }
    private GenericListModel<TestResultDTO>? _testResultsList;
    private IEnumerable<TestResultDTO> _testResults => _testResultsList?.Value ?? Enumerable.Empty<TestResultDTO>();

    protected override async Task OnInitializedAsync()
    {
        _testResultsList = await testResultsService.GetTestResultsAsync(RunId);
    }

    private void NavigateToTestResultPage(TableRowClickEventArgs<TestResultDTO> testResultRow)
    {
        if (testResultRow.MouseEventArgs.Detail == 2)
            navigation.NavigateTo($"testresults/{RunId}/testresult/{testResultRow.Item?.ID}", true);
    }
}
```

Εικόνα 4.8: Υλοποίηση της ανάκτησης των test results κατά τη φόρτωση της σελίδας



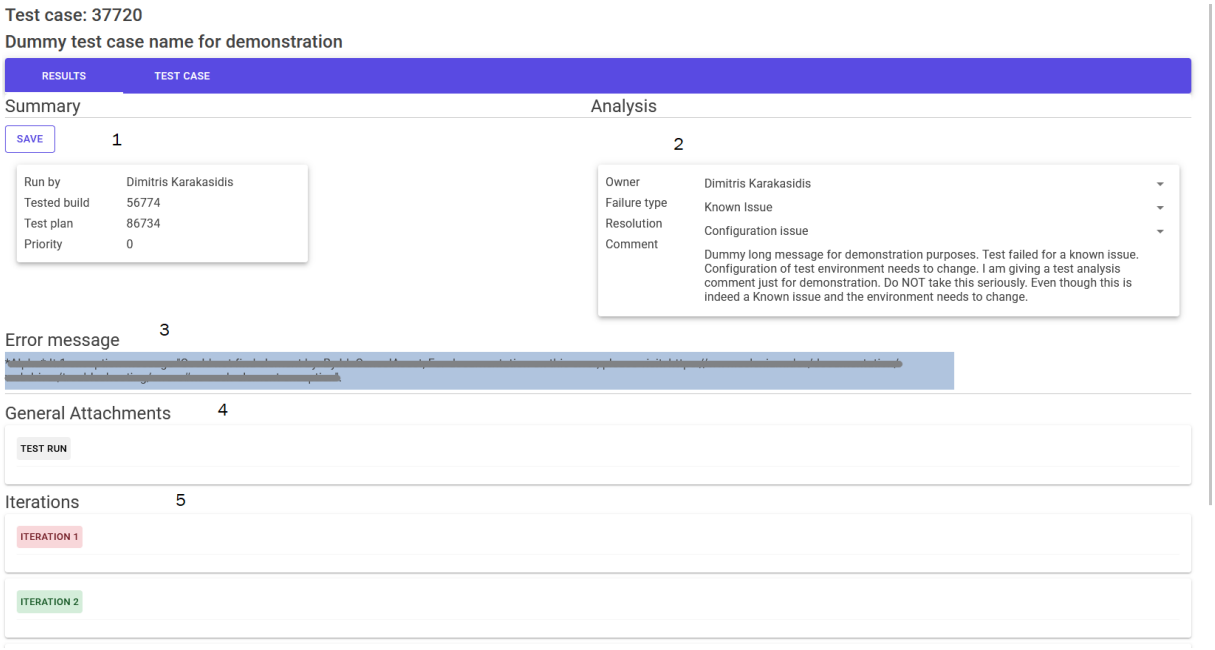
Εικόνα 4.9: Η σελίδα test results

TestResult.razor: Παρουσιάζει ένα συγκεκριμένο test result και όλες τις απαραίτητες πληροφορίες που χρειάζεται ο TAE να γνωρίζει γύρω από αυτό. Αυτές οι πληροφορίες είναι μοιρασμένες σε δύο διαφορετικά panels, καθώς αφορούν δύο διαφορετικούς τομείς ενός test case.

Ως γενικές πληροφορίες, εμφανίζονται στο πάνω μέρος της σελίδας, το ID του test case και ο τίτλος του. Αμέσως μετά εμφανίζονται τα δυο panels, RESULTS και TEST CASE.

Στο πρώτο panel, RESULTS, παρουσιάζονται τα αποτελέσματα από το τρέχον test run.

Χωρισμένο σε πέντε ενότητες, αποτυπώνει όλες τις διαθέσιμες πληροφορίες σχετικά με το run. (Εικόνα 4.10)



Εικόνα 4.10: Το panel test result με αριθμημένες ενότητες

Στην πάνω αριστερή γωνία, η ενότητα Summary, παρέχει μια περίληψη του test run στο οποίο ανήκει, με πληροφορίες για τον TAE που εκκίνησε την εκτέλεση του test run, το Build ID του System Under Test, δηλαδή την έκδοση της εφαρμογής για την οποία έτρεξαν αυτά τα tests, το test plan που ανήκει το τρέχον test και τέλος το Priority που έχει το test. Οι πληροφορίες για το test run περιλαμβάνονται σε μια μεταβλητή τύπου TestResultDTO, η οποία αρχικοποείται κατά τη φόρτωση της σελίδας.

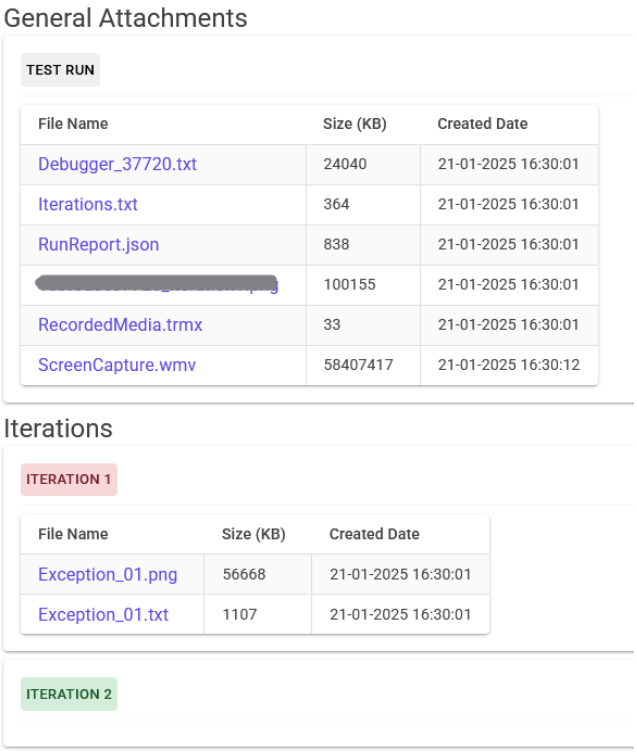
Η πάνω δεξιά γωνία περιέχει την Analysis, την ανάλυση δηλαδή του αποτελέσματος. Παρουσιάζει τις πληροφορίες του TAE που διερεύνησε το test, τον τύπο της αποτυχίας, την κατηγορία της ανάλυσης και το προσωπικό του σχόλιο σχετικά με τον λόγο που απέτυχε το test, αλλά και τι ευθύνεται για αυτήν την αποτυχία. (Εικόνα 4.11)

```
<MudPaper Elevation="4" Height="35%" Width="50%" Style="display: flex; gap: 50px; padding: 10px">
    <MudPaper Elevation="0">
        <MudText Typo="Typo.subtitle1">Owner</MudText>
        <MudText Typo="Typo.subtitle1">Failure type</MudText>
        <MudText Typo="Typo.subtitle1">Resolution</MudText>
        <MudText Typo="Typo.subtitle1">Comment</MudText>
    </MudPaper>
    <MudPaper Elevation="0" style="width:82%">
        <MudSelect T="string" @bind-Value="SelectedOwner" Underline="false">
            @foreach (var tester in _testers)
            {
                <MudSelectItem Value="tester.Name">@tester.Name</MudSelectItem>
            }
        </MudSelect>
        <MudSelect T="FailureType" @bind-Value="SelectedFailureType" Underline="false">
            @foreach (var failureType in Enum.GetValues(typeof(FailureType)).Cast<FailureType>())
            {
                <MudSelectItem Value="failureType">@failureType.GetDescription()</MudSelectItem>
            }
        </MudSelect>
        <MudSelect T="ResolutionState" @bind-Value="SelectedResolution" Underline="false">
            @foreach (var resolution in Enum.GetValues(typeof(ResolutionState)).Cast<ResolutionState>())
            {
                <MudSelectItem Value="resolution">@resolution.GetDescription()</MudSelectItem>
            }
        </MudSelect>
        <MudTextField @bind-Value="_testResult.Comment" AutoGrow="true" MaxLines="8" MaxLength="1000" FullWidth="true" Underline="false">
            @(_testResult.Comment ?? "")
        </MudTextField>
    </MudPaper>
</MudPaper>
```

Εικόνα 4.11: Υλοποίηση της ενότητας Analysis

Αμέσως κάτω από την ενότητα της ανάλυσης, και καταλαμβάνοντας τον οριζόντιο άξονα της σελίδας, βρίσκεται η ενότητα του μηνύματος σφάλματος, με τίτλο Error message. Εκεί αναγράφεται ο λόγος για τον οποίο απέτυχε το τεστ. Είναι ένα πεδίο που ενημερώνεται με δύο τρόπους. Ο ένας είναι μέσω των ίδιων των μηχανισμών του Azure DevOps, σε περίπτωση που το test απέτυχε εξ' αιτίας κάποιου προβλήματος του συστήματος. Ο δεύτερος είναι μέσω του Azure DevOps REST API.

Η επόμενες δύο ενότητες είναι στοιχισμένες κάτω από την προαναφερόμενη. Είναι αυτές που παρουσιάζουν τα αρχεία που παράχθηκαν κατά την εκτέλεση των tests. Χωρίζονται σε General attachments και Iterations. Η πρώτη από τις δύο, κρατά τα γενικά αρχεία της εκτέλεσης, ενώ η δεύτερη εμφανίζει τα αρχεία της κάθε επάναληψης του test. Και οι δυο αυτές ενότητες παρουσιάζουν τα αρχεία μέσω αναδιπλούμενων πινάκων, με πληροφορίες όπως το όνομα, το μέγεθος και την ημερομηνία δημιουργίας του αρχείου.(Εικόνα 4.12) Τα αναδιπλούμενα στοιχεία έχουν γκρι χρώμα στην πρώτη περίπτωση, ενώ στη δεύτερη κόκκινο για τις αποτυχημένες επαναλήψεις του τεστ και πράσινο για τις επιτυχημένες.



Εικόνα 4.12: Τα παραγόμενα αρχεία στους αναδιπλούμενους πίνακες

```
<!--General attachments-->
<MudText Typo="Typo.h5">General Attachments</MudText>
<MudPaper Class="pa-4 mb-2" Elevation="1">
    <MudStack Spacing="2" AlignItems="AlignItems.Start">
        <MudButton Size="Size.Small" OnClick="@(() => OnExpandCollapseClick(0))"
        Style="@GetButtonStyle("default")">
            TEST RUN
        </MudButton>
        <MudDivider />
        <MudCollapse Expanded="@IsExpanded(0)">
            @if (_generalAttachments.Any())
            {
                <MudTable Items="_generalAttachments" Dense="true" Hover="true" Bordered="true" Striped="true">
                    <HeaderContent>
                        <MudTh>File Name</MudTh>
                        <MudTh>Size (KB)</MudTh>
                        <MudTh>Created Date</MudTh>
                    </HeaderContent>
                    <RowTemplate Context="attachment">
                        <MudTd>
                            <MudLink Href="javascript:void(0);" OnClick="() => OpenAttachment(attachment.URL)">
                                @attachment.FileName
                            </MudLink>
                        </MudTd>
                        <MudTd>@attachment.Size</MudTd>
                        <MudTd>@attachment.CreatedDate.ToString("dd-MM-yyyy HH:mm:ss")</MudTd>
                    </RowTemplate>
                </MudTable>
            }
            else
            {
                <MudText>No general attachments available.</MudText>
            }
        </MudCollapse>
    </MudStack>
</MudPaper>
```

Εικόνα 4.13: Υλοποίηση του αναδιπλούμενου πίνακα General attachments

Τέλος, πάνω από την ενότητα της ανάλυσης, βρίσκεται το κουμπί Save. Κατά το click, συλλέγει τις πληροφορίες που προσέθεσε ο ΤΑΕ κατά την ανάλυσή του, τις μοντελοποιεί σε UpdateTestResultDTO και καλεί την μέθοδο UpdateTestResultAsync() (Εικόνα 4.14). Με αυτό ολοκληρώνεται η ανάλυση και ενημερώνει τον Azure DevOps.

```
private async Task SaveChanges()
{
    try
    {
        var updateResultBody = new UpdateTestResultDTO
        {
            ID = ResultId,
            FailureType = (int)SelectedFailureType,
            FailureTypeString = SelectedFailureType.ToString(),
            ResolutionStateId = (int)SelectedResolution,
            Owner = new TesterDTO { ID = string.Empty, DisplayName = SelectedOwner },
            Comment = _testResult?.Comment ?? ""
        };

        var updateResultBodyList = new List<UpdateTestResultDTO>
        {
            updateResultBody
        };

        await testResultsService.UpdateTestResultAsync(updateResultBodyList, RunId);

        Snackbar.Add("Test result updated successfully!", Severity.Success);
    }
    catch (HttpRequestException ex)
    {
        Snackbar.Add($"Failed to update test result: {ex.Message}", Severity.Error);
    }
    catch (Exception ex)
    {
        Snackbar.Add($"An unexpected error occurred: {ex.Message}", Severity.Error);
    }
}
```

Εικόνα 4.14: Υλοποίηση της αποθήκευσης της ανάλυσης

Το κομμάτι της ανάκτησης των δεδομένων και της συμπεριφοράς της τρέχουσας σελίδας, γίνεται στο αμιγώς C# μέρος αυτού του αρχείου. Η έναρξή του σηματοδοτείται με το keyword @code. Σε αυτό το μπλοκ ο C# κώδικας ορίζει τη λογική και τη συμπεριφορά της σελίδας. Χρησιμοποιείται για τη δήλωση μεταβλητών, properties και μεθόδων, καθώς επίσης και τη διαχείριση events. Έτσι, για να οριστεί ο τρόπος με τον οποίο αναδιπλώνονται οι πίνακες των attachments, γράφτηκε μια καθαρά C# μέθοδος που ελέγχει ένα Dictionary<int, bool> για την δυαδική κατάσταση του πίνακα. Αν είναι δηλαδή expanded ή όχι ο συγκεκριμένος πίνακας, και αντιστρέφει την κατάσταση (Εικόνα 4.15).

```
private Dictionary<int, bool> _expandedStates = new();

private void OnExpandCollapseClick(int iterationId)
{
    if (_expandedStates.ContainsKey(iterationId))
        _expandedStates[iterationId] = !_expandedStates[iterationId];
    else
        _expandedStates[iterationId] = true;
}
```

Εικόνα 4.15: Υλοποίηση αντιστροφής της κατάστασης ενός αναδιπλούμενου πίνακα

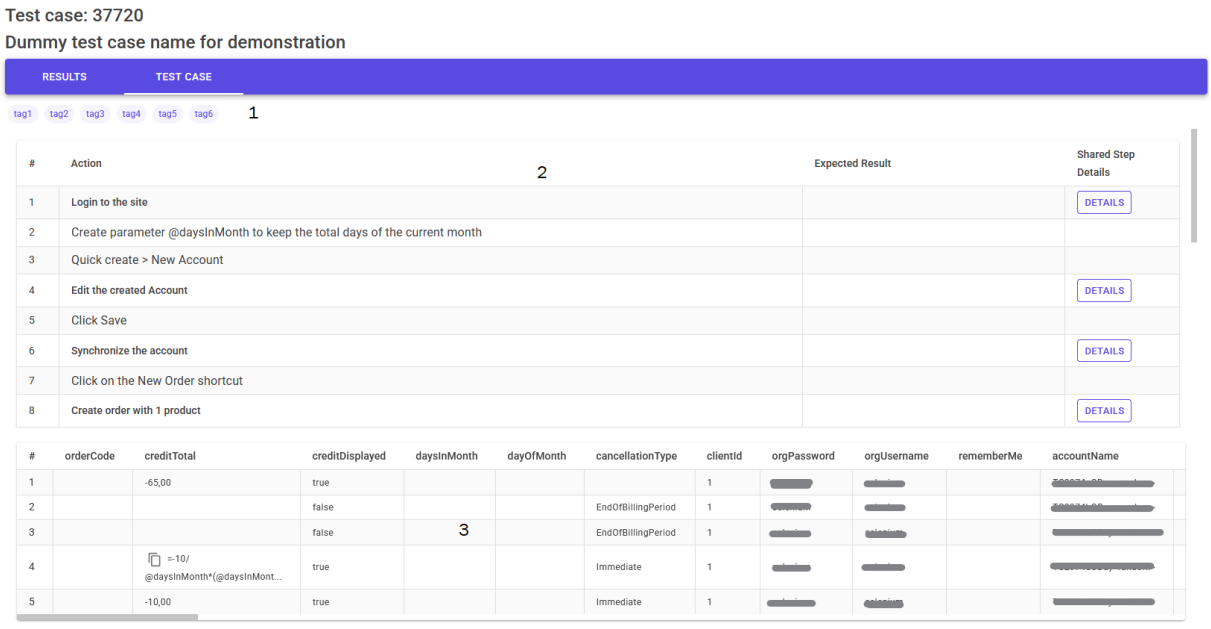
Τέλος, η σημαντικότερη μέθοδος της τρέχουσας σελίδας είναι η OnInitializedAsync(), καθώς αυτή είναι υπεύθυνη για την φόρτωση των δεδομένων και την αρχικοποίηση όλων των απαραίτητων πεδίων και μεταβλητών. Χρησιμοποιεί τις μεθόδους της injected κλάσης TestResultService που επικοινωνεί με το backend της εφαρμογής μέσω HTTP, για την ανάκτηση των δεδομένων (Εικόνα 4.16).

```
protected override async Task OnInitializedAsync()
{
    _testResult = await testResultsService.GetTestResultByIdAsync(RunId, ResultId);
    SelectedOwner = _testResult.Owner.DisplayName;
    _testers = await testResultsService.GetTesterNamesAsync();
    var _attachments = await testAttachmentsService.GetGroupedAttachmentsAsync(RunId, ResultId);

    SelectedFailureType = _testResult.FailureType.GetEnumValueFromDescription<FailureType>();
    SelectedResolution = _testResult.ResolutionState.GetEnumValueFromDescription<ResolutionState>();

    if (_attachments.TryGetValue(0, out var generalAttachments))
    {
        _generalAttachments = generalAttachments;
        _attachments.Remove(0);
    }

    _groupedAttachments = _attachments;

    _workItem = await workItemsService.GetWorkItemAsync(_testResult.TestCase.ID);

    await FetchSharedSteps();
}
```

Εικόνα 4.16: Υλοποίηση της ανάκτησης των δεδομένων για τη σελίδα test result

Στο δεύτερο panel, TEST CASE, παρουσιάζεται το test case, τα βήματα του, οι παράμετροι που χρειάζονται για να τρέξει και τα tags που το τοποθετούν στις αντίστοιχες test suites.

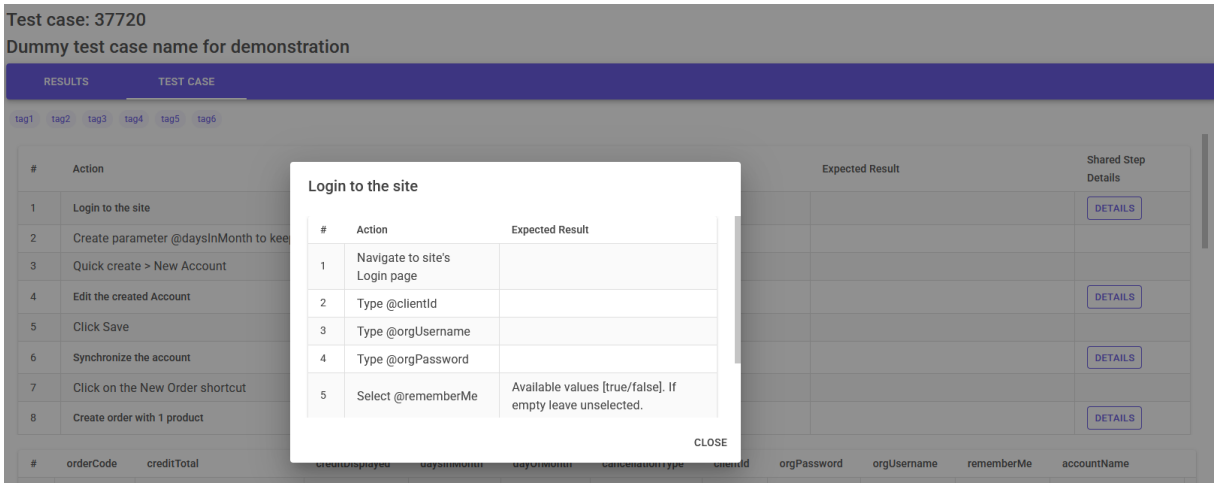
Χωρισμένο σε τρεις ενότητες, παρουσιάζει όλες τις πληροφορίες που χρειάζεται ο TAE για να ολοκληρώσει την ανάλυσή του (Εικόνα 4.17).



Εικόνα 4.17: Το panel test case με αριθμημένες ενότητες

Η πρώτη ενότητα βρίσκεται ακριβώς κάτω από το panel. Είναι ο χώρος που εμφανίζονται τα tags του test case. Βάσει αυτών, το test κατηγοριοποιείται και οργανώνεται σε σουίτες.

Η δεύτερη ενότητα περιέχει τα βήματα του test case, σε φυσική γλώσσα. Τις ενέργειες και τις επικυρώσεις που πρέπει να εκτελέσει κατά το test run. Εμφανίζεται με τη μορφή πίνακα. Η πρώτη στήλη έχει την αρίθμηση των γραμμών, η δεύτερη παρουσιάζει την ενέργεια, η τρίτη την επικύρωση, αν δεν υπάρχει παραμένει κενή, και τέλος η τέταρτη στήλη εμφανίζει ένα κουμπί DETAILS. Αυτό το κουμπί παρουσιάζεται μόνο στις περιπτώσεις που η γραμμή είναι ένα Shared Step, δηλαδή, επαναχρησιμοποιούμενα βήματα ενεργειών και επικυρώσεων. Κατά το κλικ, ανοίγει ένα pop up παράθυρο με τα βήματα του Shared Step, ακολουθώντας την ίδια λογική με αυτή του αρχικού πίνακα (Εικόνα 4.18).



Εικόνα 4.18: Το pop up παράθυρο των shared steps

Για την υλοποίηση της σελίδας ακολουθήθηκαν παρόμοια βήματα με τις προηγούμενες. Αρχικά κλήσεις στις μεθόδους της αντίστοιχης injected κλάσης για την ανάκτηση των δεδομένων και χρήση MudBlazor στοιχείων για το UI.

Ιδιαίτερο ενδιαφέρον παρουσιάζει η υλοποίηση του pop up παραθύρου των Shared Steps. Χρειάστηκε η δημιουργία ενός ακομα αρχείου, SharedStepsDialog.razor, το οποίο φιλοξενεί τα UI στοιχεία, έναν πίνακα δηλαδή με τις αντίστοιχες μορφοποιήσεις και τα απαραίτητα properties για τα δεδομένα. Το αρχείο αυτό, είναι γραμμένο όπως και τα προηγούμενα .razor αρχεία, αλλά δεν διαθέτει route. Δεν είναι δηλαδή σελίδα που μπορεί να πλοηγηθεί κάποιος χρήστης, αλλά ένα επαναχρησιμοποιούμενο UI component (Εικόνα 4.19). Έπειτα, στο αρχικό αρχείο, TestResult.razor, δημιουργήθηκε η μέθοδος ShowSharedStepDialog(). Είναι υπεύθυνη για την δημιουργία ενός UI Dialog στοιχείου στην τρέχουσα σελίδα, τη δημιουργία των παραμέτρων που θα περάσουν στο dialog αρχείο, κατά την κλήση του, ενώ τέλος καλεί την injected MudBlazor μέθοδο ShowAsync(), για την εμφάνιση του dialog (Εικόνα 4.20).

```razor
<MudDialog Style="max-height: 400px; overflow:auto">
    <DialogContent>
        <MudStack Spacing="1">
            <MudTable T="WorkItemStepDTO" Items="WorkItem.Fields.Steps" Dense="true" Bordered="true" Striped="true" Hover="true">
                <HeaderContent>
                    <MudTh>#</MudTh>
                    <MudTh>Action</MudTh>
                    <MudTh>Expected Result</MudTh>
                </HeaderContent>
                <RowTemplate Context="step">
                    <MudTh>@(WorkItem.Fields.Steps.IndexOf(step) + 1)</MudTh>
                    <MudTd><MudText>@(string.IsNullOrEmpty(step.Action) ? "" : (step.Action))</MudText></MudTd>
                    <MudTd><MudText>@(string.IsNullOrEmpty(step.ExpectedResult) ? "" : (step.ExpectedResult))</MudText></MudTd>
                </RowTemplate>
            </MudTable>
            <MudTable T="WorkItemIterationDTO" Items="WorkItem.Fields.Data.Iterations" Dense="true" Bordered="true" Striped="true" Hover="true">
                <HeaderContent>
                    <MudTh>#</MudTh>
                    @foreach (var parameterName in WorkItem.Fields.Data.ParameterNames)
                    {
                        <MudTh>@parameterName</MudTh>
                    }
                </HeaderContent>
                <RowTemplate Context="iteration">
                    <MudTd>@(WorkItem.Fields.Data.Iterations.IndexOf(iteration) + 1)</MudTd>
                    @foreach (var parameterName in WorkItem.Fields.Data.ParameterNames)
                    {
                        <MudTd>
                            <MudText>@iteration.Data[parameterName]</MudText>
                        </MudTd>
                    }
                </RowTemplate>
            </MudTable>
        </MudStack>
    </DialogContent>
    <DialogActions>
        <MudButton OnClick="Cancel">Close</MudButton>
    </DialogActions>
</MudDialog>

@code {
    [Parameter]
    public WorkItemDTO WorkItem { get; set; }

    [CascadingParameter]
    private MudDialogInstance MudDialog { get; set; }

    private void Cancel() => MudDialog.Cancel();
}
```

Εικόνα 4.19: Η υλοποίηση του component των shared steps

```razor
private Task ShowSharedStepDialog(string sharedStepTitle, int sharedStepId)
{
    var options = new DialogOptions { CloseOnEscapeKey = true };
    var parameters = new DialogParameters();

    if (_sharedStepsCache.TryGetValue(sharedStepId, out var sharedStep))
    {
        parameters.Add("WorkItem", sharedStep);
    }

     return dialogService.ShowAsync<SharedStepsDialog>(sharedStepTitle, parameters, options);
}
```

Εικόνα 4.20: Η υλοποίηση της μεθόδου για την εμφάνιση των shared steps

Η τρίτη ενότητα, περιλαμβάνει τις παραμέτρους του test case. Τις μεταβλητές που χρειάζεται, δηλαδή, για να τρέξει. Είναι επίσης δομημένη σε πίνακα, με μεταβλητό πλήθος στηλών και γραμμών. Στις στήλες παρουσιάζονται τα ονόματα των μεταβλητών, και στις γραμμές η κάθε επανάληψη του test case. Οι επαναλήψεις (Iterations) αντιπροσωπεύουν πολλαπλά σύνολα δεδομένων ή συνθηκών που χρησιμοποιεί το test κατά την εκτέλεσή του. Κάθε επανάληψη βοηθά στην επικύρωση του test case σε διάφορα σενάρια, διασφαλίζοντας μεγαλύτερη κάλυψη και αξιοπιστία.

Όπως και την περίπτωση του panel RESULTS, έτσι και σε αυτό το panel, τα δεδομένα αρχικοποιούνται στην μέθοδο OnInitializedAsync() (Εικόνα 4.16), καθώς πρόκειται για την ίδια σελίδα.

Φάκελος Services

Στον φάκελο Services, ζουν όλες οι κλάσεις που υλοποιούν την αλληλεπίδραση με το API της εφαρμογής. Κάθε λογική ενότητα υλοποιεί τη δική της service κλάση.

TestRunsService.cs: Υλοποιεί τη μέθοδο GetTestRunAsync(). Η μέθοδος καλεί ασύγχρονα το endpoint «api/testruns». Διασφαλίζει ότι η απάντηση είναι επιτυχής και το κάνει deserialize στο αντίστοιχο μοντέλο του TestRunDTO σε λίστα. Οι διαδικασίες αυτές περικλείονται σε try catch block για το exception handling. Τέλος, επιστρέφει τη λίστα των test runs. (Εικόνα 4.21)

```
public async Task<GenericListModel<TestRunDTO>> GetTestRunAsync()
{
    var response = await _httpClient.GetAsync("api/testruns");
    response.EnsureSuccessStatusCode();
    string responseBody = await response.Content.ReadAsStringAsync();
    GenericListModel<TestRunDTO>? testRun;
    try
    {
        testRun = JsonConvert.DeserializeObject<GenericListModel<TestRunDTO>>(responseBody);
    }
    catch (Exception ex)
    {
        throw new Exception($"Something went wrong while deserializing the test run response.\nException {ex.Message}");
    }

    if (testRun == null)
        throw new Exception("Test run is null after deserialization");

    return testRun;
}
```

Εικόνα 4.21: Υλοποίηση της ανάκτησης των test runs μέσω του service

TestResultsService.cs: Υλοποιεί όλες τις μεθόδους που σχετίζονται με ανάκτηση δεδομένων για τις σελίδες TestResults.razor και TestResult.razor. Με την ίδια λογική όπως και στο αρχείο TestRunsService.cs, υλοποιεί get async μεθόδους:

- **GetTestResultsAsync(int runId):** λήψη της λίστας των test results ενός συγκεκριμένου test run με βάση το ID του,
- **GetTestResultByIdAsync(int runId, int testResultId):** λήψη ενός συγκεκριμένου test result, με βάση το ID του και το ID του test run,
- **GetTesterNamesAsync():** λήψη των ονομάτων των TAEs που ανήκουν στον εκάστοτε οργανισμό

Επίσης, υλοποιεί την Patch async μέθοδο UpdateTestResultAsync(List<UpdateTestResultDTO> requestDto, int runId), για την οριστικοποίηση της ανάλυσης του αποτυχημένου test case (Εικόνα4.22).

```
public async Task UpdateTestResultAsync(List<UpdateTestResultDTO> requestDto, int runId)
{
    var response = await _httpClient.PatchAsJsonAsync($"api/testrun/{runId}/results", requestDto);
    response.EnsureSuccessStatusCode();
}
```

Εικόνα 4.22: Υλοποίηση της patch κλήσης για την οριστικοποίηση της ανάλυσης

TestAttachmentsService.cs: Υλοποιεί τη μέθοδο GetGroupedAttachmentsAsync(int runId, int testResultId), για την ανάκτηση των παραγόμενων αρχείων κατά την εκτέλεση της εφαρμογής. Επιστρέφει ένα Dictionary<int, List<TestAttachmentDTO>>, που περιέχει την ομαδοποίηση των αρχείων ανά test iteration. Η κλήση του API endpoint ακολουθεί παρόμοια λογική με τις υπόλοιπες get μεθόδους.

WorkItemsService.cs: Υλοποιεί τη μέθοδο GetWorkItemAsync(int workItemId), για την ανάκτηση όλων των πληροφοριών γύρω από ένα test case work item. Αυτό περιλαμβάνει τα steps, iterations, shared steps, tags και πληροφορίες για τους TAEs που δούλεψαν πάνω σε αυτό. Επιστρέφει ένα WorkItemDTO. Η κλήση του API endpoint ακολουθεί επίσης παρόμοια λογική με τις υπόλοιπες get μεθόδους.

QaToolsHttpClientFactoryService.cs: Η κλάση που ενοποιεί τη διαχείριση των ρυθμίσεων για τα api endpoints και της επικοινωνίας προς το backend της εφαρμογής. Μέσω αυτής, απλοποιούνται οι κλήσεις των endpoints και βελτιώνεται η συντηρησιμότητα τους.

Υλοποιεί έναν βασικό constructor για το dependency injection και μια μέθοδο επικύρωσης των δεδομένων, όπως το API URL. Επίσης υλοποιεί την CreateHttpClient() μέθοδο, υπεύθυνη για τη αρχικοποίηση ενός http client, ώστε μέσω αυτού να επιτευχθεί η επικοινωνία με το backend ή οποιαδήποτε εξωτερική υπηρεσία (Εικόνα 4.23).

```
public HttpClient CreateHttpClient()
{
    var client = _httpClientFactory.CreateClient("QaToolsClient");

    client.BaseAddress = new Uri(_qaToolsApi.BaseUrl);
    client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

    return client;
}
```

Εικόνα 4.23: Υλοποίηση της μεθόδου CreateHttpClient

Καλώντας την, κάθε κλάση που απαιτεί επικοινωνία με το API, αποκτά άμεσα τον http client χωρίς να εμπλακεί καμία άλλη διαδικασία (Εικόνα 4.24).

```
private readonly HttpClient _httpClient;

0 references | Dimitris Karakasidis, 18 days ago | 1 author, 1 change
public TestResultsService(QaToolsHttpClientFactoryService qaToolsHttpClientFactoryService)
{
    _httpClient = qaToolsHttpClientFactoryService.CreateHttpClient();
}
```

Εικόνα 4.24: Κλήση της CreateHttpClient στον constructor της κλάσης TestResultsService

Program.cs: Το σημείο εισόδου της εφαρμογής. Εκεί αρχικοποιείται και διαμορφώνεται η διαδικτυακή εφαρμογή. Ακολουθείται με δομημένη προσέγγιση για το registering των services, τη διαμόρφωση του Blazor middleware και τη ρύθμιση της εφαρμογής, αξιοποιώντας το dependency injection και τις αρχές του modular design.

Service Registration:

Το αντικείμενο builder χρησιμοποιείται εκτενώς για τη διαμόρφωση των services. Οι βασικές εγγραφές πάνω σε αυτό περιλαμβάνουν:

- **Configuration bindings:** Ρυθμίσεις για το Azure DevOps και την Blazor backend εφαρμογή. Αντιστοιχίζονται σε strongly-typed κλάσεις για ευκολία στη χρήση και στην επικύρωση της εγκυρότητας τους (Εικόνα 4.25),
- **HttpClients:** Δύο HttpClient instances (AzureDevOpsClient και QaToolsClient) εγγράφονται στον DI container, για τη διαχείριση των επικοινωνιών μέσω API στα αντίστοιχα συστήματα (Εικόνα 4.26),
- **Application Services:** Βασικά services όπως ITestRunService, ITestResultService, εγγράφονται ως Scoped, ενώ τα ClientFactoryServices εγγράφονται ως Singletons, ακολουθώντας τις βέλτιστες πρακτικές σχεδιασμού για διαδικτυακές εφαρμογές (Εικόνα 4.27),
- **Repositories:** Κλάσεις Repository, όπως η ITestResultRepository, εγγράφονται για τη διευκόλυνση της πρόσβασης και της αποθήκευσης των δεδομένων (Εικόνα 4.28),
- **Controllers:** Κλάσεις που επεκτείνουν την συστημική κλάση ControllerBase, ανακαλύπτονται μέσω της AddControllers() και εγγράφονται αυτόματα στον container (Εικόνα 4.29),
- **Frameworks και βιβλιοθήκες:** Το MudBlazor αρχικοποιείται και διαμορφώνεται για την παροχή των UI στοιχείων (Εικόνα 4.30), ενώ το Entity Framework Core ρυθμίζεται για τις αλληλεπιδράσεις με τη βάση δεδομένων (Εικόνα 4.31).

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<AzureDevOpsSettings>(builder.Configuration.GetSection("AzureDevOpsApi"));
builder.Services.Configure<QaToolsApiSettings>(builder.Configuration.GetSection("QaToolsApi"));
```

Εικόνα 4.25: Το αντικείμενο builder και το configuration των bindings

```
//Register Http Clients
builder.Services.AddHttpClient("AzureDevOpsClient");
builder.Services.AddHttpClient("QaToolsClient");
```

Εικόνα 4.26: Η εγγραφή των http clients στον DI container

```
// Register QaTools.Services Services to the container
builder.Services.AddSingleton<AzureDevOpsHttpClientFactoryService>();
builder.Services.AddSingleton<QaToolsHttpClientFactoryService>();
builder.Services.AddScoped<ITestRunService, TestRunService>();
builder.Services.AddScoped<ITestResultService, TestResultService>();
```

Εικόνα 4.27: Η εγγραφή των services στον DI container

```
// Register Repositories to the container
builder.Services.AddScoped<ITestResultRepository, TestResultRepository>();
```

Εικόνα 4.28: Η εγγραφή των repositories στον DI container

```
// Register Controllers to the container
builder.Services.AddControllers();
```

Εικόνα 4.29: Η εγγραφή των Controllers στον DI container

```
// Registering MudBlazor, a Blazor UI components framework
builder.Services.AddMudServices(options => { options.PopoverOptions.CheckForPopoverProvider = false; });
```

Εικόνα 4.30: Η εγγραφή του MudBlazor στον DI container

```
// Register DB mapping to the container
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Εικόνα 4.31: Η εγγραφή της SQL Server βάσης στον DI container

Διαμόρφωση του Middleware:

Αφού χτιστεί ο DI container, το middleware pipeline προσαρμόζεται δυναμικά με βάση το περιβάλλον (Εικόνα 4.32):

- **Development:** Εφαρμόζονται αυτόματα migrations,
- **Production:** Ενεργοποιούνται exception handling μηχανισμοί και middleware, όπως ο μηχανισμός HSTS, για προστασία έναντι των επιθέσεων man-in-the-middle

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
```

Εικόνα 4.32: Το χτίσιμο του DI container και η προσαρμογή του middleware

Error handling και Logging:

Το Nlog ενσωματώνεται για loggin σε επίπεδο βάσης, παρέχοντας έναν ισχυρό μηχανισμό για την καταγραφή των σφαλμάτων, συμπεριλαμβανομένων αυτών που προκύπτουν κατά την εκκίνηση της εφαρμογής, διασφαλίζοντας ότι τα προβλήματα είναι ανιχνεύσιμα (Εικόνα 4.33).

```
var logger = LogManager.Setup().LoadConfigurationFromAppSettings().GetCurrentClassLogger();
logger.Debug("Initializing entry point.");

try
{
    app.Run();
}
catch (Exception ex)
{
    logger.Error(ex, "Exception during initialization.");
    throw;
}
finally
{
    LogManager.Shutdown();
}
```

Εικόνα 4.33: Διαχείριση σφαλμάτων κατά την έναρξη της εφαρμογής και logging στη βάση

Βασικές παρατηρήσεις:

- Τα Services και οι Controllers είναι οργανωμένα με συνοχή, αποτυπώνοντας έναν καθαρό, ευανάγνωστο και συντηρήσιμο κώδικα,
- Η χρήση των Scoped για services και repositories, προάγει την αποδοτικότητα και μειώνει τον ανταγωνισμό των πόρων (resource contention).

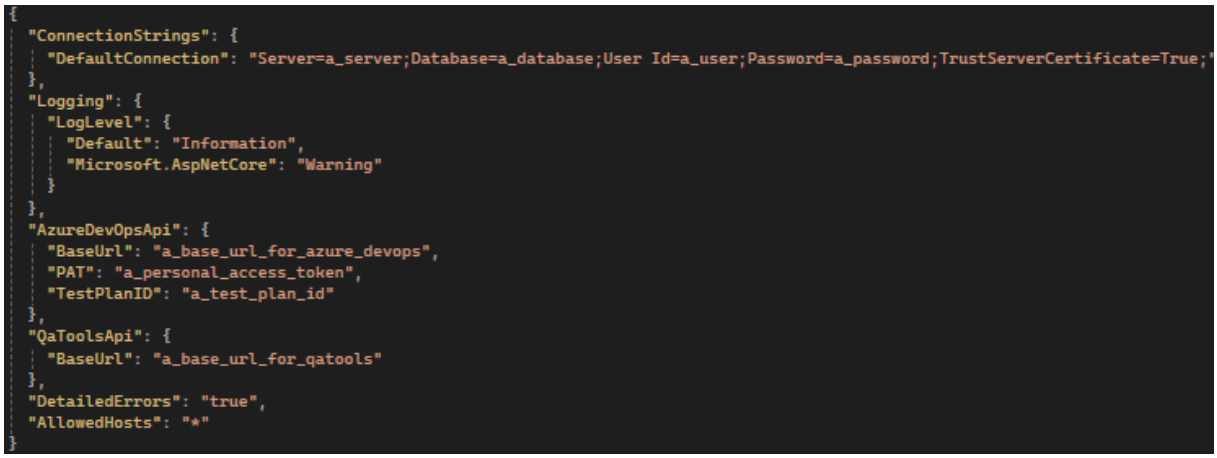
Configuration files

Η εφαρμογή χρησιμοποεί δυο configuration αρχεία. Ένα για την ίδια την εφαρμογή και ένα για το NLog.

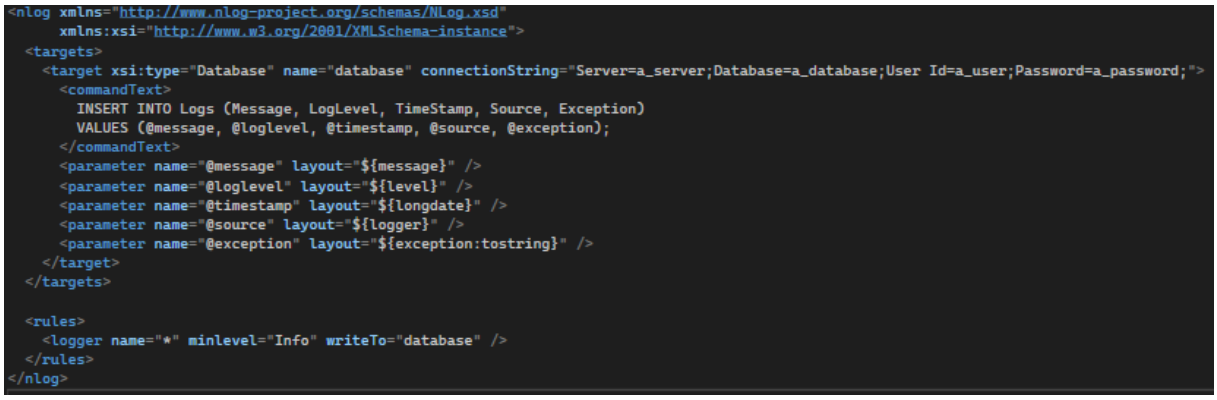
appsettings.json: χρησιμοποιείται σε .NET εφαρμογές για την αποθήκευση καίριων πληροφοριών σε ζεύγη key-value . Οι πληροφορίες μπορεί να περιέχουν connections string για βάσεις δεδομένων, API endpoints και επίπεδα logging. Η χρήση του διευκολύνει την διαχείρηση και την παραμετροποίηση των εφαρμογών.

appsettings.Development.json: αρχείο που αντικαθιστά το appsettings.json σε τοπικό development. Χρησιμοποιείται για συγκεκριμένες ρυθμίσεις στο περιβάλλον του προγραμματιστή. Η χρήση αυτών τον αρχείων αποφασίζεται κατά τον χρόνο εκτέλεσης της εφαρμογής, από την εφαρμογή (Εικόνα 4.34).

NLog.config: χρησιμοποιείται για τις ρυθμίσεις του NLog. Ορίζει κανόνες, targets (αρχείο, βάση, console) και το ελάχιστο επίπεδο των Logs που θα καταγράφονται στη βάση (info, warning, error, fatal) (Εικόνα 4.35).



Εικόνα 4.34: Το αρχείο appsettings.Development.json



Εικόνα 4.35: Το αρχείο NLog.config

4.3 Υλοποίηση του backend

Το Backend της Blazor εφαρμογής σχεδιάστηκε για να χειρίζεται την κύρια λειτουργικότητα του συστήματος, συμπεριλαμβανομένης της επεξεργασίας των δεδομένων, της ενσωμάτωσης του Azure DevOps και της παροχής APIs για το frontend. Αναπτυγμένο με C# και Entity Framework Core, το backend τηρεί αυστηρά το πρότυπο Controller-Service-Repository, το οποίο προάγει την συντηρησιμότητα, την επεκτασιμότητα και τον διαχωρισμό των ευθυνών. Το dependency injection χρησιμοποιείται εκτενώς για τη διαχείριση των εξαρτήσεων και την αποσύνδεση μεταξύ των components.

Παρέχει μια σειρά από API endpoints για τη διαχείριση των λειτουργιών που σχετίζονται με test runs, test results, test attachments και work items. Οι Controllers λειτουργούν ως το κύριο σημείο εισόδου για τη διαχείριση των HTTP αιτημάτων από το frontend. Διασφαλίζουν την αποδοτική επικοινωνία μεταξύ της frontend εφαρμογής και της βάσης δεδομένων ή του Azure DevOps.

Το service layer επεξεργάζεται την επιχειρησιακή λογική της εφαρμογής, διαχειρίζεται τις αποκρίσεις του Azure DevOps API και μετατρέπει τα δεδομένα σε χρήσιμες πληροφορίες για την διαδικτυακή εφαρμογή. Λειτουργίες όπως η ομαδοποίηση των παραγόμενων αρχείων ανά test iteration, η ανάλυση των test results ή η επεξεργασία των test steps και test parameters ενός work item, διαχειρίζονται εδώ.

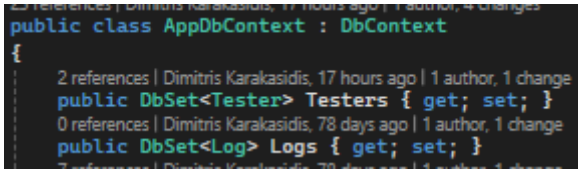
Το Entity Framework Core χρησιμοποιείται ως το κύριο εργαλείο για την αλληλεπίδραση με τη βάση δεδομένων. Απλοποιεί τις CRUD λειτουργίες, τη διαχείριση των σχέσεων μεταξύ των βάσεων και τα migrations του σχήματος. Τα μοντέλα και τα DTOs εξασφαλίζουν τη συνέπεια στη δομή δεδομένων μεταξύ backend και frontend. Επιπλέον, τα DTOs βελτιστοποιούν τη μεταφορά δεδομένων, φιλτράροντας περιττά πεδία και μειώνοντας το μέγεθος των αποκρίσεων.

Η δομή αυτού του σχεδιασμού υποστηρίζει αποτελεσματικά το frontend, πετυχαίνοντας την απρόσκοπτη εμπειρία για τον χρήστη και την αποδοτική επεξεργασία μεγάλου όγκου δεδομένων από τον Azure DevOps.

Δημιουργία και διαχείριση βάσης δεδομένων

Για τη σχεδίαση της βάσης δεδομένων, χρησιμοποιήθηκε η προσέγγιση Code-First του Entity Framework Core. Σε αυτήν την προσέγγιση, η δομή της βάσης δεδομένων ορίζεται μέσω C# κλάσεων, που ονομάζονται models, και μιας κλάσης που εκτείνει την DbContext. Προσφέρεται στο πακέτο EntityFrameworkCore της Microsoft και λειτουργεί ως γέφυρα μεταξύ της εφαρμογής και της βάσης δεδομένων.

Τα properties DbSet με όνομα Testers και Logs, αντιπροσωπεύουν τους πίνακες της βάσης δεδομένων για τις αντίστοιχες οντότητες (entities) Tester και Log (Εικόνα 4.36). Αυτά τα properties επιτρέπουν τις CRUD λειτουργίες στους αντίστοιχους πίνακες, αφαιρώντας την ανάγκη για SQL queries και επιτρέποντας στους προγραμματιστές να δουλεύουν με strongly-typed αντικείμενα.



Εικόνα 4.36: Ορισμός πινάκων Testers και Logs με EF Core

Ο constructor δέχεται ως παράμετρο το DdContextOptions<AppDbContext>, επιτρέποντας το dependency injection για τη δυναμική διαμόρφωση του context, κάτι που είναι απαραίτητο για διαφορετικά περιβάλλοντα (development, staging, production).

Η μέθοδος OnModelCreating() διαμορφώνει τις αντιστοιχίσεις των οντοτήτων (Εικόνα 4.37), για παράδειγμα:

- Η οντότητα Log διαμορφώνεται με το ID ως το κύριο κλειδί.
- Η οντότητα Tester διαμορφώνεται επίσης με το ID ως το κύριο κλειδί, ενώ το property TesterName ορίζεται ως required για να εξασφαλίσει ότι δεν επιτρέπονται null τιμές.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{

    // Logs
    modelBuilder.Entity<Log>()
        .HasKey(l => l.ID);

    // Testers
    modelBuilder.Entity<Tester>()
        .HasKey(t => t.ID);

    modelBuilder.Entity<Tester>()
        .Property(t => t.TesterName)
        .IsRequired();

    base.OnModelCreating(modelBuilder);
}
```

Εικόνα 4.37: Η διαμόρφωση των οντοτήτων Log και Tester

Για τη δημιουργία των αντίστοιχων πινάκων, χρησιμοποιείται η εντολή AddMigration (Εικόνα 4.38) με το όνομα του migration. Με την επιτυχή εκτέλεση της εντολής δημιουργείται μέσω του EF Core ένα migration script με τις αλλαγές. Στη συνέχεια, με την εντολή Update-Database (Εικόνα 4.38) εφαρμόζονται οι αλλαγές στη βάση δεδομένων.

```
Package Manager Console
Package source: All    Default project: QaTools.Services
PM> Add-Migration AddTesterTable
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20250125023835_AddTesterTable'.
Done.
PM>
```

Εικόνα 4.38: Η εκτέλεση των EF Core εντολών

Ανάπτυξη controller-service για τα test runs

Οι test runs κλάσεις της εφαρμογής είναι υπεύθυνες για την ανάκτηση και την επεξεργασία μια λίστας με test runs, που ανακτόνται από τον Azure DevOps. Δεδομένου ότι δεν απαιτείται αλληλεπίδραση με τη βάση δεδομένων, η υλοποίηση επικεντρώνεται αποκλειστικά στις API κλήσεις, την επεξεργασία των δεδομένων και την διάθεσή τους προς το frontend μέσω των controller και service layers.

Service layer

Το service layer περιλαμβάνει τη λογική που απαιτείται για την ανάκτηση και επεξεργασία των test runs. Το interface ITestRunService ορίζει το contract του service, εξασφαλίζοντας τη συνέπεια και προσφέροντας ευελιξία για μελλοντικές αλλαγές. Η βασική μέθοδος που παρέχεται είναι η Task<GenericListModel<TestRunDTO>> GetTestRunsAsync(); (Εικόνα 4.39).

```
public interface ITestRunService
{

    Task<GenericListModel<TestRunDTO>> GetTestRunsAsync();
}
```

Εικόνα 4.39: Υλοποίηση του interface test runs service

Αυτή η μέθοδος ανακτά τα test runs και επιστρέφει τα δεδομένα σε μορφή GenericListMode<TestRunDTO>. Έτσι διασφαλίζει ότι το frontend λαμβάνει έτοιμα προς χρήση δεδομένα.

Η κλάση TestRunService (Εικόνα 4.40) υλοποιεί αυτό το interface. Χρησιμοποιεί το AzureDevOpsHttpClientFactoryService για τη δημιουργία ενός HTTP client που αλληλεπιδρά με τον Azure DevOps. Η βασική μέθοδος της κλάσης, εκτελεί τα εξής βήματα:

- **Κλήση API:** το service στέλνει ένα αίτημα GET στο endpoint: _apis/test/runs?includeRunDetails=true&api-version=7.0,
- **Επικύρωση απόκρισης:** η απόκριση εξετάζεται για το status code χρησιμοποιώντας το response.EnsureSuccessStatusCode(),
- **Deserialization:** το σώμα της JSON απόκρισης γίνεται deserialized σε ένα μοντέλο GenericListModel<TestRunDTO>. Περιέχει τόσο τον αριθμό των test runs όσο και τις λεπτομέρειές τους.
- **Μετασχηματισμός δεδομένων:** Τα test runs ταξινομούνται σε φθίνουσα σειρά με βάση το ID του run, προκειμένου οι πιο πρόσφατες εκτελέσεις να εμφανίζονται πρώτες.
- **Διαχείριση σφαλμάτων:** Πιθανά exceptions κατά το deserialization ή την επεξεργασία των δεδομένων διαχειρίζονται αποτελεσματικά, παρέχοντας κατατοπιστικά μηνύματα σφάλματος.

```
2 references | Dimitris Karakasilis, 16 days ago | 1 author, 1 change
public async Task<GenericListModel<TestRunDTO>> GetTestRunsAsync()
{
    HttpResponseMessage response = await _httpClient.GetAsync("_apis/test/runs?includeRunDetails=true&api-version=7.0");
    response.EnsureSuccessStatusCode();
    string responseBody = await response.Content.ReadAsStringAsync();
    GenericListModel<TestRunDTO>? testRuns;
    try
    {
        testRuns = JsonConvert.DeserializeObject<GenericListModel<TestRunDTO>>(responseBody);
    }
    catch (Exception ex) { throw new Exception($"Something went wrong while deserializing the test runs response.\nException {ex.Message}"); }

    if (testRuns == null)
        throw new Exception("Test runs is null after deserialization");

    var reversedTestRuns = testRuns.Value.OrderByDescending(testRun => testRun.ID).ToList();

    return new GenericListModel<TestRunDTO>
    {
        Count = reversedTestRuns.Count,
        Value = reversedTestRuns
    };
}
```

Εικόνα 4.40: Υλοποίηση της κλάσης test runs service

Controller layer

Ο TestRunsController λειτουργεί ως το σημείο εισόδου για την παροχή των test runs στο frontend. Αλληλεπιδρά με το service layer για την ανάκτηση των απαραίτητων δεδομένων και παρέχει endpoints για την διαχείριση των test runs.

Ο controller βασίζεται στο interface ITestRunService, το οποίο εισάγεται στον constructor του μέσω DI (Εικόνα 4.41). Αυτή η προσέγγιση επιτρέπει:

- **Ευελιξία:** με την εισαγωγή του interface, μπορούν να χρησιμοποιηθούν διαφορετικές υλοποιήσεις του ITestRunService χωρίς να απαιτούνται αλλαγές στον controller.
- **Testability:** Το DI διευκολύνει το unit testing, επιτρέποντας τη χρήση mock services στον controller.

```
public class TestRunsController : ControllerBase
{
    private readonly ITestRunService _testRunService;

    0 references | Dimitris Karakasidis, 18 days ago | 1 author, 1 change
    public TestRunsController(ITestRunService testRunService)
    {
        _testRunService = testRunService;
    }
```

Εικόνα 4.41: Ο constructor του test runs controller

Το GetTestRuns() endpoint, είναι μια HTTP GET μέθοδος που ανακτά τη λίστα των test runs. Ορίζεται στο route /api/testruns μέσω του συστημικού Route attribute του controller (Εικόνα 4.42). Η υλοποίηση της μεθόδου έχεις τα εξής χαρακτηριστικά:

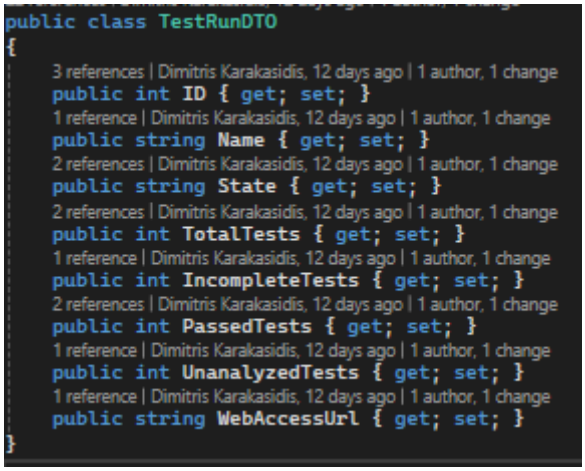
- **Κλήση service**: καλείται η μέθοδος GetTestRunsAsync() μέσω του injected service
- **Διαχείριση απόκρισης**: Τα ανακτημένα δεδομένα επιστρέφονται σε μορφή JSON με τη χρήση της συστημικής μεθόδου Ok(). Έτσι, διασφαλίζεται η τυποποιημένη απόκριση HTTP 200, όταν η ανάκτηση είναι επιτυχής.
- **Ασύγχρονη εκτέλεση:** Η χρήση των async και await εξασφαλίζει non-blocking εκτέλεση, για να βελτιώσει την ανταποκρισιμότητα της εφαρμογής.

```
[ApiController]
[Route("api/testruns")]

[HttpGet]
0 references | Dimitris Karakasidis, 18 days ago | 1 author, 1 change
public async Task<IActionResult> GetTestRuns()
{
    var testRuns = await _testRunService.GetTestRunsAsync();
    return Ok(testRuns);
}
```

Εικόνα 4.42: Υλοποίηση του get test runs endpoint

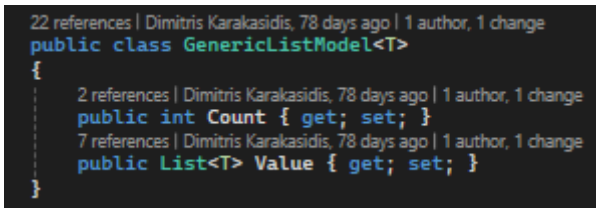
Models

Η μοντελοποίηση των test runs πραγματοποιήθηκε με την κλάση TestRunDTO. Αντιπροσωπεύει τη δομή ενός μεμονωμένου test run που ανακτάται από το Azure DevOps. Περιέχει τα απαραίτητα properties που χρειάζεται η εφαρμογή, για να παρουσιάσει ένα ολοκληρωμένο αποτέλεσμα (Εικόνα 4.43).



Εικόνα 4.43: Η μορφή του test runs DTO

Η κλάση GenericListModel<T> λειτουργεί ως γενικός wrapper για DTOs και μοντέλα. Περιέχει τα properties Count και List<T>Value για το deserialization δεδομένων από αποκρίσεις paginated APIs (Εικόνα 4.44).



Εικόνα 4.44: Η μορφή του γενικού wrapper των μοντέλων

Ανάπτυξη controller-service-repository για τα test results

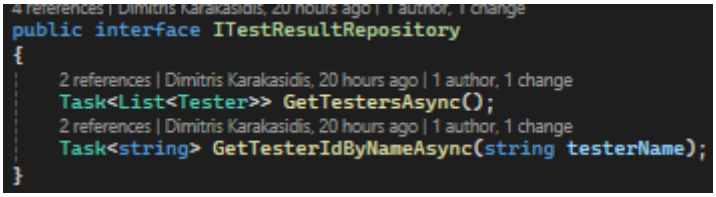
Η υλοποίηση της λειτουργικότητας των test results περιλαμβάνει τον πλήρη συνδυασμό του προτύπου CSR.

Repository layer

Το repository layer λειτουργεί ως η διεπαφή με τη βάση δεδομένων, για τις λειτουργίες που σχετίζονται με τα test results. Χρησιμοποιεί τις μεθόδους του EF Core, για να εξασφαλίσει ότι το service layer παραμένει απομονωμένο από κάθε αλληλεπίδραση με τη βάση δεδομένων. Αυτό επιτυγχάνεται μέσω του interface ITestResultRepository και της υλοποίησής του στην κλάση TestResultRepository.

Το interface ορίζει τις μεθόδους για την ανάκτηση δεδομένων σχετικά με τους TAEs του οργανισμού, από τη βάση δεδομένων της εφαρμογής και περιγράφει τις μεθόδους (Εικόνα 4.45):

- **Task<List<Tester>> GetTestersAsync()**: ανακτά τη λίστα όλων των TAEs που είναι αποθηκευμένοι στη βάση δεδομένων,
- **Task<string> GetTesterIdByNameAsync()**: Επιστρέφει το μοναδικό αναγνωριστικό (ID) για έναν tester βάσει του ονόματός του.



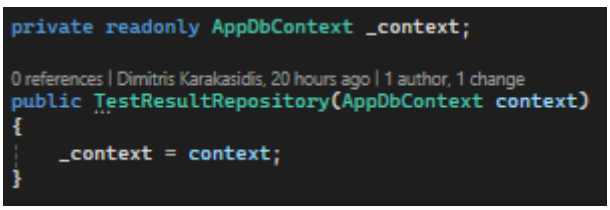
Εικόνα 4.45: Υλοποίηση του interface test result repository

Το interface του repository διασφαλίζει, όπως κάθε interface, την αφαιρετικότητα για mock testing και εύκολη αντικατάσταση των υλοποιήσεων. Επίσης, εξασφαλίζει την ευελιξία, ώστε να μην επηρεάζονται άλλα σημεία της εφαρμογής από ενδεχόμενες αλλαγές στην πρόσβαση των δεδομένων

Η κλάση TestResultRepository υλοποιεί το παραπάνω interface, αξιοποιώντας το EF Core για την επικοινωνία με τη βάση δεδομένων. Αυτή η κλάση αλληλεπιδρά με το AppDbContext, το οποίο αντιπροσωπεύει την βάση δεδομένων της εφαρμογής.

Ο constructor της κλάσης χρησιμοποιεί DI για να κάνει inject ένα instance του AppDbContext. Έτσι πετυχαίνει (Εικόνα 4.46):

- **Εύρος ζωής:** κάθε αίτημα χρησιμοποιεί ξεχωριστό instance του context της βάσης δεδομένων
- **Κοινή παραμετροποίηση:** όλες οι ρυθμίσεις και οι παραμετροποιήσεις της βάσης δεδομένων γίνονται στο ίδιο AppDbContext.



Εικόνα 4.46: Ο constructor του test result repository

Οι υλοποιήσεις των μεθόδων του interface, έχουν τα παρακάτω χαρακτηριστικά:

GetTestersAsync():

- Ασύγχρονη εκτέλεση,
- Άμεση αλληλεπίδραση: καλείται η συστημική μέθοδος του EF Core ToListAsync() για να ανακτήσει τα δεδομένα του πίνακα Testers (Εικόνα 4.47).

GetTesterIdByNameAsync():

- **Φιλτράρισμα:** μέσω της συστημικής βιβλιοθήκης LINQ και της μεθόδου Where(), φιλτράρει τις εγγραφές ως προς το όνομα του TAE,
- **Επιλογή**: με τη μέθοδο Select(), της LINQ, επιλέγει μόνο το ID από τον TAE που φίλτραρε προηγουμένως,
- **Διαχείριση σφαλμάτων:** με τη χρήση της μεθόδου FirstOrDefaultAsync() εξασφαλίζει ότι θα επιστραφεί null τιμή αν δεν βρεθεί κάποια εγγραφή που να ικανοποιεί τις προηγούμενες μεθόδους. Έτσι, αποφεύγονται τα exceptions (Εικόνα 4.47).

```
2 references | Dimitris Karakasidis, 20 hours ago | 1 author, 1 change
public async Task<List<Tester>> GetTestersAsync()
{
    return await _context.Testers.ToListAsync();
}


2 references | Dimitris Karakasidis, 20 hours ago | 1 author, 1 change
public async Task<string> GetTesterIdByNameAsync(string testerName)
{
    return await _context.Testers
        .Where(t => t.TesterName== testerName)
        .Select(x => x.ID)
        .FirstOrDefaultAsync();
}
```

Εικόνα 4.47: Υλοποίηση των μεθόδων του test result repository

Service layer

Το service layer των test result είναι υπεύθυνο για την επεξεργασία και την ανάκτηση των δεδομένων, μέσω της αλληλεπίδρασης με τον Azure DevOps και το repository layer. Αυτό το επίπεδο περιέχει την επιχειρησιακή λογική της σελίδας των test results και test result και γεφυρώνει το repository με τον controller.

Το interface ITestResultService ορίζει τις μεθόδους (Εικόνα 4.48):

- **Task<GenericListModel<TestResultDTO>> GetTestResultsAsync(int runId):** ανακτά όλα τα test results από ένα συγκεκριμένο test run,
- **Task<TestResultDTO> GetTestResultByIdAsync(int runId, int testResultId):** ανακτά ένα συγκεκριμένο test result βάσει του ID του, μέσα από ένα συγκεκριμένο test run,
- **Task<bool> UpdateTestResultAsync(List<UpdateTestResultDTO> requestDto, int runId):** ενημερώνει ένα test result
- **Task<List<TesterNameDTO>> GetTestersAsync():** ανακτά μια λίστα με όλους τους TAEs

```
public interface ITestResultService
{
    2 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    Task<GenericListModel<TestResultDTO>> GetTestResultsAsync(int runId);

    2 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    Task<TestResultDTO> GetTestResultsByIdAsync(int runId, int testResultId);

    2 references | Dimitris Karakasidis, 11 days ago | 1 author, 1 change
    Task<bool> UpdateTestResultAsync(List<UpdateTestResultDTO> requestDto, int runId);

    2 references | Dimitris Karakasidis, 21 hours ago | 1 author, 1 change
    Task<List<TesterNameDTO>> GetTestersAsync();
}
```

Εικόνα 4.48: Υλοποίηση του interface test result service

Η κλάση TestResultService υλοποιεί το παραπάνω interface. Αρχικά, ο constructor με τη χρήση DI κάνει inject ένα instance του AzureDevOpsHttpClientFactoryService και του ITestResultIRepository (Εικόνα 4.49).

```
private readonly HttpClient _httpClient;
private readonly ITestResultRepository _repository;

0 references | Dimitris Karakasidis, 21 hours ago | 1 author, 1 change
public TestResultService(AzureDevOpsHttpClientFactoryService httpClientFactoryService, ITestResultRepository testResultRepository)
{
    _httpClient = httpClientFactoryService.CreateHttpClient();
    _repository = testResultRepository;
}
```

Εικόνα 4.49: Ο constructor του test result service

Οι υλοποιήσεις των μεθόδων του service έχουν τα παρακάτω χαρακτηριστικά:

GetTestResultsAsync(): Παρόμοια υλοποίηση με την GetTestRunsAsync() του test run service. Διαφοροποιείται ως προς το API endpoint που καλεί και ως προς το μοντέλο και δεν περιέχει κάποια ιδιαίτερη επεξεργασία στα δεδομένα.

GetTestResultsByIdAsync(): Επίσης παρόμοια υλοποίηση και διαφορετικό API endpoint. Αυτή τη φορά δεν επιστρέφεται λίστα με δεδομένα test results, αλλά ένα συγκεκριμένο test result.

UpdateTestResultAsync():

Αυτή η μέθοδος ενημερώνει με χρήση HTTP PATCH ένα test result, στέλνοντας την ανάλυση του TAE (Εικόνα 4.50).

Διαφοροποιείται αρκετά από τις προηγούμενες GET μεθόδους, καθώς:

- Εξάγει από το responseBody και κρατάει σε μια μεταβλητή testerName το όνομα του TAE που έκανε την ανάλυση και εκκίνησε τη διαδικασία αποθήκευσης,
- Χρησιμοποιεί το repository layer για να ανακτήσει από τη βάση δεδομένων το ID του TAE με βάση το όνομά του και κρατάει το ID στη μεταβλητή testerID
- Θέτει τη μεταβλητή testerID στο property ID του OwnerDTO, που είναι εμφωλευμένο στο responseBody του UpdateTestResultDTO,
- Κάνει serialize το responseBody σε JSON μορφή με τη χρήση της μεθόδου JsonConvert.SerializeObject(),

- Στέλνει αυτό το JSON με HTTP PATCH στο API του Azure DevOps για να γίνει η ενημέρωση του test result
- Επικυρώνει την επιτυχή απόκριση από τον Azure DevOps
- Επιστρέφει true αν η απόκριση ήταν επιτυχής ή false στην αντίθετη περίπτωση

```
public async Task<bool> UpdateTestResultAsync(List<UpdateTestResultDTO> requestBody, int runId)
{
    string testerName = requestBody.FirstOrDefault().Owner.DisplayName;
    string testerId = await _repository.GetTesterIdByNameAsync(testerName);
    requestBody.FirstOrDefault().Owner.ID = testerId;

    var jsonBody = JsonConvert.SerializeObject(requestBody);
    var requestContent = new StringContent(jsonBody, Encoding.UTF8, "application/json");
    var response = await _httpClient.PatchAsync($"_apis/test/runs/{runId}/results?api-version=7.0", requestContent);
    response.EnsureSuccessStatusCode();

    return true;
}
```

Εικόνα 4.50: Υλοποίηση της μεθόδου για ενημέρωση της ανάλυσης ενός test result

GetTestersAsync():

Μέθοδος για την ανάκτηση των TAEs από τη βάση δεδομένων. Ενδιαφέρον παρουσιάζει και αυτή η υλοποίηση. Πέρα από την αλληλεπίδραση με το repository layer, γίνεται και ένα στοιχειώδες mapping, μεταξύ του Tester μοντέλου με το TesterNameDTO. Αυτή η κατασκευαστική τεχνική συχνά συμβαίνει για τους εξής λόγους:

- **Απλοποίηση των δεδομένων:** το DTO περιλαμβάνει μόνο τα πεδία που χρειάζεται το fronend (TesterName), για να μειώσει την περιττή μεταφορά δεδομένων (TesterID),
- **Αφαιρετικότητα:** αποσυνδέει το εσωτερικό μοντέλο της βάσης δεδομένων (Tester) από το API που εκτίθεται εξωτερικά, για να αποτρέψει την αποκάλυψη της δομής της βάσης,
- **Μετασχηματισμός:** επιτρέπει τη μορφοποίηση ή τον μετασχηματισμό των δεδομένων, όπου χρειάζεται, πριν αποσταλούν στο frontend,
- **Ασφάλεια:** αποτρέπει την έκθεση (expose) ευαίσθητων δεδομένων στο frontend (TesterID)

Έτσι, η μέθοδος (Εικόνα 4.51) εκτελεί τα παρακάτω βήματα:

- Χρησιμοποιεί το repository layer για να ανακτήσει τη λίστα με όλους τους TAEs του οργανισμού. Η λίστα αυτή ανακτάται ως List<Tester>,
- Κάνει το mapping ή αντιστοιχίζει το όνομα κάθε TAE από αυτή τη λίστα, σε ένα νέο TesterNameDTO και μετατρέπει αυτό το DTO σε λίστα. Τέλος, επιστρέφει τη λίστα.

```
public async Task<List<TesterNameDTO>> GetTestersAsync()
{
    var testers = await _repository.GetTestersAsync();

    return testers.Select(tester => new TesterNameDTO
    {
        Name = tester.TesterName
    })
    .ToList();
}
```

Εικόνα 4.51: Υλοποίηση της μεθόδου για ανάκτηση TAEs

Controller layer

Ο TestResultsController, είναι υπεύθυνος για τη διαχείριση των test results. Διαχειρίζεται τα HTTP αιτήματα και αναθέτει τη λογική στο ITestResultService. Υλοποιεί endpoinds για την ανάκτηση και την ενημέρωση των test results.

Ο constructor του είναι παρόμοιος με τον TestRunsController, με τη διαφορά ότι γίνεται inject το αντίστοιχο interface.

Οι μέθοδοί του παρουσιάζουν επίσης ομοιότητες, καθώς, ως επί το πλείστον, εξυπηρετεί απλά GET requests. Οι υλοποιήσεις φαίνονται παρακάτω:

GetTestResults(): ανακτά όλα τα test results για ένα συγκεκριμένο test run.

GetTestResult(): ανακτά λεπτομερείς πληροφορίες για ένα συγκεκριμένο test result, από ένα συγκεκριμένο test run.

UpdateTestResult(): ενημερώνει το test result ενός συγκεκριμένου test run, βάσει του UpdateTestResultDTO που παρέχεται στο αίτημα.

Μιας και πρόκειται για την πρώτη PATCH μέθοδο που υλοποιείται, έχει ενδιαφέρον να αναλυθεί. Η μέθοδος έχει τα παρακάτω χαρακτηριστικά (Εικόνα 4.52):

- Δέχεται ως παραμέτρους μια λίστα UpdateTestResultDTO και το ID του test run.
- Επικυρώνει την λίστα, ελέγχοντας αν είναι null. Στην περίπτωση αυτή, επιστρέφει BadRequest με status code 400 και μήνυμα «Invalid request data»,
- Μέσα σε ένα try catch block καλεί τη μέθοδο UpdateTestResultAsync() του service layer.
- Αν το service layer επιστρέψει true, τότε επιστρέφει Ok με status code 200 και μήνυμα «Test result updated successfully». Αντιθέτως, επιστρέφει Not Found με status code 404 και σχετικό μήνυμα,
- Τέλος, αν προκύψει κάποιο exception μέσα στο try catch block, επιστρέφει Internal Server Error με status code 500 και σχετικό μήνυμα.

```
[HttpPatch("{runId}/results")]
0 references | Dimitris Karakasidis, 11 days ago | 1 author, 1 change
public async Task<IActionResult> UpdateTestResult([FromBody] List<UpdateTestResultDTO> requestDto, int runId)
{
    if (requestDto is null)
        return BadRequest("Invalid request data");

    try
    {
        var result = await _testResultsService.UpdateTestResultAsync(requestDto, runId);

        if (!result)
            return NotFound($"No test result found for test run {runId} and test result {requestDto[0].ID}");

        return Ok("Test result updated successfully");
    }
    catch (Exception ex)
    {
        return StatusCode(500, $"An error occurred: {ex.Message}");
    }
}
```

Εικόνα 4.52: Υλοποίηση του update test result endpoint

GetTestersAsync(): ανακτά μια λίστα με όλους τους TAEs από τη βάση της εφαρμογής.

Models

Τα μοντέλα που χρησιμοποιούνται κατά κύριο λόγο για τις ανάγκες της σελίδας test results, αναφέρονται περιληπτικά παρακάτω.

- **TestResultDTO**: περιέχει όλες τις πληροφορίες που χρειάζεται ένας ΤΑΕ για να έχει την άμεση εικόνα του test run. Περιέχει πολλά πεδία, που είναι εμφωλευμένα DTOs, για παράδειγμα, TestPlanDTO, List<SubResultDTO> (Εικόνα 4.53).
- **TesterNameDTO:** περιέχει το όνομα ενός ΤΑΕ
- **SubResultDTO:** περιέχει τα αποτελέσματα ενός test result. Κάποια από αυτά είναι, ID, Outcome, ErrorMessage.
- **TestResultCaseDTO:** αποτελείται από βασικές πληρφορίες για ένα test case ενός test run, όπως το ID και ο τίτλος του.

```
public class TestResultDTO
{
    1 reference | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public int ID { get; set; }
    10 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public string? Comment { get; set; }
    6 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public string? ErrorMessage { get; set; }
    0 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public double DurationInMs { get; set; }
    4 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public string? Outcome { get; set; }
    4 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public required TestResultCaseDTO TestCase { get; set; }
    3 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public string? ResolutionState { get; set; }
    3 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public string? FailureType { get; set; }
    0 references | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public string ComputerName { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public int Priority { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public TestPlanDTO TestPlan { get; set; }
    2 references | Dimitris Karakasidis, 12 days ago | 1 author, 1 change
    public FailingSinceDTO? FailingSince { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public BuildDTO Build { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 2 changes
    public TesterDTO Owner { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public TesterDTO RunBy { get; set; }
    0 references | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public TesterDTO LastUpdatedBy { get; set; }
    1 reference | Dimitris Karakasidis, 9 days ago | 1 author, 1 change
    public List<SubResultDTO> SubResults { get; set; }
}
```

Εικόνα 4.53: Η μορφή του test result DTO

Ανάπτυξη controller-service για τα test attachments

Για την υλοποίηση της λειτουργικότητας των test attachments δεν χρειάστηκε να υλοποιηθεί το repository layer. Τα test attachments ανακτώνται από τον Azure DevOps, επεξεργάζονται στο service layer και αποστέλλονται στο frontend μέσω των endpoints του controller layer.

Service layer

Το service layer των test attachments διαχειρίζεται την ανάκτηση και οργάνωση των παραγόμενων αρχείων ενός test result. Παρέχει τη λογική για την ανάκτηση των δεδομένων και στη συνέχεια την ομαδοποίησή τους βάσει συμβάσεων στην ονοματοδοσία τους. Αποτελεί τον συνδετικό κρίκο μεταξύ του controller layer και του Azure DevOps.

Κάθε test iteration παράγει πολλά αρχεία κατά την εκτέλεσή του. Αυτά μπορεί να είναι exceptions και validation messages, μηνύματα επιπέδου info, screenshots της ιστοσελίδας στο σημείο που συνέβη το exception ή ακόμα και ολόκληρη η εκτέλεση του test σε μορφή βίντεο. Στο τέλος του ονόματός τους, αυτά τα αρχεία περιέχουν τον αύξοντα αριθμό του iteration. Για παράδειγμα, Exception_01.txt για το πρώτο iteration του test.

Για την παρουσίαση των test attachments στο σωστό test iteration, χρειάστηκε η επεξεργασία των ανακτηθέντων αρχείων, βάσει του ονόματός τους και η οργάνωσή τους σε μια δομή δεδομένων τύπου Dictionary.

Το interface ITestAttachmentService() περιγράφει τη μέθοδο:

Task<Dictionary<int, List<TestAttachmentDTO>>> GetGroupedAttachmentsAsync(int runId, int testResultId).

Η κλάση TestAttachmentService() υλοποιεί αυτό το interface. Η υλοποίηση της μεθόδου GetGroupedAttachmentsAsync(), είναι στο μεγαλύτερο μέρος της παρόμοια με τις μεθόδους Get, των προαναφερθέντων service layers. Η μεγάλη διαφοροποίηση βρίσκεται στην επεξεργασία των δεδομένων, για την ομαδοποίηση των test attachments. Η επεξεργασία αυτή, γίνεται σε μια private βοηθητική μέθοδο GroupAttachments() (Εικόνα 4.54), η οποία καλείται από την μέθοδο του interface (4.55).

Τα βήματα είναι τα ακόλουθα:

- Δέχεται ως παραμέτρους μια λίστα από TestAttachmentDTO
- **Επικύρωση:** ελέγχει αν η λίστα των attachments είναι null ή δεν περιέχει κανένα attachment, σε αυτήν την περίπτωση επιστρέφει ένα νέο Dictionary<int, List<TestAttachmentDTO>>,
- **Φιλτράρισμα:** αγνοεί τα attachments με κενό όνομα,
- Με χρήση regex αποσπά διψήφιους αριθμούς που βρίσκονται μετά από κάτω παύλα «_» και ακολουθούνται από τελεία («.») και είναι τοποθετημένοι στο τέλος του ονόματος του αρχείου.
- Με τον αριθμό που έχει εξαχθεί, δημιουργεί ένα group. Σε αυτό προστίθενται όλα τα αρχεία με την ίδια κατάληψη. Σε περίπτωση που δεν εξαχθεί αριθμός, δημιουργεί το group 0 για τα general attachments
- Τέλος, μετατρέπει τους αριθμούς και τα ονόματα σε ένα Dictionary και το επιστρέφει.

```
private Dictionary<int, List<TestAttachmentDTO>> GroupAttachments(GenericListModel<TestAttachmentDTO> attachments)
{
    if (attachments == null || !attachments.Value.Any())
        return new Dictionary<int, List<TestAttachmentDTO>>();

    var groupedAttachments = attachments.Value
        .Where(attachment => !string.IsNullOrEmpty(attachment.FileName))
        .GroupBy(attachment =>
        {
            var match = Regex.Match(attachment.FileName, @"_(\d{2})\.\w+$");
            if( match.Success )
                return int.Parse(match.Groups[1].Value);

            return 0;
        })
        .ToDictionary(group => group.Key, group => group.ToList());

    return groupedAttachments;
}
```

Εικόνα 4.54: Υλοποίηση της μεθόδου για την ομαδοποίηση των test attachments

```
public async Task<Dictionary<int, List<TestAttachmentDTO>>> GetGroupedAttachmentsAsync(int runId, int testResultId)
{
    HttpResponseMessage response = await _httpClient.GetAsync($"_apis/testresults/runs/{runId}/results/{testResultId}/attachments");
    response.EnsureSuccessStatusCode();
    string responseBody = await response.Content.ReadAsStringAsync();
    GenericListModel<TestAttachmentDTO>? attachments;
    try
    {
        attachments = JsonConvert.DeserializeObject<GenericListModel<TestAttachmentDTO>>(responseBody);
    }
    catch (Exception ex)
    {
        throw new Exception($"Something went wrong while deseriziling the test attachments for run {runId} and test result {testResultId}. \nException {ex.Message}");
    }

    if (attachments == null)
        throw new Exception("Test attachments is null after deserialization.");

    return GroupAttachments(attachments);
}
```

Εικόνα 4.55: Μεθόδος για την ανάκτηση των test attachments και χρήση της μεθόδου ομαδοποίησης

Controller layer

Ο AttachmentsController είναι υπεύθυνος για τη διαχείριση των εισερχόμενων HTTP αιτημάτων που σχετίζονται με τα test attachments. Επικοινωνεί με το service layer για την ανάκτηση και την ομαδοποίηση των δεδομένων, διασφαλίζοντας ότι είναι σωστά δομημένα πριν επιστραφούν στο frontend.

Η υλοποίηση του controller δεν παρουσιάζει κάποια διαφορά από τους controllers που αναλύθηκαν νωρίτερα. Χρησιμοποιεί το injected service layer για να καλέσει την GetGroupedAttachmentsAsync() και επιστρέφει Οκ με status code 200 σε περίπτωση επιτυχούς ανάκτησης, αλλιώς status code 500 «Internal Service Error» με ένα σχετικό μήνυμα.

Models

Το μοντέλο που χρησιμοποιείται για την αποθήκευση και τη μεταφορά των δεδομένων είναι το TestAttachmentDTO. (Εικόνα 4.56) Περιέχει πεδία απαραίτητα για την επεξεργασία και την παρουσίαση των δεδομένων στο frontend. Κάποια από αυτά είναι το όνομα του αρχείου, ο τύπος του, το URL στον Azure DevOps που χρησιμοποιείται για το κατέβασμα του κατά την ανάλυση.

```
public class TestAttachmentDTO
{
    2 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public DateTime CreatedDate { get; set; }
    2 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string URL { get; set; }
    0 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public int ID { get; set; }
    4 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string FileName { get; set; }
    0 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string Comment { get; set; }
    2 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public int Size { get; set; }
    0 references | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string ContentType { get; set; }
}
```

Εικόνα 4.56: Η μορφή του test attachment DTO

Ανάπτυξη controller-service για τα work items

Service layer

Το service layer των work items είναι υπεύθυνο για τη διαχείριση της επιχειρησιακής λογικής που σχετίζεται με τα work items. Χρησιμοποιεί το Azure DevOps API για την ανάκτηση των δεδομένων, τα επεξεργάζεται σε δομημένες μορφές με τη χρήση πολλών βοηθητικών μεθόδων. Είναι το μοναδικό service layer της εφαρμογής, που επεξεργάζεται και μετασχηματίζει έναν τεράστιο όγκο δεδομένων. Με τις μεθοδολογίες που ακολουθήθηκαν κατά την σχεδίαση της εφαρμογής, και τη χρήση έξυπνων και δυναμικών τρόπων για την επεξεργασία των δεδομένων, οι χρόνοι απόκρισης είναι εξαιρετικά ικανοποιητικοί και το αποτέλεσμα το βέλτιστο δυνατό.

Η ανάγκη για τις βοηθητικές μεθόδους που θα παρουσιαστούν παρακάτω, προέκυψε από τον τρόπο που είναι οργανωμένες οι πληροφορίες μέσα στο JSON της απόκρισης του Azure DevOps. Τα απαραίτητα δεδομένα για την παρουσίαση των work items, όπως test steps, test iterations και test parameters, προσφέρονται σε ένα υπερβολικά πολύπλοκο και δύσχρηστο string σε μορφή xml. Σε αυτό το xml υπάρχουν δεκάδες εμφωλεύσεις xml στοιχείων, πολλές από τις οποίες δεν βγάζουν νόημα ούτε μπορούν να επεξεργαστούν με κάποιον τρόπο, ώστε να ανακτηθούν από αυτές τα ζητούμενα δεδομένα. Για να επιτευχθεί, λοιπόν, η ανάκτηση, χρειάστηκε εκτεταμένη χρήση regex και επεξεργασίας string μεταβλητών με τις συστημικές μεθόδους της C#.

Αξίζει να σημειωθεί, πως η λογική προσέγγιση τέτοιου προβλήματος, θα ήταν με τη χρήση κάποιας XML βιβλιοθήκης για την ανάλυση του xml string. Δυστυχώς, οι ανεξήγητες διαφοροποιήσεις μεταξύ των xml strings σε φαινομενικά παρόμοια work items, αποτέλεσαν τροχοπέδη για αυτήν την προσέγγιση. Μια δεύτερη λογική προσέγγιση, που πατάει πάνω στην πρώτη, είναι η επεξεργασία αυτών των xml strings πριν την ανάλυσή τους από την XML βιβλιοθήκη. Αυτό όμως θα απαιτούσε εξαιρετικά μακροσκελείς και επιρρεπείς σε σφάλματα κώδικες, με αβέβαια αποτελέσματα, μιας και οι δομές των xml είναι απρόβλεπτες.

Η επόμενη λογική λύση, ήταν η χρήση regex. Αυτή έφερε, εν τέλει, τα επιθυμητά αποτελέσματα. Το εντυπωσιακό δε, είναι οι χρόνοι μέσα στους οποίους καταφέρνει να ανακτήσει από τον Azure Devops τα δεδομένα, να τα επεξεργαστεί και τέλος να τα στείλει στο frontend.

Παρακάτω παρουσιάζονται το interface του service layer και η κλάση που το υλοποιεί, μαζί με τις βοηθητικές μεθόδους, για την επεξεργασία των δεδομένων.

Το interface IWorkItemService περιγράφει τη μέθοδο Task<WorkItemDTO> GetWorkItemAsync(int workItemId).

Η κλάση WorkItemService υλοποιεί τη μέθοδο του interface, όπως ακριβώς υλοποιούνται οι service layer Get μέθοδοι που προαναφέρθηκαν, μέχρι το σημείο του deserialization. Για αυτήν την ιδιαίτερη περίπτωση, χρειάστηκε να δημιουργηθούν δυο μοντέλα. Αρχικά, το WorkItemRawDTO, για να κρατάει τα δεδομένα όπως έρχονται από το Azure DevOps. Έπειτα το WorkItemDTO που περιέχει εμφωλευμένα μοντέλα, στα πεδία των οποίων αποθηκεύονται οι πληρφορίες των επεξεργασμένων work items.

Πριν ξεκινήσει η επεξεργασία, καλείται η βοηθητική μέθοδος MapProcessedDTO(). Αυτή αντιστοιχίζει τα πεδία του μοντέλου WorkItemRawDTO στα αντίστοιχα WorkItemDTO. Η αντιστοίχιση όσων πεδίων είναι έτοιμα για κατανάλωση από το frontend, γίνεται άμεσα. Τέτοια είναι το ID του work item, η κατάσταση που έχει και ο ΤΑΕ που το δημιούργησε. Τα πεδία steps, parameters και data, που χρειάζονται επεξεργασία για να καταναλωθούν περνάνε το καθένα από την αντίστοιχη μέθοδο. Έτσι έχουμε τις μεθόδους:

List<WorkItemStepDTO> ParseSteps() (Εικόνα 4.57):

Επεξεργάζεται το xml του πεδίου steps. Με χρήση μοτίβου regex προσδιορίζει το είδος του step. Αν είναι δηλαδή step, ή shared step. Στην περίπτωση του step, αποθηκεύει το attribute id, ενώ στην αντίθετη περίπτωση αποθηκεύει το ref. Αφού έχει εξάγει όσα στοιχεία ταιριάζουν με αυτό το μοτίβο regex, αναλύει καθένα από τα στοιχεία. Αυτά πέφτουν σε δύο περιπτώσεις:

- Σε περίπτωση που πρόκειται για step, κρατάει σε μεταβλητή το είδος του. Αν πρόκειται δηλαδή για Action ή Validation. Στη συνέχεια, αρχικοποιεί ένα νέο WorkItemDTO με τις πληροφορίες που ανακτήθηκαν, ορίζοντας ως false το πεδίο IsSharedStep και null το πεδίο SharedStepRefId.
- Σε περίπτωση που πρόκειται για Shared Step, αρχικοποιεί ένα νέο WorkItemDTO ορίζοντας ως SharedStepRefId το στοιχείο που περιέχεται στο regex match και δίνει μια hardcoded τιμή για τον τύπο του shared step, ενώ θέτει τα υπόλοιπα πεδία null.

WorkItemDataDTO ParseData() (Εικόνα 4.58):

Επεξεργάζεται το xml του πεδίου data. Με χρήση μοτίβου regex εξάγει τα ονόματα των παραμέτρων από το tag xs:element. Στη συνέχεια, αποθηκεύει στο πεδίο ParameterName του μοντέλου WorkItemDataDTO, κάθε στοιχείο που εξήχθη, αγνοώντας τα tags NewDataSet και Table1.

Έχοντας πλέον σε μια λίστα τα ονόματα των μεταβλητών, αναζητά ξανά με regex τα στοιχεία που περικλείονται μέσα στο tag Table1. Τα αντιστοιχίζει με το όνομα κάθε παραμέτρου και τα αποθηκεύει στο πεδίο Data τύπου Dictionary<string, string>, ενός νέου μοντέλου WorkItemIterationDTO.

List<WorkItemParameterDTO> ParseParameters() (Εικόνα 4.59):

Επεξεργάζεται το xml του πεδίου parameters. Με χρήση μοτίβου regex εξάγει τα ονόματα των παραμέτρων από το tag param. Στη συνέχεια, δημιουργεί ένα νέο WorkItemParameterDTO για κάθε στοιχείο που εξήγαγε, αποθηκεύοντας το όνομα της παραμέτρου στο πεδίο Name.

Οι διαφορά των παραμέτρων της μεθόδου ParseData με τη μέθοδο ParseParameter, έγκειται στο γεγονός ότι η πρώτη περιέχει όλες τις παραμέτρους ενός work item, ακόμα και των shared steps που

χρησιμοποιεί αυτό, ενώ η δεύτερη περιέχει μόνο τις παραμέτρους του ίδιου του work item. Για την σωστή απεικόνιση των δεδομένων, χρειάζονται και τα δύο αυτά πεδία.

```
private List<WorkItemStepDTO> ParseSteps(string stepsXml)
{
    var steps = new List<WorkItemStepDTO>();

    var combinedRegex = new Regex(@"(<step id=\""(\d+)\"" type=\""(ActionStep|ValidateStep)\"">.*?<\/step>)|(<compref id=\""(\d+)\"" ref=\""(\d+)\""[^>]*>)",
        RegexOptions.Singleline);

    var matches = combinedRegex.Matches(stepsXml);
    foreach (Match match in matches)
    {
        if (match.Groups[1].Success)
        {
            var stepContent = match.Groups[1].Value;
            var stepType = match.Groups[3].Value;

            // Extract and clean action and expected result
            var action = CleanStepsText(ExtractText(stepContent, @"<parameterizedString[^>]*>(.*?)<\/parameterizedString>"));
            var expectedResult = stepType == "ValidateStep"
                ? CleanStepsText(ExtractText(stepContent, @"<parameterizedString[^>]*>(.*?)<\/parameterizedString>", 2))
                : null;

            if (string.IsNullOrWhiteSpace(action)) continue;

            steps.Add(new WorkItemStepDTO
            {
                Type = stepType,
                Action = action,
                ExpectedResult = expectedResult,
                IsSharedStep = false,
                SharedStepRefId = null
            });
        }
        else if (match.Groups[4].Success)
        {
            var sharedStepRef = int.Parse(match.Groups[6].Value);

            steps.Add(new WorkItemStepDTO
            {
                Type = "SharedStep",
                Action = null,
                ExpectedResult = null,
                IsSharedStep = true,
                SharedStepRefId = sharedStepRef
            });
        }
    }

    return steps;
}
```

Εικόνα 4.57: Υλοποίηση της μεθόδου για την εξαγωγή των work item steps

```
private WorkItemDataDTO ParseData(string dataXml)
{
    var testCaseData = new WorkItemDataDTO();

    if (string.IsNullOrWhiteSpace(dataXml))
        return testCaseData;

    var parameterNameRegex = new Regex(@"<xs:element name=(?:""|'|&quot;)(.*?)(?:""|'|&quot;)", RegexOptions.Singleline);
    var parameterNameMatches = parameterNameRegex.Matches(dataXml);

    foreach (Match match in parameterNameMatches)
    {
        if (match.Success)
        {
            var parameterName = match.Groups[1].Value;
            if (parameterName.Equals("NewDataSet") || parameterName.Equals("Table1"))
                continue;

            testCaseData.ParameterNames.Add(parameterName);
        }
    }

    var tableRegex = new Regex(@"<Table1>(.*?)<\/Table1>", RegexOptions.Singleline);
    var tableMatches = tableRegex.Matches(dataXml);

    foreach (Match tableMatch in tableMatches)
    {
        if (tableMatch.Success)
        {
            var tableContent = tableMatch.Groups[1].Value;
            var iteration = new WorkItemIterationDTO();

            foreach (var paramName in testCaseData.ParameterNames)
            {
                var paramRegex = new Regex($@"<{paramName}>(.*?)<\/{paramName}>", RegexOptions.Singleline);
                var paramMatch = paramRegex.Match(tableContent);
                iteration.Data[paramName] = paramMatch.Success ? CleanTextGeneric(paramMatch.Groups[1].Value) : string.Empty;
            }

            testCaseData.Iterations.Add(iteration);
        }
    }

    return testCaseData;
}
```

Εικόνα 4.58: Υλοποίηση της μεθόδου για την εξαγωγή των work item data

```
private List<WorkItemParameterDTO> ParseParameters(string parametersXml)
{
    var parameters = new List<WorkItemParameterDTO>();

    if (string.IsNullOrWhiteSpace(parametersXml))
        return parameters;

    var regex = new Regex(@"<param name=\""([^\""]+)\""", RegexOptions.Singleline);
    var matches = regex.Matches(parametersXml);

    foreach (Match match in matches)
    {
        if (match.Success && match.Groups[1].Success)
            parameters.Add(new WorkItemParameterDTO { Name = match.Groups[1].Value });
    }

    return parameters;
}
```

Εικόνα 4.59: Υλοποίηση της μεθόδου για την εξαγωγή των work item parameters

Για να επιτευχθούν οι προηγούμενες μέθοδοι, χρησιμοποιήθηκαν επιπλέον τρεις βοηθητικές. Απλοποίησαν την εξαγωγή κειμένου, την απαλοιφή ανεπιθύμητων συμβόλων και την αντικατάσταση συμβόλων με το αντίστοιχο HTML entity, ώστε να απεικονιστούν τα αντίστοιχα σύμβολα στην σελίδα.

Controller layer

Ο WorkItemsController είναι υπεύθυνος για τη διαχείριση των εισερχόμενων HTTP αιτημάτων που σχετίζονται με τα work items. Επικοινωνεί με το service layer για την ανάκτηση και την επεξεργασία των δεδομένων, διασφαλίζοντας ότι είναι σωστά δομημένα πριν επιστραφούν στο frontend.

Υλοποιεί τη μέθοδο GetWorkItem(). Η υλοποίηση του δεν παρουσιάζει καμία διαφοροποίηση από τον controller του test attachments και δεν χρειάζεται να αναλυθεί περαιτέρω.

Models

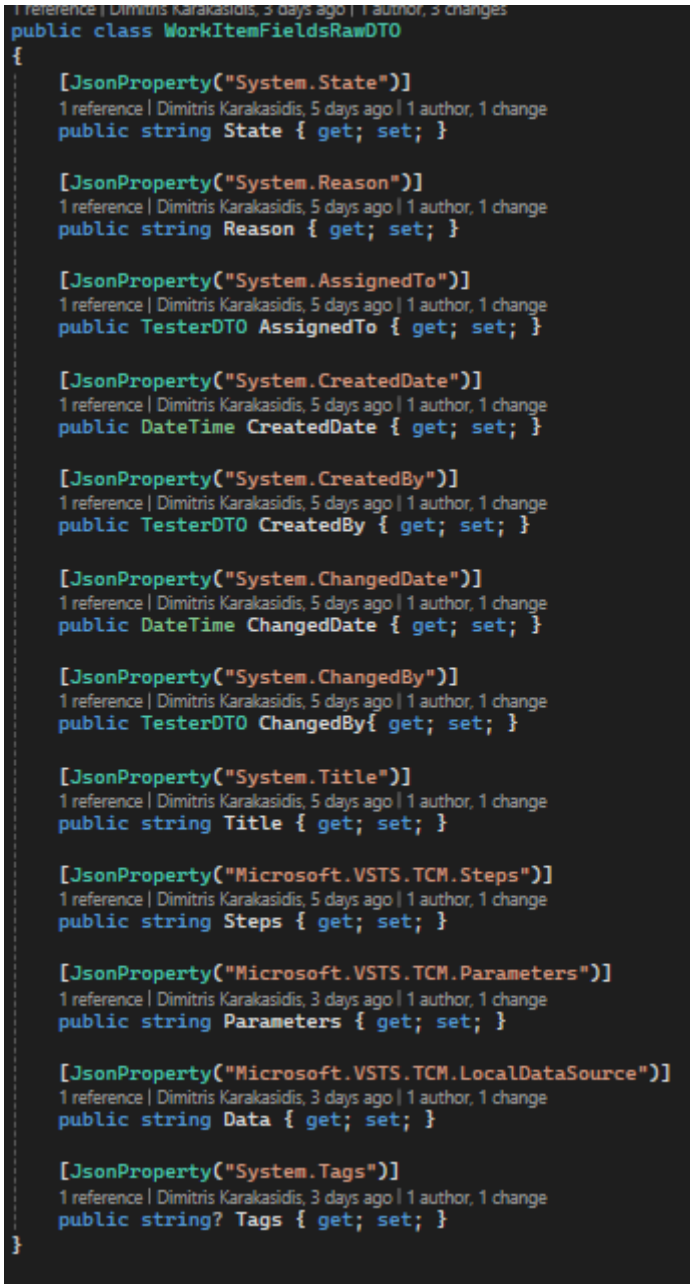
Τα μοντέλα που χρησιμοποιήθηκαν για την υλοποίηση των work items, όπως προαναφέρθηκε, έπρεπε να δημιουργηθούν εις διπλούν, με μικρές παραλλαγές σε ορισμένα σημεία, για να εξυπηρετήσουν τις ιδιαιτερότητες των αντίστοιχων πεδίων. Παραθέτονται πολύ συνοπτικά τα σημαντικά μέρη αυτών.

Τα μοντέλα για την διαχείριση των αρχικών δεδομένων, φέρουν στο όνομά τους το διακριτικό “Raw”, και είναι τα παρακάτω:

- **WorkItemRawDTO:** το μοντέλο στο οποίο γίνεται το αρχικό deserialize της απόκρισης JSON του Azure DevOps
- **WorkItemFieldsRawDTO:** εμφωλεύεται στο μοντέλο WorkItemRawDTO. Κρατάει όλες τις πληροφορίες σχετικά με το work item (Εικονα 4.60).

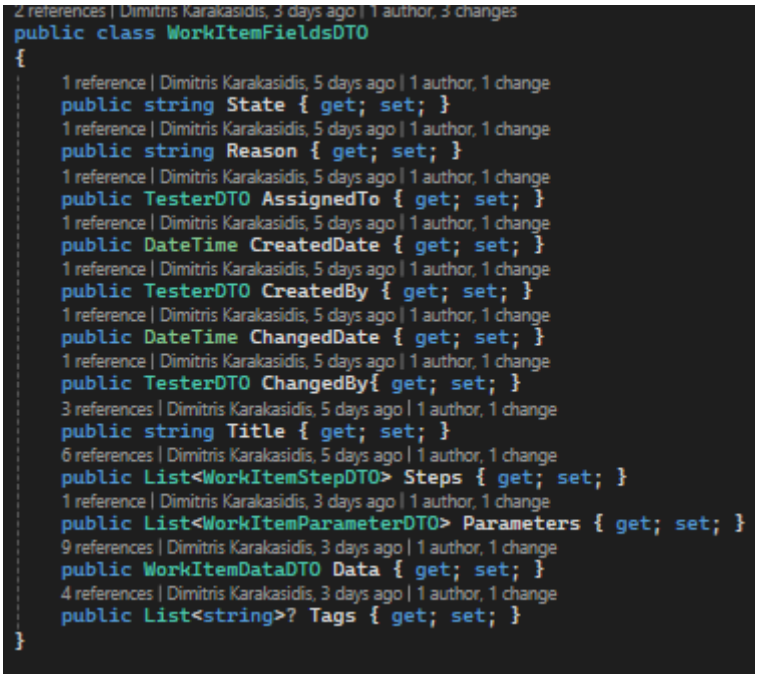
Τα μοντέλα για την διαχείριση των επεξεργασμένων τιμών, έχουν το ίδιο όνομα με τα παραπάνω, χωρίς το διακριτικό “Raw”,

- **WorkItemDTO:** κρατάει τις αντιστοιχίσεις το WorkItemRawDTO
- **WorkItemFieldsDTO:** εμφωλεύεται στο μοντέλο WorkItemDTO. Κρατάει τις αντιστοιχίσεις του WorkItemFieldsRawDTO, χωρίς τα raw steps, data, parameters. Αντί αυτών, εμφωλεύει τα παρακάτω DTOs (Εικόνα 4.61)
- **WorkItemStepDTO:** περιέχει τις πληροφορίες των βημάτων του work item
- **WorkItemParameterDTO:** περιέχει τις πληροφορίες των παραμέτρων του work item
- **WorkItemDataDTO:** περιέχει τις πληροφορίες για τα data ενός work item



```
public class WorkItemFieldsRawDTO
{
    [JsonProperty("System.State")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string State { get; set; }

    [JsonProperty("System.Reason")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string Reason { get; set; }

    [JsonProperty("System.AssignedTo")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public TesterDTO AssignedTo { get; set; }

    [JsonProperty("System.CreatedDate")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public DateTime CreatedDate { get; set; }

    [JsonProperty("System.CreatedBy")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public TesterDTO CreatedBy { get; set; }

    [JsonProperty("System.ChangedDate")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public DateTime ChangedDate { get; set; }

    [JsonProperty("System.ChangedBy")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public TesterDTO ChangedBy{ get; set; }

    [JsonProperty("System.Title")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string Title { get; set; }

    [JsonProperty("Microsoft.VSTS.TCM.Steps")]
    1 reference | Dimitris Karakasidis, 5 days ago | 1 author, 1 change
    public string Steps { get; set; }

    [JsonProperty("Microsoft.VSTS.TCM.Parameters")]
    1 reference | Dimitris Karakasidis, 3 days ago | 1 author, 1 change
    public string Parameters { get; set; }

    [JsonProperty("Microsoft.VSTS.TCM.LocalDataSource")]
    1 reference | Dimitris Karakasidis, 3 days ago | 1 author, 1 change
    public string Data { get; set; }

    [JsonProperty("System.Tags")]
    1 reference | Dimitris Karakasidis, 3 days ago | 1 author, 1 change
    public string? Tags { get; set; }
}
```

Εικόνα 4.60: Η μορφή του μοντέλου work item fields raw DTO



Εικόνα 4.61: Η μορφή του μοντέλου work item fields DTO

4.4 Επίλογος

Αυτό το κεφάλαιο παρουσίασε μια ολοκληρωμένη ανάλυση της εφαρμογής που βασίζεται σε Blazor. Ανέδειξε την ενσωμάτωση διαφόρων αρχιτεκτονικών στοιχείων για τη δημιουργία ενός αποδοτικού συστήματος διαχείρισης δεδομένων από τον Azure DevOps. Στην κέντρο αυτής της υλοποίησης βρίσκεται το πρότυπο σχεδιασμού Controller-Service-Repository, το οποίο διασφαλίζει σαφή διαχωρισμό αρμοδιοτήτων, συντηρησιμότητα και testability σε όλη την εφαρμογή.

Το controller layer λειτούργησε ως το σημείο εισόδου για τα APIs της εφαρμογής, διαχειριζόμενο HTTP αιτήματα και αναθέτοντας τη λογική επεξεργασίας στα αντίστοιχα services. Αυτή η προσέγγιση όχι μόνο απλοποίησε το routing και την επικύρωση των αιτημάτων, αλλά επίσης εξασφάλισε ότι οι controllers παρέμειναν επικεντρωμένοι στις βασικές τους αρμοδιότητες.

Το service layer λειτούργησε ως γέφυρα μεταξύ των controllers και των repositories ή των εξωτερικών APIs. Συγκέντρωσε την επιχειρησιακή λογική και διαχειρίστηκε την επεξεργασία των δεδομένων. Τα services ήταν υπεύθυνα για την κατανάλωση των αποκρίσεων του Azure DevOps, το deserialization τους, την επεξεργασία και μοντελοποίηση των δεδομένων.

Το repository layer χρησιμοποιήθηκε για την αλληλεπίδραση με την βάση δεδομένων. Για την εκτέλεση ερωτημάτων και την αποθήκευση δεδομένων χρησιμοποιήθηκε το Entity Framework Core. Επέτρεψε αποδοτικές ανακτήσεις και εγγραφές, ενώ διατήρησε ταυτόχρονα την αποσύνδεση του service layer από τη βάση δεδομένων.

Τα frontend components του Blazor ενσωματώθηκαν με μεγάλη ευκολία με το backend. Αξιοποιήθηκαν δυναμικά data-driven UI elements όπως οι πίνακες, τα dialogs και τα chips του MudTable. Τα DTOs διευκόλυναν τη μεταφορά δεδομένων μεταξύ του back και του frontend, εξασφαλίζοντας υψηλά επίπεδα συνέπειας και ασφάλειας ως προς τα ευαίσθητα δεδομένα των χρηστών.

Όλα αυτά τα στοιχεία συνέβαλαν στη δημιουργία μια στιβαρής και φιλικής προς το χρήστη εφαρμογής.

Κεφάλαιο 5ο: Συμπεράσματα και προτάσεις βελτίωσης

Η ανάπτυξη της εφαρμογής ανέδειξε τη δύναμη και την ευελιξία των σύγχρονων διαδικτυακών τεχνολογιών, όταν συνδυάζονται με ώριμα αρχιτεκτονικά πρότυπα. Ενοποιώντας τα δεδομένα των test runs, test results, test attachments και work items του Azure DevOps σε μια ενιαία πλατφόρμα, η εφαρμογή επέλυσε βασικές προκλήσεις και δυσκολίες που αντιμετωπίζουν οι Test Automation Engineers. Η υλοποίηση του προτύπου Controller-Service-Repository, η χρήση του Entity Framework Core και η χρήση DTOs, διασφάλισαν έναν καθαρό, συντηρήσιμο και επεκτάσιμο κώδικα. Παράλληλα το δυναμικό και responsive UI, που δημιουργήθηκε με τα component της βιβλιοθήκης MudBlazor, παρέχει στους χρήστες μια οπτικά ευχάριστη εμπειρία.

Το έργο πέτυχε αποτελεσματικά τους στόχους του. Προσφέρει στους TAEs αποδοτικά workflows, μειώνοντας τη χειροκίνητη αναζήτηση και βελτιώνοντας σε τεράστιο βαθμό την ορατότητα όλων των δεδομένων που σχετίζονται με τα tests. Βασικές λειτουργίες όπως η ομαδοποίηση test attachments ανά test iteration και η παρουσίαση λεπτομερών πληροφοριών των work items, επιτεύχθηκαν με ακρίβεια, προσφέροντας σημαντική εξοικονόμηση χρόνου στους χρήστες.

Μολονότι η εφαρμογή πετυχαίνει τους στόχους της, υπάρχουν πάρα πολλά περιθώρια για μελλοντικές βελτιώσεις και αναβαθμίσεις.

- **Αυθεντικοποίηση:** χρήση του Entra ID, της πλατφόρμας αυθεντικοποίησης της Microsoft, οι πληροφορίες των χρηστών θα είναι άμεσα διαθέσιμες, μέσω του Azure Portal, χωρίς την ανάγκη διατήρησής τους στην βάση της εφαρμογής.
- **Ενημερώσεις σε πραγματικό χρόνο:** χρήση του SignalR για ενημερώσεις σε πραματικό χρόνο σχετικά με test runs, test results και work items.
- **Βελτίωση διαχείρισης σφαλμάτων:** βελτίωση των μηχανισμών για τη διαχείριση σφαλμάτων και του logging τόσο στο back όσο και στο frontend, για καλύτερη εμπειρία των χρηστών και ευκολότερο debugging.
- **Εισαγωγή caching:** χρήση αποτελεσματικών τεχνικών caching για δεδομένα που χρησιμοποιούνται συχνά, για την μείωση των κλήσεων στο API και τη βελτίωση της απόδοσης.
- **Επεκτάσεις για άλλα εργαλεία CI/CD:** υποστήριξη και σε άλλες πλατφόρμες, όπως το Jenkins, GitLab CI/CD και DroneCI μέσω των REST API τους. Αφαιρώντας τη λογική των πλατφορμών από το κύριο σώμα της εφαρμογής και χτίζοντας ξεχωριστά services για την κάθε πλατφόρμα.

Με την ανάπτυξη αυτών των προτάσεων, η εφαρμογή μπορεί να εξελιχθεί σε ένα ακόμα πιο ισχυρό εργαλείο. Θα είναι σε θέση να απλοποιεί περαιτέρω τις διαδικασίες για την ανάλυση των tests και να μεγιστοποιεί την παραγωγικότητα των χρηστών.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] M. Fowler, "Microservices: a definition of this new architectural term," ThoughtWorks, 2014. [Online].

[2] ISTQB, "Certified Tester Advanced Level Test Automation Engineer Syllabus," [Online]. Available: https://www.istqb.org.

[3] R. S. Pressman, Software Engineering: A Practitioner's Approach, 8th ed. New York: McGraw-Hill, 2014.

[4] L. Bass, I. Weber, and L. Zhu, DevOps: A Software Architect's Perspective. Addison-Wesley, 2015.

[5] M. Fewster and D. Graham, Software Test Automation: Effective Use of Test Execution Tools, 2nd ed. Addison-Wesley, 2012.

[6] Microsoft, “Introduction to Blazor,” Microsoft Documentation, 2025. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0.

[7] D. Roth, “Blazor hosting models: Server-side and client-side,” Microsoft Blog, 2023. [Online]. Available: https://devblogs.microsoft.com/aspnet/blazor-server-and-webassembly/.

[8] Microsoft, “What’s new in ASP.NET Core and Blazor in .NET 8,” Microsoft Build, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/whats-new/.

[9] MudBlazor Team, “MudBlazor - Blazor UI Component Library,” MudBlazor Documentation, 2025. [Online]. Available: https://mudblazor.com/.

[10] Microsoft, “Introduction to Blazor and Component Libraries,” Microsoft Documentation, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/blazor/components.

[11] MudBlazor Team, “Themes and Customization in MudBlazor,” MudBlazor Documentation, 2025. [Online]. Available: https://mudblazor.com/features/themes.

[12] Postman Team, “Postman API Platform,” Postman Documentation, 2025. [Online]. Available: https://www.postman.com/.

[13] Microsoft, “Azure DevOps REST API Documentation,” Microsoft Docs, 2025. [Online]. Available: https://learn.microsoft.com/en-us/rest/api/azure/devops.

[14] Microsoft, “Azure Pipelines - Build, Test, and Deploy with CI/CD,” Microsoft Docs, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/.

[15] SeleniumHQ, “Selenium WebDriver,” Selenium Documentation, 2025. [Online]. Available: https://www.selenium.dev/documentation/en/webdriver/.

[16] NUnit Team, “NUnit Documentation,” NUnit.org, 2025. [Online]. Available: https://nunit.org/.

[17] Microsoft, “Integrating NUnit with Azure DevOps,” Microsoft Documentation, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/test/nunit.

[18] Microsoft, “ASP.NET Core MVC Controllers,” Microsoft Docs, 2025. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/.

[19] Microsoft, “Repository Pattern in ASP.NET Core,” Microsoft Docs, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design.

[20] Microsoft, “Dependency Injection in .NET,” Microsoft Docs, 2025. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection.