Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Προηγμένα Συστήματα Πληροφορικής»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**ΣΥΣΤΗΜΑΤΑ ΔΙΑΧΕΙΡΙΣΗΣ ΑΙΤΗΣΕΩΝ SUPPORT AND MANAGEMENT OF TICKETING SYSTEMS AND SLAS**
Ονοματεπώνυμο Φοιτητή	**Παλαιολόγος Πλάτων**
Πατρώνυμο	**Κωνσταντίνος**
Αριθμός Μητρώου	**ΜΠΣΠ14064**
Επιβλέπων	**Χρήστος Δουληγέρης, Καθηγητής**
Συνεπιβλέπων	**Δρ. Σαράντης Μητρόπουλος**

Ημερομηνία Παράδοσης **Ιούλιος 2017**

Επιτελική Σύνοψη

Η παρούσα εργασία μελετάει το πρόβλημα κατάχρησης των συστημάτων αιτήσεων υποστήριξης από τους πελάτες στον χώρο της εξωτερικής ανάθεσης έργων πληροφορικής και ταυτοποιεί σαν βασική αιτία το γεγονός ότι οι συμφωνίες μεταξύ των οργανισμών και των εργολάβων, δεν ξεκαθαρίζουν σωστά τι σημαίνουν οι προτεραιότητες που ορίζονται για τα σφάλματα, εστιάζουν κυρίως στα χρονικά πλαίσια μέσα στα οποία ο εργολάβος οφείλει να αποκρίνεται και να λύνει τυχόν σφάλματα και δεν θέτουν μια σωστή βάση επικοινωνίας μεταξύ των δύο πλευρών. Εμείς, αρχικά, προτείνουμε ένα σύνολο πληροφοριών απαραίτητο για τον ακριβέστερο ορισμό των προτεραιοτήτων των σφαλμάτων και παρουσιάζουμε πώς με την βελτίωση αυτή μπορεί να βελτιωθεί και η σχέση του εργολάβου με τους πελάτες του αλλά και η παραγωγικότητά του. Επίσης, παραθέτουμε μια εφαρμογή, την SLASUP, που αξιοποιεί τα παραπάνω δίνοντας προτεραιότητα στις αιτήσεις δυναμικά, με βάση τα στοιχεία που έχει εισαγάγει ο χρήστης σε συνδυασμό με τους ορισμούς που περιέχονται στη συμφωνία, καθώς και παρέχοντας έναν σαφή κύκλο ζωής αίτησης βασισμένο στα δικαιώματα και τις υποχρεώσεις που έχουν οι εμπλεκόμενες ομάδες.

Abstract

This dissertation discusses cases of client misuse or abuse of ticketing systems in the field of IT project outsourcing and identifies as the main cause the fact that Service Level Agreements fail to accurately describe the fault priorities presented, mostly elaborate on the corresponding response times and the policies involved and do not provide a basis for communication between the parties involved. As a solution, we provide a set of properties that can further describe the fault priorities and propose that ticketing systems should take advantage of these additional properties in order to check the information submitted. Additionally, we present SLASUP, a ticketing system that automatically assigns priorities to tickets based on other input provided by the user and the definitions provided by the Service Level Agreement. The ideas presented in the thesis were evaluated using the Balanced Scorecard technique and commercial ticketing systems as a comparison. It was concluded that improving the definitions included in a Service Level Agreement as suggested would greatly improve the cooperation between the parties involved as well as the efficiency of the support process and, in consequence, the relationship of the vendor with their clients and their profitability.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα Καθηγητή Σαράντη Μητρόπουλο για την πολύτιμη καθοδήγηση και βοήθεια του στην εκπόνηση της εργασίας καθώς και τον υπεύθυνο Καθηγητή Χρήστο Δουληγέρη για την ανάθεση της.

Επίσης, θα ήθελα να ευχαριστήσω την οικογένεια μου για την ηθική συμπαράσταση της.

Ιούλιος 2017

Παλαιολόγος Πλάτων

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ

ΠΙΝΑΚΑΣ ΠΙΝΑΚΩΝ

1 Εισαγωγή

Τα τελευταία χρόνια οι αυξημένες απαιτήσεις των πελατών από τις εταιρείες που προσφέρουν υπηρεσίες on-line έχουν διαμορφώσει τον χώρο: οι εταιρείες αυτές δεν πρέπει απλά να φροντίζουν τα προϊόντα τους να είναι καλής ποιότητας αλλά και να παρέχουν στον πελάτη τρόπους να επικοινωνούν παράπονα και βλάβες γρήγορα και εύκολα.

Τα παραπάνω είναι ιδιαίτερα σημαντικά για εταιρείες ανάπτυξης λογισμικού, όπου πελάτες τους δεν είναι οι τελικοί χρήστες των υπηρεσιών αλλά οι οργανισμοί ή οι επιχειρήσεις που τις αναθέτουν σε αυτούς (outsourcing). Στην αγορά αυτή, η διαχείριση αστοχιών και βλαβών αποτελεί πολλές φορές τον πλέον σημαντικό παράγοντα σε μια πιθανή συνεργασία, ιδιαίτερα όταν η υποστήριξη πελατών πληρώνεται ξεχωριστά από το λογισμικό. Μεγάλοι οργανισμοί που για τις υπηρεσίες που προσφέρουν διαχειρίζονται πολλά διαφορετικά έργα ανεπτυγμένα από διαφορετικές εταιρείες, όπως οργανισμοί και επιχειρήσεις τηλεπικοινωνιών, τράπεζες κλπ., βασίζονται σε ειδικά πληροφοριακά συστήματα για τη υποστήριξη των πελατών τους.

Έτσι, έχει διαμορφωθεί ειδική αγορά από επιχειρήσεις που παράγουν συστήματα υποστήριξης πελατών (helpdesk systems) ή και συστήματα γενικότερης διαχείρισης (IT Management Systems) που εκτός από υποστήριξη πελατών, προσφέρουν και εργαλεία διαχείρισης έργου (project management).

Προφανώς, στη σημερινή εποχή με τόσες διαφορετικές επιλογές σε τεχνολογίες και αρχιτεκτονικές, ο ανταγωνισμός είναι σφοδρός.

1.1 Περιγραφή του υπό μελέτη προβλήματος

Για την περιγραφή του υπό μελέτη προβλήματος θα πάρουμε σαν παράδειγμα την περίπτωση ενός μεγάλου οργανισμού τηλεπικοινωνιών, ο οποίος δεν περιορίζεται στα πλαίσια μιας χώρας. Ο οργανισμός αυτός διαχειρίζεται τα συστήματα τηλεφωνίας πολλαπλών παρόχων, δηλαδή πελατών του, των οποίων οι έδρες μπορεί να βρίσκονται σε διάφορες χώρες. Το κάθε σύστημα μπορεί να αποτελείται από επί μέρους έργα κάποια από τα οποία είναι ανεπτυγμένα από εξωτερικές εταιρείες ανάπτυξης λογισμικού (*outsourcing*) και ενδέχεται να εδρεύουν, επίσης, σε διαφορετικές χώρες.

Στο περιβάλλον του οργανισμού αυτού έρχονται σε επαφή πολλές ομάδες οι οποίες ανήκουν σε διαφορετικό επαγγελματικό, εκπαιδευτικό καθώς και κοινωνικο-πολιτικό υπόβαθρο και οι περισσότερες από αυτές τις ομάδες λαμβάνουν λιγότερο ή περισσότερο ενεργούς ρόλους στη διαδικασία της υποστήριξης των πελατών. Για παράδειγμα, οι ομάδες των παρόχων τηλεφωνίας, δηλαδή των πελατών του οργανισμού, έχουν ένα λιγότερο τεχνικό υπόβαθρο και είναι αυτές που θα αναφέρουν τυχόν σφάλματα ή αστοχίες ενώ οι ομάδες των εταιρειών ανάπτυξης λογισμικού θα έχουν, σαφώς, πιο τεχνικό υπόβαθρο και θα είναι αυτές που θα επιλύουν τα σφάλματα. Όλες οι εμπλεκόμενες ομάδες ενδέχεται να αποτελούνται από άτομα διαφορετικών εθνικοτήτων.

Επειδή σε τέτοιου βεληνεκούς συστήματα εμπλέκονται παραπάνω από μία επιχειρήσεις και πολλές ομάδες, γίνονται, συνήθως, συμφωνίες που ορίζουν τον τρόπο με τον οποίο θα

διαχειρίζονται οι βλάβες στα συστήματα αυτά, τα λεγόμενα Service Level Agreements (SLAs), τα οποία επεξηγούνται περισσότερο σε επόμενη ενότητα.

Προφανώς, τα μοντέρνα συστήματα υποστήριξης πελατών παρέχουν τρόπους ώστε να λαμβάνονται υπόψιν αυτές οι συμφωνίες. Όμως, οι συμφωνίες αυτές χρησιμοποιούνται περισσότερο ελεγκτικά και για την τήρηση των χρόνων απόκρισης από τις ομάδες υπεύθυνες για ένα έργο. Στις περισσότερες περιπτώσεις, στις συμφωνίες αυτές ορίζονται κάποιες προτεραιότητες σφαλμάτων γενικά. Για παράδειγμα, ως καίριας προτεραιότητας ορίζεται ένα σφάλμα όταν εξαιτίας του δεν είναι διαθέσιμο ένα σύστημα. Ένας τέτοιος ορισμός αρκεί στη συγκεκριμένη περίπτωση αλλά δεν αρκεί για άλλες προτεραιότητες όπως, παραδείγματος χάριν, χαμηλή, μέτρια ή υψηλή προτεραιότητα. Συνεπώς, οι προτεραιότητες αυτές δεν ξεκάθαρες αρκετά ώστε να κατανοούνται πλήρως από όλες τις εμπλεκόμενες ομάδες.

Σαν αποτέλεσμα, επειδή οι σχετικές διαδικασίες είναι πλέον τυποποιημένες μέσω των συστημάτων αυτών, δεν ελέγχεται η ακρίβεια των δεδομένων και της προτεραιότητας που δηλώνεται σε κάθε αίτηση για υποστήριξη αλλά μόνο το κατά πόσο τηρούνται οι χρόνοι απόκρισης από τις ομάδες υποστήριξης.

Αυτό σημαίνει ότι τα συστήματα παρέχουν περιορισμούς και πίεση στις ομάδες οι οποίες είναι υπεύθυνες με την επίλυση των σφαλμάτων χωρίς όμως να τους προστατεύουν από καταχρήσεις ή από παρεξηγήσεις από την πλευρά των πελατών, όταν υπάρχουν. Για παράδειγμα, αν σε μια τέτοια συμφωνία έχει οριστεί ότι μια καίρια βλάβη πρέπει να έχει διορθωθεί μέσα σε μία εργατομέρα, ένας πελάτης μπορεί να το χρησιμοποιήσει αυτό ώστε κάποιες σημαντικές μεν, όχι καίριες δε, βλάβες να επιλύονται άμεσα και, έτσι, να δημιουργήσει καθυστερήσεις και προβλήματα στη ροή των εργασιών που αφορούν σε αυτό το έργο. Αντίστοιχα, άλλος πελάτης μπορεί να βάζει λάθος προτεραιότητες στα σφάλματα ή στις αιτήσεις επειδή αυτές έχουν οριστεί γενικά και όχι με τρόπο σαφή και κατανοητό.

Ανάλογα τη φύση του συστήματος, των μεγεθών και ομάδων που εμπλέκονται και της συμφωνίας, τα παραπάνω μπορεί να μην αποτελούν προβλήματα ή να αποτελούν σπάνια φαινόμενα. Αλλά σε ένα περιβάλλον όπως αυτό του οργανισμού που έχουμε σαν παράδειγμα, όπου μιλάμε για συστήματα τηλεφωνίας τα οποία μπορεί να έχουν εκατοντάδες χιλιάδες συνδρομητές, μπορεί να προκαλούνται σοβαρές καθυστερήσεις σε άλλες απαραίτητες διαδικασίες, όπως η ανάπτυξη των έργων.

1.2 Σκοπός και στόχοι της εργασίας

Βασικός σκοπός της εργασίας αυτής είναι η επίλυση του υπό μελέτη προβλήματος. Ο σκοπός αυτός αναλύεται στους εξής επί μέρους στόχους:

- Θα προτείνουμε μια εφικτή και πραγματική λύση στο υπό μελέτη πρόβλημα.
- Θα παρέχουμε τις απαιτήσεις ενός συστήματος το οποίο θα υλοποιεί την παραπάνω λύση.
- Θα παρουσιάσουμε, επίσης, μια πρότυπη εφαρμογή, υλοποιημένη με βάση τις παραπάνω απαιτήσεις.

- Τέλος, η εργασία παρέχει ένα εναλλακτικό μοντέλο για την υλοποίηση συστημάτων υποστήριξης πελατών.

1.3 Βασικοί ορισμοί

Οι παρακάτω ορισμοί θεωρούνται απαραίτητοι για την πλήρη κατανόηση της εργασίας.

1.3.1 Service Level Agreement – SLA

Service Level Agreement (SLA – Συμφωνία Επιπέδων Υποστήριξης) είναι μία συμφωνία μεταξύ ενός παρόχου υπηρεσιών και ενός πελάτη. Η συμφωνία αυτή αποτελεί μέρος του συμβολαίου μεταξύ των δύο και θίγει ζητήματα όπως οι υπηρεσίες που παρέχονται, επιδόσεις, έλεγχος, διαχείριση βλαβών, νομικά ζητήματα, ασφάλεια και διακοπή παροχής υπηρεσιών. Ένα πολύ συχνό μέρος μια τέτοια συμφωνίας είναι ότι οι υπηρεσίες πρέπει να παρέχονται πάντα όπως έχουν συμφωνηθεί στο συμβόλαιο. [1]

Πρέπει να τονιστεί εδώ ότι μια σημαντική πτυχή ενός SLA είναι η δημιουργία μιας βάσης για την σωστή επικοινωνία μεταξύ των δύο εμπλεκόμενων πλευρών. Όπως αναφέραμε πιο πάνω, το τεχνικό υπόβαθρο διαφέρει πολλές φορές μεταξύ παρόχου και πελάτη. Έτσι, τα SLA πολλές φορές αποτυπώνονται με μη τεχνικούς όρους για σαφήνεια και σωστή κατανόηση.

Για παράδειγμα, στον χώρο των τηλεπικοινωνιών, στον οποίο αναφερθήκαμε παραπάνω, μια τέτοια συμφωνία μεταξύ του οργανισμού του παραδείγματος και μιας εταιρείας ανάπτυξης λογισμικού ορίζει τα χαρακτηριστικά της υποστήριξης πελατών όπως τον χρόνο απόκρισης σε περίπτωση βλάβης, τα χρονικά πλαίσια μέσα στα οποία μια βλάβη θα έχει αναγνωριστεί ή επιλυθεί από τις αρμόδειες ομάδες κλπ.

1.3.2 Outsourcing (Εξωτερική ανάθεση)

Εξωτερική ανάθεση ονομάζεται η πρακτική κατά την οποία ένας οργανισμός μεταφέρει την ευθύνη μιας υπηρεσίας, διαδικασίας ή και έργου σε μια εξωτερική ομάδα. Πρακτικά, αυτό σημαίνει ότι η εξωτερική ομάδα θα αναλάβει την σχεδίαση, ανάπτυξη και συντήρηση του έργου ή ένα υποσύνολο των ευθυνών αυτών ανάλογα με το τι έχει συμφωνηθεί.

1.3.3 Αίτηση υπόστηριξης (support ticket, ticket)

Μια αίτηση υποστήριξης είναι το σύνολο των πληροφοριών που παρέχει ο χρήστης μιας υπηρεσίας στον πάροχο της υπηρεσίας και περιγράφουν ένα πρόβλημα – ή μερικές φορές και απλά μια απορία – που αντιμετωπίζει ο χρήστης. Συνήθως, η πληροφορία αυτή δίδεται μέσω της διεπαφής ενός συστήματος υποστήριξης πελατών.

1.3.4 Object-relational Mapping (ORM)

Object-relational Mapping ή ORM ονομάζεται η τεχνική κατά την οποία τα δεδομένα σε μια βάση δεδομένων αναπαρίστανται ως αντικείμενα σε μια γλώσσα προγραμματισμού. Η τεχνική αυτή βοηθάει την επικοινωνία δεδομένων μεταξύ ασύμβατων συστημάτων, ιδιαίτερα σε αντικειμενοστραφή περιβάλλοντα.

Πιο χαρακτηριστικό παράδειγμα χρήσης είναι η επεξεργασία από μια εφαρμογή δεδομένων που αποθηκεύονται σε βάση. Η πληροφορία στη βάση δεδομένων, συνήθως, υπάρχει σε

απλούς τύπους δεδομένων, π.χ. κείμενο ή ακέραιοι αριθμοί ενώ στην εφαρμογή, ειδικά σε αντικειμενοστραφή περιβάλλοντα, αναπαρίστανται με αντικείμενα ή άλλες δομές που αποτελούνται από πιο απλούς τύπους δεδομένων. Τα συστήματα ORM κάνουν δυνατή αυτή την αντιστοίχιση αντικειμένου – βάσης.

1.3.5 Representational State Transfer (REST)

Το REST αποτελεί έναν τρόπο επικοινωνίας μεταξύ συστημάτων στο διαδίκτυο, κατά τον οποίο τα συστήματα αυτά χειρίζονται πόρους μέσω αναπαραστάσεων μορφής JSON, XML ή HTML και HTTP μεθόδων όπως GET, POST, PUT, DELETE κλπ. [6]

Ένα σύστημα που υλοποιεί το REST, ή αλλιώς ένα RESTful σύστημα, πρέπει να υπακούει στους εξής περιορισμούς:

- Το σύστημα θα πρέπει να είναι δομημένο με βάση το μοντέλο client-server. Αυτό είναι πολύ σημαντικό διότι, με αυτόν τον τρόπο, διαχωρίζονται οι ανάγκες της διεπαφής χρήστη από τις ανάγκες της διαχείρισης της πληροφορίας. Συνεπώς, προσδίδει ευελιξία και επεκτασιμότητα στο σύστημα αφού τα συστατικά του είναι ανεξάρτητα.
- Ο server, του παραπάνω μοντέλου, δεν θα κρατάει πληροφορία για τον client, δηλαδή το σύστημα θα είναι stateless και πληροφορία για την συνεδρία (session) θα κρατάει ο client, όταν αυτό είναι απαραίτητο.
- Οι απαντήσεις των αιτημάτων θα πρέπει να ορίζουν αν είναι δυνατό ή επιθυμητό να κρατιούνται στη κρυφή μνήμη ή όχι, ώστε να αποφευχθεί ο κίνδυνος να ληφθούν παλιά ή ανακριβή δεδομένα.
- Το σύστημα θα πρέπει να είναι οργανωμένο σε επίπεδα, πιθανώς με proxy servers ή load balancer servers μεταξύ του client και του τελικού server.
- Όλα τα REST συστήματα θα πρέπει να έχουν ομοιόμορφες διεπαφές, έτσι ώστε η αρχιτεκτονική να είναι απλή και το κάθε τμήμα της να μπορεί να εξελίσσεται ανεξάρτητα. Αυτός ο περιορισμός σημαίνει ότι:
    1. οι πόροι του συστήματος θα μπορεί να αναγνωριστούν, πιθανώς μέσω κάποιου URI, και θα είναι διαφορετικοί σαν οντότητες από τις αναπαραστάσεις που θα επιστρέφονται στον client.
    2. κάθε αίτημα του client θα πρέπει να περιέχει όλη την απαραίτητη πληροφορία για την αιτούμενη λειτουργία.
    3. όταν ο client έχει την αναπαράσταση ενός πόρου, η πληροφορία που συμπεριλαμβάνεται στην αναπαράσταση του πόρου αυτού θα είναι αρκετή για να μπορεί να τον τροποποιήσει ή να τον διαγράψει.
    4. μετά την πρώτη κλήση προς τον server, ένας client θα πρέπει να μπορεί να χρησιμοποιήσει συνδέσμους που του έχει προμηθεύσει ο server, ώστε να μην υπάρχει η ανάγκη οι σύνδεσμοι αυτοί να οριστούν στον κώδικα του client.
- Προαιρετικά, θα πρέπει ο server να μπορεί να συμπληρώνει ή να επεκτείνει λειτουργίες του client προμηθεύοντάς τον με εκτελέσιμο κώδικα, π.χ. σε μορφή servlet ή javascript.

Τα οφέλη του REST είναι πολλαπλά. Το γεγονός ότι τα συστήματα αυτά δεν έχουν καταστάσεις (stateless) και χρησιμοποιούν συγκεκριμένες, εξωτερικά ορισμένες διεπαφές σημαίνει ότι τα συστήματα αυτά μπορεί να επαναχρησιμοποιηθούν, να αναβαθμιστούν ή να διορθωθούν χωρίς να επηρεάζονται τα υπόλοιπα συστήματα. Τέλος, τέτοια συστήματα είναι και επεκτάσιμα και απλά στη σχεδίαση γιατί η βασική δομή και διεπαφές τους είναι παρόμοιες.

1.3.6 Model-View-Controller (MVC)

Model-View-Controller ονομάζεται μια αρχιτεκτονική εφαρμογών η οποία έχει σαν στόχο την ανεξαρτητοποίηση της εμφάνισης από την διαχείριση της πληροφορίας. Αυτό το επιτυγχάνει με την διαίρεση ενός συστήματος σε τρία τμήματα: το “model”, το οποίο είναι η πληροφορία και η διαδικασίες ανάκτησης και ανανέωσης της, το “view”, το οποίο ασχολείται με την αποτύπωση της πληροφορίας στον χρήστη, και τον “controller”, ο οποίος διαχειρίζεται την επικοινωνία μεταξύ των δύο.

Βασικά οφέλη της αρχιτεκτονικής αυτής είναι ότι τα τρία τμήματα ενός συστήματος μπορεί να αναπτύσσονται ταυτόχρονα, παρόμοιες λογικές να συνδυάζονται σε έναν controller, model ή view και αντίστοιχα τα τμήματα μπορούν να είναι σχετικά ανεξάρτητα μεταξύ τους. Ο διαχωρισμός αυτών των λογικών επιτρέπει την εύκολη διόρθωση σφαλμάτων ή ανάπτυξη του κάθε τμήματος χωρίς να επηρεάζονται τα υπόλοιπα. Τέλος, κάθε model μπορεί να έχει παραπάνω από ένα views.

Η αρχιτεκτονική αυτή, όμως, απαιτεί γνώση πολλαπλών τεχνολογιών από τους προγραμματιστές. Επίσης, ένα μειονέκτημα της αρχιτεκτονικής αυτής είναι και το γεγονός ότι για έναν δεδομένο πόρο μπορεί να χρειάζεται να συντηρούνται παραπάνω από ένας τρόποι αναπαράστασης π.χ. κλάσεις Java, αρχεία json κλπ.

1.3.7 Inversion of Control (IoC)

Στον παραδοσιακό προγραμματισμό, ο προσαρμοσμένος κώδικας είναι αυτός που ελέγχει τη ροή της εφαρμογής, εκτελώντας προσαρμοσμένο ή έτοιμο (από βιβλιοθήκες) κώδικα για τις εκάστοτε ανάγκες της. Το Inversion of control (IoC) είναι μια προγραμματιστική αρχή κατά την οποία βιβλιοθήκες έτοιμου κώδικα, συνήθως ενός framework, ελέγχιυν τη ροή της εφαρμογής χρησιμοποιώντας προσαρμοσμένο κώδικα που έχει υλοποιηθεί για συγκεκριμένες εργασίες.

Συνήθως, σε εφαρμογές που υλοποιούνται με βάση την αρχή αυτή, χρησιμοποιούνται frameworks που προσφέρουν τη βασική ροή του εκάστοτε τύπου εφαρμογής και οι προγραμματιστές υλοποιούν κώδικα που εξυπηρετεί πιο συγκεκριμένους σκοπούς και θα καλείται σε συγκεκριμένες περιπτώσεις. Το Spring Framework που αναφέρουμε σε επόμενη ενότητα ακολουθεί τη λογική αυτή για εφαρμογές ιστού.

Το IoC έχει τους εξής στόχους:

- Να αποσυνδέσει την εκτέλεση μιας διεργασίας από την ανάπτυξή της
- Να εστιάζει η μονάδα στο σκοπό για τον οποίο σχεδιάζεται
- Να μην εξαρτάται η ανάπτυξη κάθε μονάδας από τον τρόπο με τον οποίο λειτουργούν άλλες
- Να είναι πιο εύκολη η αντικατάσταση ή η διόρθωση μια μονάδας

1.3.8 Transaction

Συναλλαγή (transaction) ονομάζεται μια ακολουθία εντολών ή κλήσεων μεθόδων οι οποίες αντιμετωπίζονται ως μία διαδικασία και ολοκληρώνεται είτε με δέσμευση (commit) των αλλαγών ή αποτελεσμάτων είτε με αναστροφή των αλλαγών (rollback). Ως συναλλαγές συνήθως ορίζονται διαδικασίες οι οποίες αν διακοπούν λόγω κάποιου σφάλματος, οι αλλαγές που έχουν γίνει μέχρι εκείνο το σημείο μπορεί να προκαλέσουν πρόβλημα στην ακεραιότητα των δεδομένων.

Στην περίπτωση του Spring, μέθοδοι οι οποίες χρειάζονται δεδομένα ή πρόσβαση στη βάση πρέπει πάντα να εκτελούνται στα πλαίσια μιας συναλλαγής.

1.4 Παραδοτέα της εργασίας

Στην ενότητα αυτή αναφέρονται συνοπτικά τα παραδοτέα της εργασίας:

1. Το έντυπο κείμενο της διπλωματικής εργασίας.
2. Η πρότυπη εφαρμογή (πηγαίος κώδικας και το αρχείο εφαρμογής .war).
3. Η παρουσίαση της εργασίας σε μορφή PowerPoint.

1.5 Δομή της εργασίας

Η εργασία χωρίζεται σε επτά ενότητες. Η πρώτη ενότητα είναι η εισαγωγή, στην οποία θέτουμε τον χώρο και το υπό μελέτη πρόβλημα και περιγράφουμε κάποιες έννοιες που θα χρειαστούν για την κατανόηση της εργασίας.
Στην δεύτερη ενότητα, παραθέτουμε την προτεινόμενη λύση καθώς και τις απαιτήσεις με βάση τις οποίες αναπτύξαμε την εφαρμογή και, τέλος, κάποιες έννοιες που προκύψανε και χρησιμοποιούνται.
Στην τρίτη ενότητα περιγράφουμε την δομή της εφαρμογής που αναπτύξαμε, τις τεχνολογίες που χρησιμοποιήσαμε καθώς και άλλες σχετικές αποφάσεις που πήραμε.
Στην τέταρτη ενότητα παρουσιάζουμε την εφαρμογή και εξηγούμε τις διάφορες λειτουργίες που υποστηρίζονται.
Στην πέμπτη ενότητα αξιολογούμε τόσο την λύση και την επίδραση της σε έναν δεδομένο οργανισμό καθώς και την εφαρμογή έναντι σε αντίστοιχες εφαρμογές του εμπορίου.
Τέλος, στην έκτη ενότητα παρέχουμε τα συμπεράσματα της εργασίες και στην έβδομη τη βιβλιογραφία που αξιοποιήθηκε.

2 Η προτεινόμενη λύση

Τα σύγχρονα συστήματα υποστήριξης πελατών δίνουν περισσότερη βάση στα τμήματα των SLA που αναφέρονται σε χρόνους απόκρισης και διευθέτησης βλαβών. Η πρακτική αυτή δεν είναι ούτε λανθασμένη ούτε παράλογη, δεδομένου του ότι σε συστήματα που παρέχουν υπηρεσίες σε πραγματικό χρόνο η διαθεσιμότητα των υπηρεσιών αυτών είναι η πρώτη προτεραιότητα. Από την άλλη, όμως, ενέχει τον κίνδυνο να δημιουργήσει το πρόβλημα το οποίο εξετάζουμε. Επίσης, τα SLA κατά κύριο λόγο ορίζουν τα χρονικά αυτά περιθώρια καθώς και τις πολιτικές σε περίπτωση αδυναμίας να τηρηθούν.

Η λύση που προτείνουμε στην εργασία αυτή έχει δύο πτυχές. Πρώτον, τα SLA να ορίζουν με μεγαλύτερη ακρίβεια τις προτεραιότητες των σφαλμάτων εισάγωντας ένα πλαίσιο ορισμών ή μεγεθών τα οποία να είναι πιο συγκεκριμένα και, πιθανώς, μετρήσιμα.

Δεύτερον, τα συστήματα διαχείρισης αιτήσεων υποστήριξης να λαμβάνουν υπόψην τη νέα πληροφορία. Έτσι, θα γίνεται πιο εύκολος ο έλεγχος της πληροφορίας που εισάγεται ο χρήστης και κατά πόσο συμφωνεί με το SLA.

2.1 Ιδανικό SLA

Η λύση που θα προτείνουμε σε αυτό το κεφάλαιο βασίζεται σε ένα μεγάλο βαθμό στο πόσο αναλυτικό και πλήρες είναι το εκάστοτε SLA. Όπως αναφέραμε στην Ενότητα 1.3.1, το SLA είναι ένα βασικό έγγραφο σε οποιοδήποτε συμβόλαιο που περιγράφει με λιγότερο τεχνικούς όρους διάφορες δεσμεύσεις, δικαώματα και υποχρεώσεις μεταξύ των εμπλεκόμενων ατόμων ή ομάδων όσον αφορά σε μια υπηρεσία.

Όσον αφορά στην αντιμετώπιση σφαλμάτων, το πιο βασικό κομμάτι ενός SLA πρέπει να είναι ο ορισμός όρων όπως σφάλμα/βλάβη και καινούργιες απαιτήσεις. Συνήθως, ως σφάλμα αναφέρεται η αποτυχία της υπηρεσίας να λειτουργήσει όπως έχει υποσχεθεί ο πάροχος.

Έπειτα, στο SLA είναι απαραίτητο να αναφέρονται ορισμοί για τη βαρύτητα ενός σφάλματος. Για παράδειγμα, το να μην φαίνεται το όνομα πατρός του πελάτη σε μια οθόνη με πληροφορίες προς τον πελάτη δεν είναι ένα σφάλμα πολύ μεγάλης βαρύτητας, δεδομένου ότι είναι πληροφορία που ο πελάτης την ξέρει καλά. Η ίδια, όμως, αστοχία σε έναν κατάλογο ονομάτων για τον χρήστη μια εφαρμογής διαχείρισης πελατών μπορεί να δημιουργήσει μεγάλο πρόβλημα αν υπάρχουν παρόμοια ή πανομοιότυπα ονόματα.

Ανάλογα με τη βαρύτητα κάθε σφάλματος, ορίζονται και διαφορετικοί χρόνοι αντιμετώπισης. Πολλές φορές, ορίζονται διάφοροι τέτοιοι χρόνοι: ο χρόνος αναγνώρισης, δηλαδή μέσα σε πόσο χρόνο πρέπει ο πάροχος να αναγνωρίσει το σφάλμα σαν σφάλμα ή να απαντήσει αναλόγως στην περίπτωση που δεν αποτελεί σφάλμα, ο χρόνος αντιμετώπισης, δηλαδή μέσα σε πόσο χρόνο οφείλει ο πάροχος να έχει δώσει λύση στο σφάλμα, αν είναι όντως σφάλμα, ο χρόνος πρώτης απάντησης, που πολλές φορές ταυτίζεται με τον χρόνο αναγνώρισης κ.ο.κ.

Παρόλα αυτά, για να οριστεί η βαρύτητα ενός σφάλματος, δεν αρκεί πάντα μια απλή περιγραφή του τι σημαίνει η κάθε κατηγορία που προκύπτει. Στο παράδειγμα του οργανισμού

τηλεπικοινωνιών, ο οργανισμός αυτός έχει υπογράψει συμβόλαιο με μία εταιρεία πληροφορικής για ένα από τα συστήματα (εργολαβία). Στα πλαίσια του συμβολαίου αυτού, έχει υπογραφεί ένα αντίστοιχο SLA. Ας πούμε ότι στο SLA αυτό έχουν αναγνωριστεί πέντε κατηγορίες βαρύτητας σφαλμάτων (προτεραιότητες): πληροφοριακή (ο πελάτης έχει μια απορία σχετικά με την υπηρεσία), χαμηλή, μέτρια, υψηλή και καίρια. Ας πάρουμε την απλή περίπτωση όπου έχουν οριστεί μόνο χρόνοι αντιμετώπισης και υποθέτουμε ότι σε περίπτωση πληροφοριακού «σφάλματος», ο πάροχος έχει πολύ χρόνο να απαντήσει, αντίστοιχα σε περίπτωση σφάλματος χαμηλής βαρύτητας, ενώ σφάλματα υψηλής βαρύτητας πρέπει να έχουν αντιμετωπιστεί μέσα σε μία το πολύ εργατομέρα.

Στο SLA αυτό θα πρέπει να έχουν οριστεί με ακριβή τρόπο το τι σημαίνουν οι παραπάνω κατηγορίες. Για παράδειγμα, μπορεί να οριστούν τύποι σφάλματος όπως τεχνικό σφάλμα, έλλειψη κάποιου χαρακτηριστικού, σφάλμα λογικής κ.ο.κ. κάθε ένας εκ των οποίων θα συνδέεται με μία ή παραπάνω προτεραιότητες ή θα έχει δική του βαρύτητα. Π.χ. ένα τεχνικό σφάλμα είναι σοβαρό αλλά όχι τόσο σοβαρό όσο η έλλειψη κάποιου χαρακτηριστικού.

Όμως, ένας τύπος σφάλματος από μόνος του δεν μπορεί να καθορίσει αν ένα σφάλμα είναι καίριο ή όχι. Ένα τεχνικό σφάλμα μπορεί να επηρεάζει έναν τελικό πελάτη ή και μια ολόκληρη κατηγορία. Στον χώρο των τηλεπικοινωνιών, ένα σφάλμα μπορεί να επηρεάσει τους συνδρομητές μιας ολόκληρης χώρας. Οπότε, προκύπτει σαν κριτήριο και ο αριθμός «κρουσμάτων» ή αριθμός περιπτώσεων που επηρεάστηκαν από το συγκεκριμένο σφάλμα. Είναι λογικό να υποθέσουμε ότι η βαρύτητα ενός σφάλματος είναι ανάλογη του μεγέθους της ζημιάς που έχει προκληθεί.

Ο αριθμός των «κρουσμάτων» είναι ένα βασικό κριτήριο για την προτεραιότητα. Ένα σφάλμα που επηρεάζει έναν μόνο συνδρομητή δεν θα θεωρείται τόσο σημαντικό όσο ένα σφάλμα που επηρεάζει εκατοντάδες. Εκτός αν ο συγκεκριμένος συνδρομητής είναι ένας μεγάλος πελάτης π.χ. μια πολυεθνική εταιρεία ή ένας μεγάλος οργανισμός. Είναι πολύ πιθανό, για τέτοιες περιπτώσεις πελατών, να πρέπει να λύνονται τα σφάλματα πιο γρήγορα για στρατηγικούς λόγους ή καθαρά και μόνο για να έχει ο πελάτης μια καλή εικόνα των υπηρεσιών που προσφέρονται.

Για αυτό το λόγο, είναι σημαντικό να υπάρχει και κάποια κατηγοριοποίηση των πελατών, π.χ. φυσικό πρόσωπο, νομικό πρόσωπο, επιχείρηση (πιθανώς και με μεγέθη), κρατικός οργανισμός κ.ο.κ. Αναλόγως με τη φύση της υπηρεσίας, κάθε τέτοια κατηγορία θα συνδέεται και με μία ή παραπάνω προτεραιότητες.

Ορίζοντας τα παραπάνω μεγέθη και προσδίδοντάς τους σημαντικότητα, είτε μέσω σύνδεσης με τις προτεραιότητες που έχουν οριστεί είτε με αντίστοιχες βαρύτητες, θεσπίζεται ένα πιο ακριβές πλαίσιο επικοινωνίας μεταξύ των εμπλεκόμενων ομάδων και οι προτεραιότητες πλέον μεταφράζονται σε χαρακτηριστικά μετρήσιμα και κατανοητά.

2.2 Ανάλυση απαιτήσεων

Με βάση τα παραπάνω, θέλουμε να πραγματοποιήσουμε ένα Σύστημα Διαχείρισης Αιτήσεων Υποστήριξης το οποίο να εκμεταλλεύεται ένα SLA όπως το περιγράψαμε στην προηγούμενη ενότητα.

Ένα σύστημα αιτήσεων υποστήριξης πρέπει να:

- Δίνει την δυνατότητα στους χρήστες να μπορούν να κάνουν αιτήσεις υποστήριξης μέσω μιας διεπαφής όπου θα μπορούν να συμπληρώνουν όλες τις απαραίτητες πληροφορίες καθώς και να προσθέτουν σχόλια αλλά και αρχεία.
- Υποστηρίζει έναν κύκλο ζωής μιας αίτησης υποστήριξης αποδοτικό και ευθύ, ώστε η διαδικασία υποστήριξης να είναι όσο γίνεται πιο απλή και γρήγορη.
- Δίνει τη δυνατότητα σε όλες τις εμπλεκόμενες ομάδες να αλλάζουν στοιχεία των αιτήσεων, ακολουθώντας πάντα τον κύκλο ζωής τους.
- Επιτρέπει στον εργαζόμενο να μπορεί να βλέπει γρήγορα και εύκολα ό,τι πληροφορία χρειάζεται για την αίτηση υποστήριξης που έχει αναλάβει.
- Παρέχει διεπαφές για αναζήτηση αιτήσεων υποστήριξης.

Πέρα από τα παραπάνω, το σύστημα που θα παρουσιαστεί θα πρέπει να υλοποιεί ένα SLA με τρόπο τέτοιο ώστε:

- Να διευκολύνει την διαδικασία υποβολής και εξέλιξης μιας αίτησης υποστήριξης.
- Να τηρούνται οι όροι που έχουν συζητηθεί στο SLA.

Τέλος, τέτοια συστήματα συνήθως είναι βοηθητικά για μία επιχείρηση ή έναν οργανισμό. Με εξαίρεση την περίπτωση που είναι τμήμα μιας σουίτας λογισμικού ή του in-house πληροφοριακού συστήματος του οργανισμού, θέλουμε ένα τέτοιο σύστημα να μην είναι απαιτητικό τεχνολογικά και να μην προκαλεί φόρτο στον server που θα λειτουργεί.

Από τα παραπάνω καταλήγουμε σε πιο τεχνικές απαιτήσεις της εφαρμογής, που περιγράφονται στις παρακάτω επιμέρους ενότητες.

2.2.1 Τεχνικές απαιτήσεις

Όπως προαναφέρθηκε, ένα σύστημα διαχείρισης αιτήσεων υποστήριξης είναι κατά βάση βοηθητικό. Είτε θα είναι ένα από τα πολλά συστήματα που θα χρησιμοποιεί μια επιχείρηση ή ένας οργανισμός είτε θα είναι κομμάτι ενός ευρύτερου συστήματος. Σε κάθε μία από τις περιπτώσεις αυτές, σαν εφαρμογή θα πρέπει να έχει μικρό φυσικό μέγεθος και να είναι δυναμικά προσαρμόσιμη.

Σημαντικό χαρακτηριστικό και απαίτηση, επίσης, αποτελεί η απλότητα. Οι άνθρωποι που θα χρησιμοποιούν την εφαρμογή, θα τη χρησιμοποιούν για να τους απλουστεύσει όλα ή μερικά από τα καθήκοντά τους. Συνεπώς, η εφαρμογή θα πρέπει να είναι:

- εύκολη στην εκμάθηση, ώστε οι νέοι χρήστες να μπορούν να την χειρίζονται άμεσα.
- απλή στη χρήση.

- ευφυής, ώστε να απαλλάσσει τον χρήστη από την εισαγωγή παραπάνω πληροφορίας που δεν χρειάζεται.

Ουσιαστικά, η εφαρμογή θα πρέπει να περιμένει από τον χρήστη μόνο τα απαραίτητα προκειμένου να επιτελέσει μια διεργασία, δίνοντας, όμως, τη δυνατότητα για λεπτομέρειες, σε περίπτωση που ο χρήστης επιθυμεί να τις δώσει.

2.2.2 Απαιτήσεις που αφορούν στους χρήστες

Οι χρήστες που θα χειρίζονται την εφαρμογή ενδέχεται να προέρχονται από διαφορετικούς οργανισμούς/επιχειρήσεις. Συνεπώς, η εφαρμογή θα πρέπει να κρατάει ανεξάρτητη και αποκλειστική πληροφορία για τους χρήστες. Συνεπώς, επιβάλλεται να παρέχει διεπαφές και εργαλεία διαχείρισης χρηστών. Θα πρέπει να υποστηρίζονται οι παρακάτω λειτουργίες:

- Δημιουργία νέου χρήστη.
- Τροποποίηση στοιχείων χρήστη.
- Διαγραφή χρήστη, η οποία, όμως, να είναι αντιστρέψιμη.

Για λόγους ασφαλείας, θα οριστούν δύο κατηγορίες χρηστών: οι απλοί χρήστες και οι διαχειριστές. Οι απλοί χρήστες θα μπορούν μόνο να τροποποιήσουν συγκεκριμένα προσωπικά χαρακτηριστικά ενώ οι διαχειριστές θα έχουν πλήρη δικαιώματα.

2.2.3 Απαιτήσεις που αφορούν στις αιτήσεις εξυπηρέτησης

Η εφαρμογή θα πρέπει να υποστηρίζει έναν κύκλο ζωής αίτησης εξυπηρέτησης με συγκεκριμένα και λογικά στάδια, όπως:

- Καινούργια, το αρχικό στάδιο μιας αίτησης εξυπηρέτησης
- Υπό επεξεργασία, όταν η αίτηση έχει ανατεθεί στις αντίστοιχες ομάδες και οι απαραίτητες ενέργειες είναι υπό εξέλιξη.
- Απαντημένη, όταν έχει προταθεί λύση ή έχει δωθεί απάντηση από την υπεύθυνη ομάδα.
- Ολοκληρωμένη, τελική κατάσταση όταν το πρόβλημα έχει επιλυθεί.

Ανάλογα με την ομάδα στην οποία ανήκει ο χρήστης, θα μπορεί να επηρεάσει με διαφορετικό τρόπο τον κύκλο ζωής μιας αίτησης εξυπηρέτησης:

- Ένας χρήστης που αναφέρει σφάλματα, παραδείγματος χάρη, μπορεί να εισάγει αιτήσεις εξυπηρέτησης και να αποδεχτεί ή να απορρίψει την λύση που έχει προταθεί από άλλους χρήστες, σε περίπτωση αιτήσεων που έχει εισάγει ο ίδιος.
- Ένας χρήστης υπεύθυνος με την επίλυση σφαλμάτων θα μπορεί να αναλάβει μια αίτηση και να προτείνει λύση ή να αναφέρει ότι το σφάλμα επιλύθηκε.
- Τέλος, όλοι οι χρήστες θα μπορούν να προσθέτουν σχόλια σε οποιοδήποτε στάδιο του κύκλου ζωής.

Αν και δεν είναι απαραίτητο, είναι χρήσιμο να υπάρχει και μια τρίτη κατηγορία χρηστών οι οποίοι θα είναι υπεύθυνοι για την διόρθωση λαθών, την μεταφορά και ανάθεση αιτήσεων σε

περιπτώσεις που υπάρχουν παραπάνω από μία ομάδες IT και τη γενικότερη υποστήριξη της διαδικασίας σε τέτοια σενάρια. Οι χρήστες αυτοί θα μπορούν να αλλάζουν όλα τα στοιχεία μιας αίτησης οποιαδήποτε στιγμή και θα έχουν ρόλο υποστηρικτικό και διορθωτικό.

Η εφαρμογή θα πρέπει να παρέχει διεπαφές για αναζήτηση αιτήσεων με διαφορετικά κριτήρια, όπως:

- Στήλη με τις πιο πρόσφατες αιτήσεις
- Στήλη με καινούργιες αιτήσεις που δεν έχουν ακόμα ανατεθεί
- Στήλη με τις αιτήσεις που έχουν ανατεθεί σε συγκεκριμένο χρήστη
- Ξεχωριστή διεπαφή με πολυκριτηριακή αναζήτηση

Είναι προφανές ότι η εφαρμογή θα πρέπει να παρέχει και μια διεπαφή για την υποβολή νέω αιτήσεων.

2.2.4 Απαιτήσεις που αφορούν στο χειρισμό των SLA

Ο στόχος μας είναι να δημιουργήσουμε μια δυναμική εφαρμογή διαχείρισης αιτήσεων. Παρόλα αυτά, θέλουμε η αίτηση αυτή να αντικατοπτρίζει πιο συνολικά ένα δεδομένο SLA. Για αυτό το λόγο, αντί να θέτει ο χρήστης μια προτεραιότητα για την αίτηση που εισάγει, θα την θέτει το σύστημα, αυτόματα, με βάση τα άλλα στοιχεία που δίδονται.

Η εφαρμογή, συνεπώς, θα πρέπει να μπορεί να λάβει υπόψην της άρθρα του SLA που αφορούν σε:

- Χρόνους, ανά προτεραιότητα αίτησης, μέσα στους οποίους θα πρέπει να έχει λυθεί το πρόβλημα ή το σφάλμα ή να έχει δωθεί μια τελική απάντηση.
- Προτεραιότητες αιτήσεων.
- Τύπους σφάλματος.
- Τύπους πελατών καθώς και μεμονομένους πελάτες.
- Αριθμό «κρουσμάτων».

Από τα παραπάνω στοιχεία, τα τρία τελευταία (τύποι σφάλματος, τύποι πελατών και αριθμός «κρουσμάτων»), συνδέονται με την προτεραιότητα αίτησης. Ο χρήστης θα επιλέγει αυτά τα στοιχεία και η εφαρμογή θα παράγει αυτόματα την προτεραιότητα της αίτησης με βάση τους κανόνες από το SLA.

2.3 Έννοιες

Λαμβάνοντας υπόψην τις παραπάνω απαιτήσεις, ένα Σύστημα Διαχείρισης Αιτήσεων Υποστήριξης θα βασίζει τη λειτουργία του πάνω σε έννοιες οι οποίες θα έχουν οριστεί μέσα στο SLA ή θα εξάγωνται από την πληροφορία που περιέχεται σε αυτό.

Στην ενότητα αυτή θα ορίσουμε τις έννοιες που θεωρούμε απαραίτητες για ένα τέτοιο σύστημα. Στις έννοιες αυτές θα βασιστεί, επίσης, και η εφαρμογή που θα υλοποιήσουμε σαν παράδειγμα.

Χρήστες και δικαιώματα

Κεντρικό σημείο της εφαρμογής είναι οι χρήστες και τα δικαιώματα που έχει ο καθένας. Επειδή η εφαρμογή αναφέρεται σε περιβάλλον επιχείρησης, άρα υπάρχει κάποια ιεραρχία, ήταν απαραίτητο να υπάρχουν χρήστες με πλήρη δικαιώματα και χρήστες με ελαττωμένα δικαιώματα. Για τον λόγο αυτόν, στην εφαρμογή κάποιοι χρήστες είναι διαχειριστές (administrators) και κάποιοι όχι.

Ένας διαχειριστής έχει το δικαίωμα να κάνει τα εξής:

- Να επεξεργάζεται κάποια από τα στοιχεία του.
- Να προσθέτει νέους χρήστες.
- Να επεξεργάζεται τα στοιχεία άλλων χρηστών.
- Να ενεργοποιεί/απενεργοποιεί άλλους χρήστες.

Οι υπόλοιποι χρήστες θα μπορούν μόνο να επεξεργάζονται κάποια από τα στοιχεία τους.

Οι χρήστες της εφαρμογής, επίσης, χωρίζονται σε ομάδες ανά έργο (project): η ομάδα που θα αναφέρει σφάλματα (ticket submitters), η ομάδα IT υπεύθυνη για την διερεύνηση και διόρθωση λαθών (it) και, τέλος, η ομάδα υπεύθυνη για την επικοινωνία των δύο άλλων ομάδων και την διαχείριση των αιτήσεων (support).

Οι *ticket submitters* μπορούν να υποβάλλουν νέες αιτήσεις, να προωθήσουν αιτήσεις σε άλλες ομάδες υπό κάποιες προυποθέσεις και να αποδεχτούν ή να απορρίψουν λύσεις που μπορεί να προτείνουν άλλες ομάδες.

Οι χρήστες *IT* μπορούν να προωθούν αιτήσεις σε άλλες ομάδες υπό κάποιες προυποθέσεις και να προτείνουν λύσεις.

Τέλος, οι χρήστες *support* μπορούν να προωθούν αιτήσεις σε όλες τις ομάδες πιο ελεύθερα και να μεταφέρουν την αίτηση από οποιοδήποτε σημείο του κύκλου ζωής της σε οποιοδήποτε σημείο. Επίσης, είναι η μόνη ομάδα που θα μπορεί να αλλάξει την προτεραιότητα της αίτησης σε περίπτωση λάθους από τη σύστημα ή άλλης εξέλιξης.

Αιτήσεις

Επόμενο βασικό στοιχείο της εφαρμογής είναι οι αιτήσεις υποστήριξης (support tickets) καθώς και ο κύκλος ζωής τους.

Προτού περιγράψουμε την αίτηση ως οντότητα, θα χρειαστεί να περιγράψουμε κάποια μεγέθη. Τα μεγέθη που θα περιγραφούν είναι μέρη του SLA που προτείναμε στην ενότητα 2.1.

Πιο βασικό στοιχείο της εφαρμογής είναι η προτεραιότητα κάθε αίτησης και παράγεται δυναμικά από την εφαρμογή. Μόνο χρήστες που ανήκουν στις ομάδες *support* μπορούν να επηρεάσουν αυτήν την πληροφορία σε μια αίτηση. Όλα τα υπόλοιπα μεγέθη που ορίζονται παρακάτω επηρεάζουν την προτεραιότητα. Επίσης, κάθε προτεραιότητα προσθέτει και έναν χρονικό περιορισμό για την ολοκλήρωση της αίτησης ή αλλιώς «κλείσιμο» και, συνεπώς, τη λύση του σφάλματος ή μια τελική απάντηση.

Ορίζονται οι εξής προτεραιότητες, σε σειρά σημαντικότητας:

1. Πληροφοριακή (Informative). Πρόκεται για απλή απορία και πρέπει να έχει «κλείσει» μέσα σε πέντε μέρες.
2. Χαμηλή (Low). Πρόκεται για σφάλμα με μικρό αντίκτυπο. Η αίτηση πρέπει να έχει «κλείσει» μέσα σε πέντε μέρες.
3. Μεσαία (Medium). Το σφάλμα δεν έχει σοβαρό αντίκτυπο. Η αίτηση πρέπει να έχει «κλείσει» σε τρεις μέρες.
4. Υψηλή (High). Το σφάλμα έχει σοβαρό αντίκτυπο στους πελάτες. Η αίτηση πρέπει να έχει κλείσει σε δύο μέρες.
5. Ύψιστη (Critical). Θα χρησιμοποιείται για πολύ σημαντικούς πελάτες ή για περιπτώσεις μη διαθεσιμότητας ολόκληρων υπηρεσιών. Η αίτηση πρέπει να έχει κλείσει σε μία μέρα.

Στην εφαρμογή χρησιμοποιούμε οκτώ κατηγορίες σφαλμάτων, εκ των οποίων οι πέντε μπορεί να χρησιμοποιηθούν κατά την υποβολή της αίτησης ενώ οι άλλες τρεις (Missing Requirement, No Fault, Other System) είναι «τελικές», για χρήση σε συγκεκριμένες περιπτώσεις:

1. Τεχνικό σφάλμα (Technical Fault), κατηγορία που υποδηλώνει αστοχία στον κώδικα και είναι μεσαίας προτεραιότητας
2. Σφάλμα λογικής (Business Fault), κατηγόρια που υποδηλώνει λανθασμένη λογική ή λογικό λάθος και είναι υψηλής προτεραιότητας
3. Σφάλμα παραμετροποίησης (Configuration Fault), που υποδηλώνει λάθος στα δεδομένα της βάσης και είναι μεσαίας προτεραιότητας
4. Ελλιπής υλοποίηση (Missing Implementation), που υποδηλώνει έλλειψη στην υλοποίηση του έργου, δηλαδή κάποιες από τις απαιτήσεις του πελάτη δεν έχουν υλοποιηθεί, και είναι ύψιστης προτεραιότητας
5. Ελλιπείς απαιτήσεις (Missing Requirement), που υποδηλώνει έλλειψη στις απαιτήσεις από την πλευρά του πελάτη και είναι χαμηλής προτεραιότητας
6. Κανένα σφάλμα (No fault), «τελική» που υποδηλώνει ότι δεν υπάρχει κάποιο σφάλμα και είναι «πληροφοριακής» προτεραιότητας
7. Ερώτηση (Question), για περιπτώσεις όπου ο πελάτης επιθυμεί κάποια αποσαφήνισει ή οδηγία και είναι «πληροφοριακής» προτεραιότητας
8. Ξένο σύστημα (Other system), για περιπτώσεις όπου το πρόβλημα υπάρχει ή πηγάζει από αστοχία σε τρίτο σύστημα και είναι χαμηλής προτεραιότητας

Η εφαρμογή υποστηρίζει τους εξής τύπους πελατών:

1. ιδιώτης (PRIVATE NON FAMILY), που συνδέεται με χαμηλή προτεραιότητα
2. οικογένεια (PRIVATE FAMILY), που συνδέεται με μεσαία προτεραιότητα
3. μικρή επιχείρηση με έναν υπάλληλο ή ελεύθερος επαγγελματίας (BUSINESS SINGLE), που συνδέεται με μεσαία προτεραιότητα
4. μικρή επιχείρηση (BUSINESS SMALL), που συνδέεται με υψηλή προτεραιότητα

5. μεγάλη επιχείρηση (BUSINESS LARGE), που συνδέεται με ύψιστη προτεραιότητα
6. η αίτηση αφορά σε όλους τους τύπους πελάτη (ALL), που συνδέεται με μεσαία προτεραιότητα
7. επιτρέπει την επιλογή από λίστα με πελάτες (SPECIFIC CUSTOMER), που συνδέεται με την προτεραιότητα του εκάστοτε πελάτη

Οι κατηγορίες αριθμού κρουσμάτων είναι επιλογές που αντικατοπτρίζουν την επίπτωση του σφάλματος στους πελάτες από μια ποσοτική οπτική. Πιο συγκεκριμένα, ορίζονται οι εξής:

1. Κανένας (None), που σημαίνει ότι είτε δεν είναι σφάλμα είτε δεν έχει επηρεάσει κάποιον πελάτη ακόμα και συνδέεται με την «πληροφοριακή» προτεραιότητα
2. Συγκεκριμένος αριθμός (Number inserted), που επιτρέπει στην χρήστη να εισάγει συγκεκριμένο αριθμό και η προτεραιότητα θα προκύψει ανάλογα με τον αριθμό που θα εισάγει ο χρήστης
3. Μερικοί (Few), δηλαδή το σφάλμα έχει επηρεάσει μικρό αριθμό πελατών και συνδέεται με την χαμηλή προτεραιότητα
4. Αρκετοί (Many), δηλαδή το σφάλμα έχει επηρεάσει μεγάλο αριθμό πελατών και συνδέεται με την υψηλή προτεραιότητα
5. Όλοι όσοι χρησιμοποιούν την υπηρεσία (All that use problematic service), δηλαδή η συγκεκριμένα υπηρεσία ή λειτουργία δεν είναι λειτουργική και συνδέεται με την ύψιστη προτεραιότητα

Να σημειωθεί ότι υπάρχει ξεχωριστός πίνακας στη βάση που αντιστοιχεί τις προτεραιότητες αίτησης με εύρος αριθμού ατόμων.

Ως αιτήσεις υπόστηριξης ορίζουμε το ελάχιστο σύνολο της πληροφορίας απαραίτητο για να περιγράψει ένα πρόβλημα που παρατηρεί ο πελάτης (ο οποίος ανήκει στους *ticket submitters*) και να μπορεί να το διαχειριστεί η ομάδα που έχει αναλάβει την υλοποίηση ή/και υποστήριξη του έργου.

Η πληροφορία αυτή περιλαμβάνει τα εξής στοιχεία:

- Το *username* του χρήστη που υποβάλλει την αίτηση
- Η κατηγορία του σφάλματος
- Ο τύπος του πελάτη
- Ο πελάτης (στην περίπτωση που έχει επιλεχθεί συγκεκριμένος πελάτης και όχι κατηγορία)
- Η κατηγορία αριθμού κρουσμάτων
- Ο αριθμός κρουσμάτων (στην περίπτωση που έχει επιλεχθεί να συμπληρωθεί ο αριθμός)
- Έργο στο οποίο αναφέρεται η αίτηση
- Περιγραφή του σφάλματος

- Ένα μήνυμα και μέχρι πέντε προαιρετικά αρχεία (attachments)

Η αίτηση αποτελεί μια οντότητα της εφαρμογής και, συνεπώς, υπάρχουν σχετικοί πίνακες στη βάση. Οι συγκεκριμένοι πίνακες περιέχουν την εξής πληροφορία που συμπληρώνεται από το σύστημα ή από χρήστες κατά την επεξεργασία της αίτησης:

- Την κατάσταση της αίτησης (οι διάφορες καταστάσεις του αιτήματος θα περιγραφούν παρακάτω)
- Η προτεραιότητα της αίτησης
- Ο χρήστης στον οποίο έχει ανατεθεί
- Η ομάδα στην οποία έχει ανατεθεί
- Ημερομηνία μέχρι την οποία πρέπει να έχει κλείσει η αίτηση
- Ημερομηνία υποβολής της αίτησης
- Ημερομηνία τελευταίας αλλαγής της αίτησης

Κύκλος ζωής αίτησης

Μέσα στην εφαρμογή, μία αίτηση έχει έναν κύκλο ζωής: αρχικά, κάποιος χρήστης παρατηρεί ένα πρόβλημα και υποβάλλει μια αίτηση υποστήριξης. Στη συνέχεια, οι ομάδες υπεύθυνες για την διεκπεραίωση διορθώσεων ή αλλαγών επεξεργάζονται την αίτηση και, αν όντως υπάρχει κάποιο σφάλμα, προβαίνουν στις απαραίτητες ενέργειες και «κλείνουν» την αίτηση. Αν η πληροφορία που έχει δώσει ο χρήστης δεν είναι αρκετή, τότε ζητάνε παραπάνω στοιχεία. Στην περίπτωση, που δεν υπάρχει σφάλμα, απαντάνε εξηγώντας γιατί το φαινόμενο που παρατήρησε ο χρήστης δεν ήταν σφάλμα και «κλείνουν» την αίτηση. Αν η αίτηση δεν έχει «κλείσει» μέσα στο χρόνικό όριο που αναλογεί στην προτεραιότητα της, τότε είναι εκπρόθεσμη και δεν μπορεί κανένας χρήστης να την επεξεργαστεί.

Για να τηρηθεί ο κύκλος ζωής που περιγράφηκε έχουμε ορίσει στην εφαρμογή τέσσερις καταστάσεις αίτησης:

- Ανοιχτή (Open): Αρχικό στάδιο της αίτησης.
- Υπό επεξεργασία (In progress): Η αίτηση βρίσκεται υπό επεξεργασία. Σε αυτή τη φάση, οι υπεύθυνες ομάδες ερευνούν το σφάλμα ή την ερώτηση και ετοιμάζουν απάντηση.
- Ολοκληρωμένη (Complete): Οι υπεύθυνη ομάδα έχει δώσει λύση ή απάντηση και αναμένεται να την αποδεχτεί ο χρήστης που υπέβαλε την αίτηση.
- Κλειστή (Closed): Ο χρήστης που υπέβαλε την αίτηση δέχτηκε την απάντηση ή λύση που προτάθηκε.

Ο κύκλος ζωής μιας αίτησης, λοιπόν, περιγράφεται από τα παρακάτω βήματα:

Βήμα 1: Ένας χρήστης της ομάδας *ticket submitters* υποβάλλει μια νέα αίτηση. Η αίτηση αυτή αρχικοποιείται με τα στοιχεία που εισήγαγε ο χρήστης, την προτεραιότητα που υπολογίζεται με βάση αυτά τα στοιχεία και την κατάσταση «Ανοιχτή» και ανατίθεται στην ομάδα *support* του συγκεκριμένου έργου.

Βήμα 2: Η ομάδα *support* ελέγχει τα στοιχεία που εισήγαγε ο χρήστης καθώς και τα στοιχεία που παράχθηκαν από την εφαρμογή. Σε περίπτωση που τα στοιχεία είναι σωστά, αναθέτει την αίτηση στην ομάδα *IT* και η αίτηση μεταβαίνει στην κατάσταση «Υπό επεξεργασία» και στο επόμενο βήμα.

Η ομάδα *support* έχει πλήρη δικαιώματα στην αίτηση, που σημαίνει ότι μπορεί και από αυτό το βήμα να την κλείσει αν θεωρήσει ότι τα στοιχεία είναι λανθασμένα και γενικά να τροποποιήσει την αίτηση ή να επέμβει στη διαδικασία με όποιον τρόπο θεωρήσει απαραίτητο για την σωστή διεκπεραίωση της αίτησης.

Βήμα 3: Τώρα η αίτηση βρίσκεται σε κατάσταση «Υπό επεξεργασία». Η αίτηση θα ανατεθεί σε κάποιο άτομο από την ομάδα *IT* το οποίο θα ερευνήσει την υπόθεση. Όταν το άτομο αυτό έχει κάποια απάντηση ή έχει δώσει λύση, η αίτηση μεταβαίνει στην κατάσταση «Ολοκληρωμένη».

Βήμα 4: Η αίτηση ανατίθεται πίσω στην ομάδα *ticket submitters*. Ο χρήστης που την υπέβαλλε έχει το δικαίωμα να κλείσει την αίτηση αν θεωρήσει ότι ή απάντηση ή λύση είναι ικανοποιητική αλλιώς μπορεί να επαναφέρει την αίτηση στην κατάσταση «Υπό επεξεργασία». Τα βήματα 3 και 4 επαναλαμβάνονται μέχρι ο χρήστης που την υπέβαλλε να δεχτεί την προτεινόμενη λύση ή απάντηση.

Όπως και στο βήμα 2, έτσι και σε άλλες περιστάσεις η αίτηση ανατίθεται πρώτα στην ομάδα *support* για έγκριση της πληροφορίας που εμπεριέχεται.

3 Υλοποίηση της SLASUP

Στην ενότητα αυτή θα παρουσιάσουμε την υλοποίηση της εφαρμογής. Πιο συγκεκριμένα, θα αναφερθούν οι τεχνολογίες που χρησιμοποιήθηκαν και πώς αξιοποιήθηκαν καθώς και άλλες τεχνικές αποφάσεις που λήφθηκαν. Το όνομα της εφαρμογής είναι SLA SUpport – SLASUP.

Για την καλύτερη κατανόηση και ροή της εργασίας, θα ξεκινήσουμε παρουσιάζοντας τις τεχνολογίες που χρησιμοποιήθηκαν και θα συνεχίσουμε με την δομή και την τεκμηρίωση της εργασίας.

3.1 Τεχνολογίες

Στην ενότητα αυτή περιγράφουμε με λίγα λόγια τις τεχνολογίες που χρησιμοποιήσαμε για την ανάπτυξη της εφαρμογής.

3.1.1 Angular.js

Angular.js (1.X) ονομάζεται ένα Javascript MVC Framework της Google. Η Angular είναι ανοιχτού κώδικα και έχει σαν στόχο να αποδεσμεύσει την διαχείριση του DOM από την λογική, να αποδεσμεύσει πλήρως το front-end τμήμα της εφαρμογής από το back-end, ώστε να γίνεται παράλληλα η υλοποίηση αλλά και να είναι επαναχρησιμοποιήσιμα, και, τέλος, να οργανώσει σε μια δομή το χτίσιμο μιας εφαρμογής.

Η Angular έχει δύο βασικά χαρακτηριστικά: τις δομές που ορίζει σαν μέρος του framework και το dependency injection. Dependency injection ονομάζεται η πρακτική κατά την οποία τα χαρακτηριστικά κάποιου αντικειμένου (dependency) εισάγονται στην κατάσταση (state) ενός άλλου αντικειμένου, το οποίο τα χρησιμοποιεί.

Στο framework αυτό ορίζονται διάφοροι τύποι δεδομένων, ο καθένας εκ των οποίων έχει διαφορετικά χαρακτηριστικά και χρήση:

Scope

Το Scope είναι, αρχιτεκτονικά, το πιο σημαντικό αντικείμενο της Angular. Σαν αντικείμενο, αποτελεί ένα πλαίσιο εκτέλεσης (execution context). Η Angular τυπικά δημιουργεί μια ιεραρχική δομή από αντικείμενα τέτοιου τύπου που «καθρεπτίζει» το DOM της εφαρμογής. Τη δημιουργία του αντικειμένου αυτού τη διαχειρίζεται εσωτερικά.

Directives

Τα directives είναι ειδικές html ετικέτες (tags) ή ιδιότητες (properties) τα οποία τροποποιούν τη συμπεριφορά των υπαρχόντων ετικετών που παρέχει η html και το DOM γενικότερα. Τα directives της Angular ξεκινάνε πάντα με την πρόθεση «ng-» ή «data-ng-» (η δεύτερη θεωρείται ορθή όταν η html μιας εφαρμογής περνάει από ελέγχους ορθότητας).

Controllers

Οι Angular controllers επεκτείνουν τα αντικείμενα τύπου Scope μέσω μιας κατασκευαστικής συνάρτησης. Τα αντικείμενου αυτού του τύπου ουσιαστικά αντιστοιχούνται στο DOM μιας όψης, δηλαδή ενός συνόλου στοιχείων του DOM, μέσω της ετικέτας «ng-controller» και χρησιμοποιούνται για να διαχειρίζονται τιμές και οποιαδήποτε λογική χρειάζεται η συγκεκριμένη όψη. Ο ρόλος των αντικειμένων αυτών είναι συγκεκριμένος: αρχικά, δίνουν μια αρχική κατάσταση στο Scope στο οποίο αντιστοιχούν και, δεύτερον, του προσθέτουν συμπεριφορά.

Services

Τα Angular services είναι singletons που επιτρέπουν στα υπόλοιπα αντικείμενα της Angular να μοιράζονται συναρτήσεις, λογικές ή και τιμές. Επειδή είναι singletons, τα services αυτά υπάρχουν μοναδικά σαν αντικείμενα στο Scope της εφαρμογής και χρησιμοποιούνται μέσω dependency injection.

Constants

Το αντικείμενο αυτό είναι επίσης singleton και χρησιμοποιείται για δήλωση σταθερών τιμών που μπορεί να χρειάζονται άλλα μέλη της εφαρμογής.

Επιλέξαμε να χρησιμοποιήσουμε την Angular.js για την εφαρμογή γιατί είναι πιο απλή και γρήγορη στη εγκατάσταση και, επίσης, υποστηρίζει περισσότερες προσαρμοσμένες λογικές.

3.1.2 Java Server Page

Μία Java Server Page είναι ουσιαστικά ένα έγγραφο html το οποίο εκτός από τις ετικέτες της html (html tags) έχει και ετικέτες JSP οι οποίες υποδηλώνουν δυναμικό περιεχόμενο.

Πιο συγκεκριμένα, υπάρχουν οι εξής ετικέτες:

- Page directives - <%@ page … %>, το οποίο θέτει διάφορες ιδιότητες σχετικές με το περιεχόμενο της σελίδας.
- Tag library directives - <%@ taglib … %>, το οποίο χρησιμοποιείται για την εισαγωγή βιβλιοθηκών.
- jsp:userBean, στοιχείο που δημιουργεί ένα αντικείμενο και αρχικοποιεί μια μεταβλητή με αυτό
- ${ ... }, οδηγία που εμφανίζει την τιμή ενός αντικειμένου, μιας μεταβλητής ή την τιμή που προκύπτει από μια έκφραση
- jsp:setProperty, στοιχείο που θέτει τιμή σε ιδιότητα ενός αντικειμένου
- άλλες ετικέτες που εκτελούν λειτουργίες όπως λογικές συνθήκες, επαναληπτικούς βρόγχους κλπ.

3.1.3 Spring Framework

Το Spring Framework αποτελεί ανοιχτού-κώδικα λογισμικό που διευκολύνει και αυτοματοποιεί την ανάπτυξη εφαρμογών J2EE. Βασικό στοιχείο του αποτελεί η IoC υλοποίηση του: μέσω αρχείων παραμετροποίησης, το Spring δημιουργεί και διαχειρίζεται αυτόματες Java κλάσεις (beans) για τις πιο απλές και γενικές εργασίες της εφαρμογής, επιτρέποντας στον προγραμματιστή να ασχοληθεί καθαρά και μόνο με την υλοποίηση κλάσεων που θα εκτελούν τις πιο ειδικές ανάγκες της εφαρμογής.

Το κύριο module του Spring (Spring Core Container) περιλαμβάνει τις βασικές βιβλιοθήκες του Spring, δηλαδή το BeanFactory και το ApplicationContext. Το BeanFactory αποτελεί μία διεπαφή (interface) προς τα beans που αποτελούν την εφαρμογή καθώς και προς εργαλεία διαχείρισης τους. Το ApplicationContext είναι μια κεντρική διεπαφή που παρέχει πρόσβαση στις παραμέτρους της εφαρμογής καθώς και αναφορές σε άλλα modules ή εργαλεία.

Το Spring προσφέρει δική του υποστήριξη για τις περισσότερες μοντέρνες βάσεις δεδομένων με ειδικές κλάσεις για διαχείριση πόρων, εξαιρέσεων, συναλλαγών καθώς και λειτουργίες για τη διαχείριση δεδομένων τύπου CLOB ή BLOB. Παράλληλα, επιτρέπει την χρήση εξωτερικών βιβλιοθηκών, στην περίπτωση που ο προγραμματιστής το επιθυμεί. Επίσης, το Spring υποστηρίζει πλήρως το Hibernate Framework (το οποίο περιγράφεται σε επόμενη ενότητα).

Επιπροσθέτως, το Spring περιλαμβάνει δικό του MVC Framework μέσω της υλοποίησης ενός Servlet API με μεθόδους και εργαλεία για την διαχείριση αιτημάτων HTTP. Η κλάση DispatcherServlet, συγκεκριμένα, είναι η κλάση που διαχειρίζεται και φιλτράρει όλα τα αιτήματα που δέχεται η εφαρμογή. Μέσω παραμετροποίησης, η κλάση αυτή προωθεί το κάθε αίτημα σε επιμέρους διεπαφές ή το απορρίπτει. Ο πιο βασικός των τύπων αυτών είναι η διεπαφή Controller. Οι διεπαφές αυτές ουσιαστικά δέχονται συγκεκριμένα αιτήματα, τα οποία δηλώνονται στην υλοποίηση των μεθόδων τους, και διαμορφώνουν και επιστρέφουν τις κατάλληλες απαντήσεις.

3.1.4 Hibernate ORM

Το Hibernate ORM, τμήμα του Hibernate Framework, αποτελεί υλοποίηση των προδιαγραφών JPA (Java Persistence API), και επιτρέπει data persistence σε κλάσεις που μπορεί να χρησιμοποιούν αντικειμενοστραφείς τεχνικές όπως κληρονομικότητα, πολυμορφισμό καθώς και Java collections.

Επίσης, το Hibernate υποστηρίζει αργή προετοιμασία (lazy initialization) αντικειμένων και μια ποικιλία στρατηγικών ανάκτησης δεδομένων. Δεν απαιτεί επιπρόσθετες κολώνες ή πίνακες στη βάση δεδομένων και, με την βοήθεια Java annotations, διευκολύνει την υλοποίηση.

Κομμάτι του Hibernate ORM είναι και η HQL (Hibernate Query Language), μια αντικειμενοστραφής μορφή της SQL, η οποία έχει παρόμοια δομή και σύνταξη με την δεύτερη αλλά αντί για ονόματα πινάκων και κολώνων χρησιμοποιούνται τα αντίστοιχα ονόματα Java αντικειμένων και πεδίων.

3.1.5 MySQL

Η MySQL είναι ένα ανοιχτού κώδικα σύστημα διαχείρισης σχεσιακών βάσεων δεδομένων της Oracle. Υποστηρίζει ένα ευρύ φάσμα εντολών της ANSI SQL 99 καθώς και επεκτάσεις της. Εκτός από τα παραπάνω, παρέχει δικό της Caching καθώς και transaction management.

3.1.6 Maven

Το Maven της Apache είναι ένα εργαλείο που αυτοματοποιεί τη διαδικασία χτισίματος εφαρμογών σε Java, κυρίως, αλλά και άλλες γλώσσες. Βασίζεται σε ένα αρχείο παραμετροποίησης, το pom.xml, όπου ορίζονται οι εξαρτήσεις της εφαρμογής ή, πιο γενικά, του έργου, καθώς και ο τρόπος με τον οποίο θα χτιστεί και η μορφή του τελικού αρχείου. Στη συνέχεια, το maven βρίσκει και κατεβάζει αυτόματα όλες τις δηλωμένες εξαρτήσεις καθώς και τις εξαρτήσεις τους και χτίζει την εφαρμογή στο τελικό αρχείο που έχει οριστεί.

Το Maven πέρα από το χτίσιμο και τις εξαρτήσεις, μπορεί να υποστηρίξει και άλλες λειτουργίες μέσω της χρήσης plug-in, όπως π.χ. το αυτόματο ανέβασμα σε application server ή την αποκρυπτογράφηση (obfuscation) και ελαχιστοποίηση (minification) κώδικα.

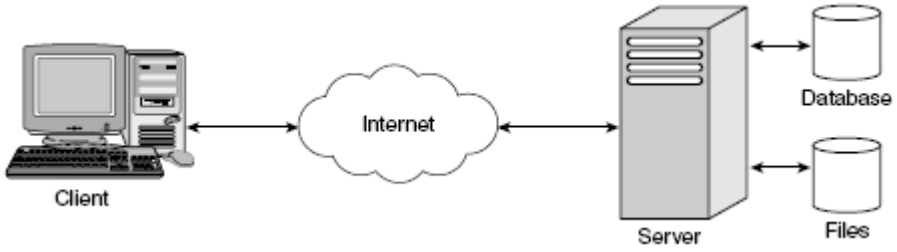
3.1.7 Log4J

Η Log4J είναι μια βιβλιοθήκη ενίσχησης των αρχείων καταγραφής της Java. Προσφέρει παραπάνω επιλογές στον τρόπο με τον οποίο καταγράφεται η πληροφορία καθώς και στη διαχείριση των αρχείων καταγραφής.

3.2 Δομή της εφαρμογής

Στην ενότητα αυτή θα περιγράψουμε την δομή της εφαρμογής, δηλαδή τα τμήματα από τα οποία αποτελείται.

Δεδομένης της φύσης της - η εφαρμογή θα υποστηρίζει πολλούς χρήστες οι οποίοι δεν θα βρίσκονται στον ίδιο χώρο και θα διαχειρίζονται κοινά δεδομένα - είναι λογικό η εφαρμογή να είναι διαδικτυακή (web application). Αυτό σημαίνει ότι η εφαρμογή θα έχει τρία μεγάλα τμήματα: ένα τμήμα που θα τρέχει στον εξυπηρετητή (back-end) (ή server-side), ένα τμήμα που θα τρέχει στον περιηγητή ιστού του κάθε χρήστη (front-end) (ή client-side) και, τέλος, μια βάση δεδομένων.



Εικόνα 1. Η βασική δομή μίας διαδικτυακής εφαρμογής

Για λόγους ασφάλειας αλλά και σωστής σχεδίασης, στο «front-end» τμήμα της εφαρμογής δεν θα πρέπει να υπάρχει καθόλου λογική, παρά μόνο σε ό,τι αφορά την επιθυμητή αναπαράσταση των δεδομένων που λαμβάνονται από το «back-end» τμήμα. Αντίθετα, στο *back-end* κομμάτι θα υπάρχει όλη η λογική για τη διαχείριση των δεδομένων.

Το front-end κομμάτι της εφαρμογής είναι ουσιαστικά μια ιστοσελίδα που θα εκτελείται στον περιηγητή του χρήστη. Συνεπώς, θα χρησιμοποιηθούν οι τεχνολογίες HTML, Javascript και τρία αρχεία JSP. Δεδομένου του ότι εδώ θα έχουμε λογική μόνο για την σωστή (και αισθητική) απεικόνιση των δεδομένων, είναι καλύτερο να χρησιμοποιήσουμε μόνο τις απαραίτητες

τεχνολογίες. Για τους λόγους που εξηγήσαμε παραπάνω, χρησιμοποιήσαμε το Angular.js javascript framework για αυτό το τμήμα της εφαρμογής.

Για το back-end της εφαρμογής, θα χρησιμοποιήσουμε το Spring Framework της Java. Το Spring Framework μας παρέχει έναν έτοιμο σκελετό για διαδικτυακές εφαρμογές.

Τέλος, για τη βάση δεδομένων επιλέξαμε την MySQL που είναι η πιο διαδεδομένη. Τεχνικά, δεδομένου του ότι το σύστημα δεν διαχειρίζεται πολύ πολύπλοκη πληροφορία, οποιαδήποτε σχεσιακή βάση δεδομένων είναι κατάλληλη για τις ανάγκες της εφαρμογής.

Επειδή η εφαρμογή μας θα χειρίζεται πολλούς τύπους δεδομένων και οντότητες (αιτήσεις, χρήστες κλπ.) επιλέγουμε να χρησιμοποιήσουμε, επίσης, τη βιβλιοθήκη Hibernate ORM. Το συγκεκριμένο ORM υποστηρίζεται απο το Spring Framework που θα χρησιμοποιήσουμε.

3.3 Τεκμηρίωση εφαρμογής

Στην ενότητα αυτή θα περιγράψουμε την υλοποίηση της εφαρμογής δίνοντας έμφαση στα πιο σημαντικά σημεία και τεχνικές. Σε επόμενες υποενότητες θα αναλυθούν τα κυριότερα αρχεία και σημεία της υλοποίησης.

Όπως προαναφέραμε, η εφαρμογή χωρίζεται σε τρία τμήματα: τον *client-side* κώδικα, τον *server-side* κώδικα και τη βάση δεδομένων, την οποία δεν θα περιγράψουμε σε αυτήν την ενότητα. Συνολικά, η εφαρμογή έχει δομηθεί με βάση το μοντέλο MVC. Αυτό σημαίνει ότι το client-side τμήμα της εφαρμογής θα πραγματοποιεί REST κλήσεις στο αντίστοιχο API του server-side τμήματος, το οποίο θα αποτελείται από μεθόδους ορισμένες σε έναν ή παραπάνω spring controllers.

Το client-side τμήμα της εφαρμογής αποτελείται από τρία επίπεδα: τις όψεις (views), τους ελεγκτές των όψεων (Angular controllers) και τις βιβλιοθήκες (Angular services, factories κλπ). Τα views και οι Angular controllers διαχειρίζονται το περιβάλλον του χρήστη και την απεικόνιση της πληροφορίας ενώ οι Angular services καλύπτουν ανάγκες κοινές για όλα τα views καθώς και την επικοινωνία με το REST API.

Βάση της εφαρμογής αποτελούν τα .jsp αρχεία index.jsp και dashboard.jsp, τα οποία εισάγουν .html αρχεία που περιλαμβάνουν επιμέρους όψεις ανάλογα με τις επιλογές του χρήστη. Αντίστοιχα, οι Angular controllers UserController και ContentController διαχειρίζονται την εφαρμογή συνολικά. Η κάθε επιμέρους όψη έχει αντίστοιχα και έναν Angular controller που την διαχειρίζεται.

Όταν η εφαρμογή χρειάζεται να πραγματοποιήσει κάποια κλήση στο back-end, ο Angular controller της όψης στην οποία ο χρήστης βρίσκεται τη δεδομένη στιγμή χρησιμοποιεί τις αντίστοιχες μεθόδους του UhttpService, στο οποίο ορίζονται συναρτήσεις για αυτόν τον σκοπό. Όπως το UhttpService, έτσι και τα υπόλοιπα Angular services ορίζουν συναρτήσεις για συγκεκριμένες ανάγκες της εφαρμογής.

Το server-side τμήμα της εφαρμογής απότελείται επίσης από τρία επίπεδα: τους REST Controllers, τα Services και τα Data Access Objects (DAOs). Όλα τα παραπάνω είναι συγκεκριμένου τύπου Spring Beans, Java κλάσεις.

Στους REST Controllers ορίζονται οι REST κλήσεις, οι οποίες αποτελούν το API της εφαρμογής και οι οποίες είναι διαθέσιμες στο client-side τμήμα της εφαρμογής. Οι REST Controllers φιλτράρουν τα αιτήματα που δέχονται και ετοιμάζουν τις απαντήσεις με την βοήθεια των Services.

Τα Services υλοποιούν όλη τη λογική (το business layer) της εφαρμογής. Οι REST Controllers καλούν μεθόδους των Services, ανάλογα με το αίτημα πάντα, και «δένουν» το αποτέλεσμα αυτό στις απαντήσεις που θα στείλουν στον client.

Τέλος, τα Data Access Objects ή DAOs, χειρίζονται τις συναλλαγές με τη βάση δεδομένων. Αυτό σημαίνει ότι τα Services διαβάζουν πληροφορία από τη βάση δεδομένων μέσω των DAO.

3.3.1 Project configuration

Λόγω του maven και του Spring αλλά και του γεγονότος ότι η εφαρμογή είναι διαδικτυακή, είναι απαραίτητο να ορίσουμε κάποια βασικά αρχεία παραμετροποίησης.

pom.xml

Το πρώτο και πιο βασικό αρχείο παραμετροποίησης της εφαρμογής είναι το pom.xml, στο οποίο καθορίζουμε τις βιβλιοθήκες που χρειαζόμαστε και τον τρόπο με το οποίο το maven θα χτίζει την εφαρμογή. Βασικά σημεία στο αρχείο αυτό αποτελούν τα δύο βασικά plugins που χρησιμοποιούμε: το maven-compiler-plugin για το χτίσιμο της εφαρμογής και το maven-war-plugin για τη σύνθεση της εφαρμογής σε ένα αρχείο .war.

web.xml

Από τη στιγμή που η εφαρμογή είναι διαδικτυακή, βασικό αρχείο για τη λειτουργία της είναι το web.xml. Αρχικά, στο αρχείο αυτό ορίζουμε πού υπάρχουν δύο άλλα αρχεία παραμετροποίησης, το applicationContext.xml και το log4j.properties, αρχείο παραμετροποίησης της βιβλιοθήκης Log4J. Στη συνέχεια ορίζουμε δύο listeners, τον Log4jConfigListener, ο οποίος χρησιμόποιείται από τη βιβλιοθήκη Log4J και διαβάζει και υλοποιεί το αρχείο log4j.properties, και τον ContextLoaderListener, η βασική κλάση διαχείρισης του Spring Context.

Το bean «ContextLoaderListener», ουσιαστικά επιτελεί δύο βασικούς ρόλους: πρώτον, δένει τον κύκλο ζωής του ServletContext με το ApplicationContext. Το ServletContext, όπως το ApplicationContext, είναι interface που παρέχει στο runtime της εφαρμογής αναφορές σε άλλα Spring beans και στοιχεία, σχετικά με τα servlet του Spring. Ο δεύτερος ρόλος του ContextLoaderListener, και ο βασικότερος, είναι ότι διαβάζει και παράγει αυτόματα τα beans που δηλώνονται στα αρχεία παραμετροποίησης.

Βασικό σημείο του web.xml είναι ο ορισμός του DispatcherServlet. Το Java servlet αυτό δέχεται τα αιτήματα, τα φιλτράρει και τα προωθεί στους REST Controllers για επεξεργασία. Πιο εκτενής παραμετροποίηση για το servlet αυτό πραγματοποιείται στο dispatcher-servlet.xml. Επίσης, ορίζουμε το servlet mapping, δηλαδή τι μορφή πρέπει να έχουν τα αιτήματα προς server-side.

dispatcher-servlet.xml

Στο dispatcher-servlet.xml ορίζονται παράμετροι και beans σχετικά με τη συμπεριφορά του DispatcherServlet.

Αρχικά, η σημείωση «resources» ορίζει αιτήματα τα οποία ζητάνε πόρους (όπως π.χ. στατικά αρχεία, βιβλιοθήκες και αρχεία .css) και τα οποία δεν παιρνάνε από φιλτράρισμα. Τέτοια αιτήματα δεν απευθύνονται στο REST API της εφαρμογής και, συνεπώς, τα διευθετεί αυτόματα το Spring.

Στη συνέχεια αντιστοιχούμε τις δύο κύριες όψεις, τα αρχεία .jsp, με αυτόματους REST Controllers (οι οποίοι είναι Spring beans τύπου ParameterizableViewController). Εδώ επιλέγουμε αυτό το configuration γιατί, όπως και οι πόροι, έτσι και τα δύο βασικά .jsp μας θέλουμε να επιστρέφονται όταν τα ζητάει το application. Η διαφορά εδώ από τους πόρους είναι τα .jsp αρχεία δεν είναι στατικά αλλά servlets. Συνεπώς, χρειάζονται Spring Controllers για να τα διαχειριστούν. Οι ParameterizableViewController είναι αυτόματοι Controllers του Spring οι οποίοι δέχονται και παραμέτρους.

Είναι σημαντικό να τονιστεί ότι οι παραπάνω παράμετροι καθορίζουν και τη δρομολόγηση της εφαρμογής. Για να αποκτήσει πρόσβαση στην εφαρμογή, ο χρήστης θα πρέπει να «χτυπήσει» το url της εφαρμογής με κατάληξη «/index» ώστε να φορτωθεί το index.jsp. Άμα τα στοιχεία που δώσει είναι σωστά, τότε η εφαρμογή θα τον ανακατευθύνει στο «/dashboard», δηλαδή το dashboard.jsp.

Ένα ακόμα σημαντικό bean που ορίζουμε είναι ο «multipartResolver», ο οποίος επιτρέπει στην εφαρμογή να υποστηρίζει αρχεία σαν μέρος των αιτημάτων. Στο bean αυτό μπορούν να οριστούν και επιμέρους ιδιότητες όπως το μέγιστο μέγεθος που να επιτρέπει η εφαρμογή ανά αρχείο, την οποία και θέτουμε.

Τέλος, χρησιμοποιούμε το bean «BeanNameUrlHandlerMapping» για να αντιστοιχίσουμε τις καταλήξεις των αιτημάτων με τους αντίστοιχους controllers. Για παράδειγμα, αιτήματα των οποίων το URL περιέχει το «/user» θα προωθούνται από το DispatcherServlet στον UserController.

applicationContext.xml

Στο αρχείο αυτό ορίζουμε τα beans που χρειάζονται για την διαχείριση των συναλλαγών της εφαρμογής.

Πρώτο και βασικό βήμα είναι ο ορισμός μιας σύνδεσης με τη βάση δεδομένων για τις ανάγκες της εφαρμογής. Για τον λόγο αυτό, ορίζουμε το bean «dataSource» και του δίνουμε σαν παραμέτρους τα στοιχεία σύνδεσης με τη βάση.

Όντας μια διαδικτυακή εφαρμογή, χρειαζόμαστε, επίσης, διαχείριση των συνεδριών με τη βάση. Για τον λόγο αυτόν ορίζουμε το bean «sessionFactory», δίνοντας του σαν παράμετρο το bean «dataSource» που ορίσαμε παραπάνω. Είναι σημαντικό να τονιστεί ότι το bean αυτό είναι τύπου «LocalSessionFactoryBean», κλάσης του Hibernate. Αυτή η επιλογή οφείλεται στο γεγονός ότι χρησιμοποιούμε γενικά το Hibernate για την διαχείριση της πληροφορίας. Συνεπώς, είναι καλύτερο να αφήσουμε το Hibernate να διαχειρίζεται και τις συνεδρίες με τη βάση.

Ομοίως, ορίζουμε το bean «transactionManager» τύπου HibernateTransactionManager για τη διαχείριση των συναλλαγών με τη βάση. Δεδομένου του ότι ο μόνος πόρος που χρειάζεται διαχείριση στο σύστημα είναι οι συναλλαγές με τη βάση, χρησιμοποιούμε τον transaction manager του Hibernate.

log4j.properties

Στο αρχείο αυτό περιγράφεται η μορφή των αρχείων καταγραφής που θα παράγει η εφαρμογή. Η Log4J μας επιτρέπει να έχουμε διαφορετικά αρχεία καταγραφής ανά πακέτο ή και κλάση.

3.3.2 Server-side

Όπως προαναφέραμε, το server-side της εφαρμογής αποτελείται από τρία επίπεδα:

- Τους REST Controllers, οι οποίοι διαχειρίζονται τα αιτήματα και τις απαντήσεις.
- Τα Services, που υλοποιούν όλη τη λογική της εφαρμογής.
- Τα DAOs, τα οποία πραγματοποιούν τις συναλλαγές με τη βάση.

Τα παραπάνω επίπεδα, βέβαια, αναφέρονται στην υλοποίηση και τη δομή της εφαρμογής σε συνδυασμό με το Spring. Για τις ανάγκες της εφαρμογής έχουν οριστεί διάφορα πακέτα κλάσεων, κάθε ένα εκ των οποίων περιέχει κλάσεις που επιτελούν μια συγκεκριμένη λειτουργία καθώς και άλλες οντότητες.

Στις επόμενες υποενότητες, λοιπόν, θα περιγράψουμε τα πακέτα και το ρόλο τους στη λειτουργία της εφαρμογής, δίνοντας έμφαση όπου χρειάζεται, και άλλες οντότητες όπως τα αιτήματα με τα οποία επικοινωνούν τα δύο τμήματα της εφαρμογής.

3.3.2.1 Πακέτο Common

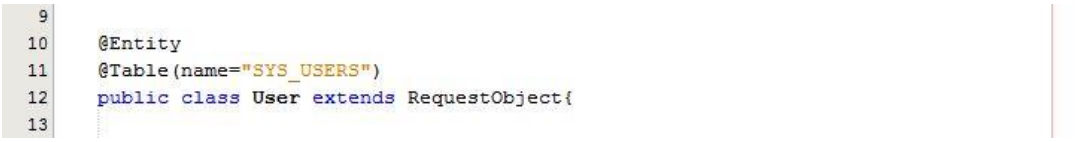
Το πακέτο *Common* περιέχει κλάσεις βοηθητικού χαρακτήρα καθώς και την κλάση *Constants*, στην οποία ορίζονται διάφορες σταθερές της εφαρμογής.

3.3.2.2 Πακέτο Model

Το πακέτο «model» περιέχει όλα τα POJOs, δηλαδή τα αντικείμενα Java που χρησιμοποιούνται στην εφαρμογή. Τα αντικείμενα αυτά προφανώς αποτελούν ένα πολύ σημαντικό κομμάτι της εφαρμογής μιας και αποτελούν την αποτύπωση της πληροφορίας.

Επιπροσθέτως, λόγω του Hibernate ORM, τα αντικείμενα αυτά επιτελούν διπλό ρόλο. Το Hibernate ORM μας επιτρέπει να χρησιμοποιήσουμε τα POJOs σαν χαρτογράφηση για την επικοινωνία με τη βάση. Αυτό σημαίνει ότι με τη βοήθεια κάποιον Java annotations, μπορούμε να δηλώσουμε σε κάθε POJO σε ποιον πίνακα της βάσης αντιστοιχεί, αν αντιστοιχεί σε κάποιον πίνακα, και, στη συνέχεια, με συγκεκριμένες αυτόματες μεθόδους που δηλώνονται στο αντικείμενο Session που παράγει το bean «sessionFactory», μπορούμε να πάρουμε πληροφορία από τον πίνακα αυτόν.

Πιο συγκεκριμένα, στον ορισμό της κλάσης προσθέτουμε τα δύο annotations που φαίνονται στην παρακάτω εικόνα:

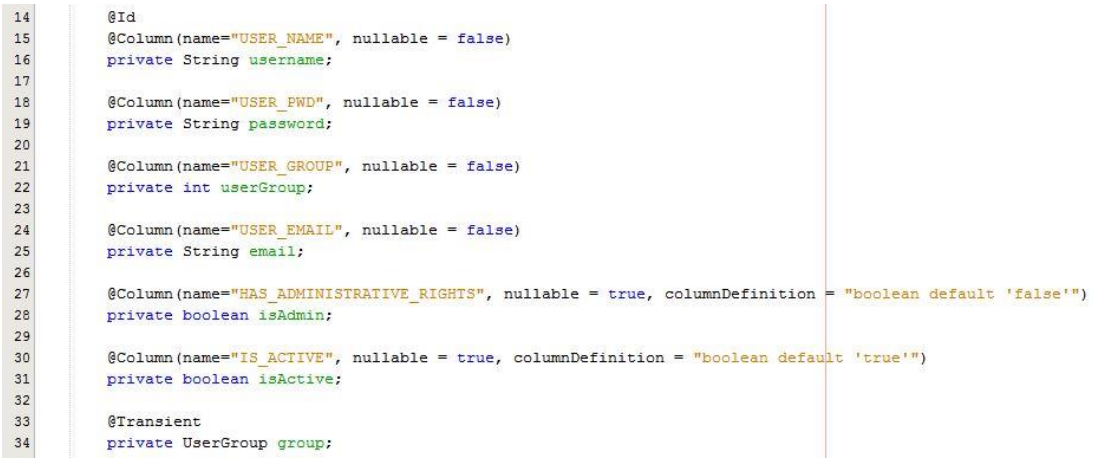


```
9
10  @Entity
11  @Table(name="SYS_USERS")
12  public class User extends RequestObject{
13
```

Εικόνα 2. Τα βασικά annotations μιας κλάσης

Το annotation «@Entity» δηλώνει στο Hibernate ότι η κλάση «User» αποτελεί ένα «Entity» του Hibernate, δηλαδή αντιστοιχεί σε κάποιον πίνακα στη βάση. Το δεύτερο annotation, «@Table(name="SYS_USERS")» δηλώνει ότι το Entity αυτό αντιστοιχεί στον πίνακα «SYS_USERS» της βάσης.

Στη συνέχεια, αντίστοιχούμε τα πεδία του αντικειμένου με κολώνες του πίνακα που ορίσαμε στο annotation «@Table», όπως φαίνεται στην παρακάτω εικόνα:



```
14      @Id
15      @Column(name="USER_NAME", nullable = false)
16      private String username;
17
18      @Column(name="USER_PWD", nullable = false)
19      private String password;
20
21      @Column(name="USER_GROUP", nullable = false)
22      private int userGroup;
23
24      @Column(name="USER_EMAIL", nullable = false)
25      private String email;
26
27      @Column(name="HAS_ADMINISTRATIVE_RIGHTS", nullable = true, columnDefinition = "boolean default 'false'")
28      private boolean isAdmin;
29
30      @Column(name="IS_ACTIVE", nullable = true, columnDefinition = "boolean default 'true'")
31      private boolean isActive;
32
33      @Transient
34      private UserGroup group;
```

Εικόνα 3. Χαρτογράφηση με τη βάση

Το annotation «@Id» δηλώνει ποια κολώνα είναι το primary key του πίνακα και ποιο πεδίο του αντικειμένου αντιστοιχεί σε αυτήν, ώστε οι αυτόματες μέθοδοι του Hibernate να μπορούν να τη χρησιμοποιήσουν.

Το annotation «@Column» αντιστοιχεί το πεδίο του αντικειμένου με κάποια κολώνα. Για παράδειγμα, το πεδίο «username» αντιστοιχεί στην κολώνα «USER_NAME». Η ιδιότητα «nullable» υποδηλώνει πεδία που στη βάση δεν δέχονται κενές τιμές. Αντίστοιχα, στην ιδιότητα «columnDefinition» αναγράφουμε ιδιαίτερες οδηγίες. Στο παράδειγμα μας, στο πεδίο «isAdmin» αναγράφουμε ότι η αυτόματη τιμή της μεταβλητής μας είναι «false».

Τέλος, σε κάποια αντικείμενα μπορεί να θέλουμε να δηλώσουμε πεδία τα οποία δεν πρέπει να λάβει υπόψην του το Hibernate. Αυτό το επιτυγχάνουμε με το annotation «@Transient». Στο παράδειγμά μας, το πεδίο «group», το οποίο είναι σύνθετο και τύπου «UserGroup», δεν θα ληφθεί υπόψη όταν το Hibernate θα φέρει δεδομένα από τη βάση για να παράξει ένα αντικείμενο τύπου «User».

Εδώ πρέπει να γίνει μια σημείωση για την έννοια του «persistence»: αυτό που κάνει το JPA και, κατ'επέκταση, το Hibernate είναι να κρατάει στη μνήμη τα αντικείμενα που έχει παράξει με πληροφορία από τη βάση, διαμορφώνοντας, έτσι, το λεγόμενο «persistence layer». Για τη σωστή διαχείριση των αντικειμένων αυτών, είναι μια καλή ιδέα να υλοποιούμε τις μεθόδους «equals» και «hashcode» της διεπαφής «Serializable». Οι μέθοδοι αυτοί χρησιμοποιούνται για

τον έλεγχο της ισότητας και ομοιότητας αντικειμένων και σε περιπτώσεις δομών που δεν επιτρέπουν διπλότυπες εγγραφές ή μέλη, όπως το Set της JAVA, είναι απαραίτητες. Στην περίπτωση του Hibernate, είναι επίσης απαραίτητες οι δύο αυτές μέθοδοι διότι το Hibernate κρατάει στην μνήμη το πολύ ένα αντικείμενο για κάθε γραμμή κάθε πίνακα.

3.3.2.3 Πακέτο Controller

Το πακέτο *Controller* περιέχει όλους του REST Controllers της εφαρμογής. Όπως αναφέραμε σε προηγούμενη ενότητα, οι REST Controllers ουσιαστικά διαχειρίζονται αιτήματα και, πιο συγκεκριμένα, τα αιτήματα που τους αντιστοιχούν μέσω του bean *BeanNameUrlHandlerMapping*.

Όλοι οι controllers δέχονται αιτήματα που τελειώνουν με το λεκτικό το οποίο έχει οριστεί στο dispatcher-servlet.xml (π.χ. ο UserController δέχεται αιτήματα των οποίων το URL τελειώνει σε «/user»). Επεξεργάζονται τα αιτήματα αυτά και επιστρέφουν μία απάντηση.

Πιο τεχνικά, ο ορισμός μιας κλάσης ως REST Controller γίνεται με το Java annotation *@RestController*, όπως φαίνεται στην παρακάτω εικόνα:

```
17  /**
18   *
19   * @author Plato
20   */
21  @RestController
22  public class UserController {
```

Εικόνα 4. Ορισμός ενός REST Controller

Το Spring υποστηρίζει παραμετροποίηση της εφαρμογής μέσω Java annotations και παρέχει ένα εκτεταμένο πλαίσιο από annotations για τέτοιες χρήσεις. Επιτρέπει, όμως, και τη χρήση αρχείων xml για παραμετροποίηση. Εμείς για την εφαρμογή αυτή επιλέξαμε να χρησιμοποιήσουμε έναν συνδυασμό των δύο. Συγκεκριμένα, όπως είδαμε και σε προηγούμενη ενότητα, πραγματοποιήσαμε την παραμετροποίηση της βάσης και των αιτημάτων σε αρχεία xml αλλά τις κλάσεις και τα εργαλεία του Spring τα ορίζουμε μέσω annotations.

Κάθε controller έχει μία μέθοδο, την «processRequest» η οποία ορίζεται όπως φαίνεται στην παρακάτω εικόνα:

```
39      @RequestMapping(value="**/user", method=RequestMethod.POST, consumes = "application/json")
40      @SuppressWarnings("CallToPrintStackTrace")
41      public ResponseEntity<String> processRequest(HttpServletRequest request){...58 lines }
```

Εικόνα 5. Η processRequest του UserController

Με μια πρόχειρη ματιά καταλαβαίνουμε ότι το πρώτο annotation αποτελεί παραμετροποίηση της μεθόδου. Συγκεκριμένα, το annotation *@RequestMapping* παίρνει τρεις παραμέτρους:

- value: η παράμετρος αυτή ορίζει ένα πρότυπο με βάση το οποίο θα πρέπει να δομούνται τα URL των αιτημάτων για να τα δέχεται. Συνεπώς, η μέθοδος *processRequest* του *UserController* δέχεται μόνο αιτήματα των οποίων το URL τελειώνει με το λεκτικό «user».
- method: η παράμετρος αυτή δηλώνει την HTTP μέθοδο που θα δέχεται.

- consumes: η παράμετρος αυτή δηλώνει το content-type του αιτήματος που δέχεται.

Συνεπώς, η συγκεκριμένη μέθοδος δέχεται POST αιτήματα, τύπου «application/json» των οποίων το URL τελειώνει με το λεκτικό «user».

Θεωρητικά, στο REST θα πρέπει να χρησιμοποιούμε την αντίστοιχη HTTP μέθοδο ανάλογα με τη φύση του αιτήματος. Για λόγους απλότητας και συντομίας, όμως, επιλέξαμε να χρησιμοποιήσουμε αυτή τη δομή ώστε να είναι κεντρικοποιημένη η διαχείριση των πόρων και των αιτημάτων σε μία μέθοδο.

Η πληροφορία που μεταφέρει το αίτημα μας, όπως φαίνεται και από το content-type που δέχεται η *processRequest*, είναι σε μορφή json. Επειδή σε κάθε *REST Controller* μας βολεύει η πληροφορία αυτή να αναπαρίσταται με διαφορετικό τρόπο, χρησιμοποιήσαμε το αντικείμενο *ObjectMapper* της βιβλιοθήκης *Jackson* για την μετατροπή του αντικειμένου. Για παράδειγμα:

```
48          ObjectMapper mapper = new ObjectMapper();
49          user = mapper.readValue(request.getInputStream(), User.class);
50          LOGGER.debug("Parsed request: " + (user!=null?user.toString():"empty"));
```

Εικόνα 6. Μετατροπή πληροφορίας με χρήση της βιβλιοθήκης Jackson

Όπως φαίνεται στην παραπάνω εικόνα (Εικόνα 6), αφού δημιουργήσουμε ένα αντικείμενο τύπου *ObjectMapper*, χρησιμοποιούμε την μέθοδο του *readValue.* Η μέθοδος αυτή παίρνει σαν ορίσματα το *stream* του αιτήματος και την κλάση με βάση την οποία θέλουμε να αναπαραστήσουμε την πληροφορία (στην περίπτωση του *UserController* την κλάση *User*) και επιστρέφει ένα στιγμιότυπο της κλάσης αυτής με την πληροφορία που περιέχεται στο αίτημα. Να σημειωθεί ότι για να πετύχει η διαδικασία αυτή πρέπει η κλάση που δίδεται σαν δεύτερο όρισμα να έχει πεδία με ίδια ονόματα με τα πεδία πάνω στο σώμα του αιτήματος διότι η βιβλιοθήκη *Jackson* βασίζεται σε *Java reflection* για να πραγματοποιήσει τη μετατροπή, δηλαδή προσπαθεί να αντιστοιχίσει κάθε πεδίο του αιτήματος με κάποιο από τα πεδία της κλάσης η οποία δόθηκε σαν όρισμα.

Κάτι που πρέπει να αναφέρουμε στο σημείο αυτό είναι ότι όλες οι κλάσεις που χρησιμοποιούνται σαν αναπαράσταση των αιτημάτων στους *Controllers* είναι υποκλάσεις της κλάσης *RequestObject*, η οποία ορίζει τρία πεδία απαραίτητα για κάθε αίτημα:

- *action*, το οποίο περιέχει κάποια από συγκεκριμένες τιμές, ορισμένες στην *Constants*, και δηλώνει τι ακριβώς ζητάει ο client.
- *rUsername και rPassword:* τα οποία είναι τα στοιχεία του χρήστη που είναι συνδεδεμένος (και πρακτικά στέλνει το αίτημα)

Από τα παραπάνω στοιχεία, το *action* είναι υποχρεωτικό και πρέπει πάντα να υπάρχει αλλιώς ο *REST Controller* δεν θα μπορεί να ξέρει τι πρέπει να κάνει με τα στοιχεία του αιτήματος.

Έχοντας όλη την απαραίτητη πληροφορία, ο *REST Controller* θα καλέσει, ανάλογα με την τιμή του πεδίου *action*, κάποια μέθοδο κάποιου *Service*:

```
51
                if(Constants.USER_LOGIN.equals(user.getAction())){
53                  result = userService.validateUserCredentials(user);
54              } else if(Constants.USER_GROUPS.equals(user.getAction())) {
55                  result = userService.getUserGroups();
56              } else if(Constants.USER_MODIFY.equals(user.getAction())) {
57                  result = userService.modifyUser(user);
58              } else if(Constants.USER_NEW.equals(user.getAction())) {
59                  result = userService.createNewUser(user);
60              } else if(Constants.USER_RETRIEVE.equals(user.getAction())
61                      || Constants.USER_MODIFY_RETRIEVE.equals(user.getAction())) {
62                  result = userService.retrieveUser(user);
63              } else if(Constants.USER_DELETE.equals(user.getAction())){
64                  result = userService.deactivateUser(user);
65              } else if(Constants.USER_FIRST_LOGIN.equals(user.getAction())){
66                  result = userService.performFirstLogin(user);
67              } else if(Constants.PROJECTS.equals(user.getAction())){
68                  result = userService.getProjects();
69              } else if(Constants.USERS_GET.equals(user.getAction())){
70                  result = userService.getAllUsers(user);
71              }
```

Εικόνα 7. Κλήσεις σε Service

Όπως φαίνεται στην παραπάνω εικόνα, το πεδίο *action* του αντικειμένου *user*, το οποίο διαβάζεται μέσω της *getter* μεθόδου *getAction*, συγκρίνεται με τιμές από την κλάση *Constants* και ανάλογα την τιμή καλείται και κάποια μέθοδος του αντικειμένου *userService*. Το αποτέλεσμα των μεθόδων αποθηκεύεται στην μεταβλητή *result,* τύπου *ExtendedObject*.

Ο τύπος *ExtendedObject* έχει δημιουργηθεί καθαρά και μόνο για την επικοινωνία μεταξύ των *REST Controllers* και των *Services* και περιέχει δύο πεδία: ένα πεδίο *object*, τύπου *Object*, για την μεταφορά αντικειμένων και ένα πεδίο *errorMessage*, τύπου *ErrorMessage*, για την μεταφορά μηνυμάτων σφάλματος.

Οι *REST Controllers* επιστρέφουν αντικείμενο τύπου *ResponseEntity<String>*, το οποίο περιέχει σαν σώμα ένα *String* και τα υπόλοιπα πεδία που χρειάζονται για ένα πλήρες *HTTP Response*. Η λογική που έχουμε υλοποιήσει ουσιαστικά ελέγχει πρώτα αν το αντικείμενο *result* δεν είναι κενό. Αν δεν είναι κενό και έχει τιμή στο πεδίο *object*, η διαδικασία θεωρείται επιτυχημένη. Αν δεν είναι κενό αλλά δεν έχει τιμή στο πεδίο *object* και έχει τιμή στο πεδίο *errorMessage*, τότε υπήρξε σφάλμα κατά τη διαδικασία ή αρκεί κάποιο μήνυμα σαν απάντηση – τα *headers* της απάντησης δεν θα είναι με σφάλμα. Τέλος, αν το *result* είναι κενό, σημαίνει ότι το πεδίο *action* του αιτήματος είχε τιμή η οποία δεν αναγνωρίστηκε από το σύστημα και, συνεπώς, δεν το επεξεργάστηκε. Στην περίπτωση αυτή απαντάμε με το HTTP σφάλμα *NOT_IMPLEMENTED*.

```
72              if(result!=null && result.hasObject()){
73                  responseContent = mapper.writeValueAsString(result.getObject());
74                  LOGGER.debug("object returned! Written: " + responseContent);
75              } else if(result!=null && result.getErrorMessage() != null) {
76                  responseContent = mapper.writeValueAsString(result.getErrorMessage());
77                  LOGGER.debug("no object returned! Written: " + responseContent);
78              } else {
79                  LOGGER.debug("unsupported action!");
80                  responseContent = mapper.writeValueAsString(systemService.getErrorMessageById(Constants.NOT_IMPLEMENTED));
81                  httpStatus = HttpStatus.NOT_IMPLEMENTED;
82              }
```

Εικόνα 8. Διαχείριση της απάντησης από το service

Όπως φαίνεται και από την εικόνα, η λογική αυτή ανανεώνει την τιμή δύο μεταβλητών: της *responseContent* που είναι τύπου String και της *httpStatus* η οποία έχει αρχικοποιηθεί με την HTTP κατάσταση *OK (200)* και αλλάζει σε *NOT_IMPLEMENTED (501)*.

Στην περίπτωση που προκληθεί κάποιο τεχνικό σφάλμα όσο το αίτημα βρίσκεται υπό επεξεργασία, η τιμή της μεταβλητής *httpStatus* γίνεται *INTERNAL_SERVER_ERROR (500)* και η τιμή της μεταβλητής *responseContent* τίθεται σε ένα μήνυμα λάθους.

Για ασφάλεια και σιγουριά ότι η απάντηση θα δωθεί σε οποιαδήποτε περίπτωση, όλη η παραπάνω διαδικασία πραγματοποιείται μέσα σε έναν βρόγχο *try – catch* της Java. Στο βρόγχο *finally*, ο οποίος έχει το χαρακτηριστικό ότι θα τρέξει πάντα ανεξάρτητα με το περιεχόμενο των βρόγχων *try* και *catch,* τοποθετήσαμε την δημιουργία του αντικειμένου που θα απαντήσει ο *REST Controller*, ώστε να μην υπάρχει φόβος ότι μπορεί το αντικείμενο αυτό, και συνεπώς η HTTP απάντηση, να είναι κενό:

```
83          } catch(Exception e){
84              LOGGER.debug("exception caught: " + e.getMessage());
85              e.printStackTrace();
86              try{
87                  responseContent = mapper.writeValueAsString(systemService.getErrorMessageById(Constants.GENERAL_ERROR));
88              } catch(Exception e1){}
89              httpStatus = HttpStatus.INTERNAL_SERVER_ERROR;
90          } finally {
91              response = new ResponseEntity<String>(responseContent, httpStatus);
92          }
```

Εικόνα 9. Error handling με βρόγχο finally

Ο FileController διαχειρίζεται αιτήματα που περιέχουν αρχεία. Λόγω ιδιαιτεροτήτων των αιτημάτων αυτών, δεν μας κάλυπτε μία μέθοδος αλλά έπρεπε να έχουμε ξεχωριστές μεθόδους για τη μεταφόρτωση αρχείων και την διαγραφή τους. Παρ'όλα αυτά, οι τεχνικές που αναλύσαμε πιο πάνω ισχύουν με την διαφορά ότι αντί για μία μέθοδο *processRequest*, έχουμε δύο *processRequest*, μία για download, μία για upload, και μία μέθοδο *processDeleteRequest.*

3.3.2.4 Πακέτο Service

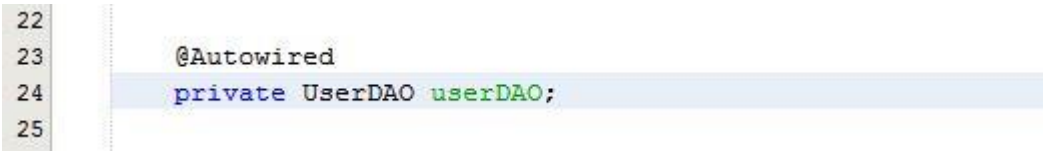
Το πακέτο *Service* περιέχει τις κλάσεις που χειρίζονται τη λογική της εφαρμογής και επεξεργάζονται την πληροφορία ώστε να δοθεί απάντηση στα αιτήματα. Ονομάζεται *Service* διότι οι κλάσεις αυτές ορίζονται ως *Spring Services*.

```
16
17      @Service
18      @SuppressWarnings("CallToPrintStackTrace")
19      public class UserServiceImpl implements UserService {
```

Εικόνα 10. Ορισμός ενός Service

Όπως φαίνεται στην παραπάνω εικόνα, ένα *Spring Service* ορίζεται μέσω του annotation *@Service*. Αυτό σημαίνει ότι η συγκεκριμένη κλάση αποτελεί *Spring Bean* και, συνεπώς, είναι διαθέσιμη σε οποιαδήποτε άλλη κλάση μέσω του μηχανισμού *dependency injection* του Spring. Επίσης, παρατηρούμε ότι η κλάση αυτή υλοποιεί ένα *interface* (στην προκειμένη περίπτωση, η κλάση *UserServiceImpl* υλοποιεί το *interface UserService,* και τα δύο προσαρμοσμένος κώδικας και όχι από τις βιβλιοθήκες του *Spring*.

Για να χρησιμοποιήσουμε ένα *Spring Bean* σε κάποια άλλη κλάση, αρκεί να το κάνουμε «inject» με τον ακόλουθο τρόπο:



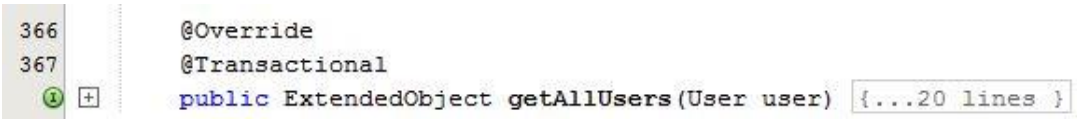
Εικόνα 11. Παράδειγμα dependency injection: εισάγουμε το UserDAO για χρήση στο UserService

Όπως φαίνεται στην παραπάνω εικόνα, για να χρησιμοποιήσουμε ένα *Spring Bean* σε κάποια άλλη κλάση ή αντικείμενο αρκεί να δηλώσουμε μια μεταβλητή τύπου ίδιου με το *interface* του και να σημειώσουμε το *annotation @Autowired*. Το *ApplicationContext*, όπως είχαμε πει σε προηγούμενη ενότητα, θα φροντίσει να παρέχει το *bean* που ζητάει ο κώδικας.

Αυτό είναι δυνατό διότι όλα τα *Spring Beans* είναι *singletons*. Αυτό σημαίνει ότι ο πάροχος IoC (Inversion of Control) του Spring δημιουργεί ένα μοναδικό αντικείμενο (instance) για κάθε *singleton Bean*. Τα αντικείμενα αυτά είναι διαθέσιμα όπως και τα αυτόματα *Spring beans* μέσω του *ApplicationContext*. Το annotation *@Autowired* αναζητά και παίρνει μία αναφορά στο *instance* του αντικειμένου που ψάχνει ο κώδικας χωρίς να υπάρχει ανάγκη για δημιουργία/αρχικοποίηση με τον τελεστή *new*.

Μέσα στην υλοποίηση ενός *Service* υπάρχουν δύο τύπων μέθοδοι: οι μέθοδοι που χρησιμοποιούν οι *REST Controllers* για την επεξεργασία των αιτημάτων και άλλες βοηθητικές μέθοδοι που χρησιμοποιούνται είτε ιδιωτικά από το ίδιο το *Service* είτε από άλλα *Services*. Ο διαχωρισμός αυτός γίνεται διότι στην πρώτη περίπτωση, ανεξαρτήτου *Service*, η μέθοδος θα επιστρέφει ένα αντικείμενο τύπου *ExtendedObject*. Στη δεύτερη περίπτωση, η κάθε μέθοδος επιστρέφει ό,τι βολεύει αναλόγως τη χρήση και την περίπτωση.

Στην ακόλουθη εικόνα φαίνεται ένα ορισμός μεθόδου ενός *Service*:



Εικόνα 12. Μία μέθοδος όπως ορίζεται σε ένα Service

Το πρώτο annotation - *@Override* – χρειάζεται διότι η μέθοδος αυτή είναι δηλωμένη στο *interface* της κλάσης. Επίσης, παρατηρούμε ότι η μέθοδος επιστρέφει αντικείμενο τύπου *ExtendedObject*, άρα πιθανώς να καλείται από κάποιον *REST Controller* (συγκεκριμένα καλείται από τον *UserController*).

Το annotation *@Transactional* χρησιμοποιείται όταν θέλουμε μια μέθοδος να εκτελείται πάντα στα πλαίσια μιας συναλλαγής (transaction). Το annotation αυτό πρέπει να χρησιμοποιείται όταν η μέθοδος (ή και η κλάση ολόκληρη) χρειάζεται πρόσβαση σε κάποιον πόρο, είτε αυτός ο πόρος είναι η βάση δεδομένων είτε ουρά. Μιας και στην εφαρμογή μας ο μόνος τέτοιος πόρος είναι η βάση δεδομένων, με αυτό το annotation θα σημειώνονται όσες μέθοδοι διαχειρίζονται αντικείμενα με το annotation *@Entity*.

```
365
366     @Override
367     @Transactional
        public ExtendedObject getAllUsers(User user) {
369         LOGGER.debug("Received request to get all users by: " + user.getUsername());
            ExtendedObject response = new ExtendedObject();
371
372         try {
373             response.setObject(userDAO.getAllUsers());
374         } catch (Exception e) {
375             LOGGER.error("Exception caught! Details: " + e.getMessage());
376             e.printStackTrace();
377             try {
378                 response.setObject(null);
379                 response.setErrorMessage(systemDAO.getErrorMessageById(Constants.GENERAL_ERROR));
380             } catch (Exception e1) {
381                 e1.printStackTrace();
382             }
383         }
384
385         LOGGER.debug("returning " + response.toString());
386         return response;
387     }
```

Εικόνα 13. Η μέθοδος getAllUsers

Στην παραπάνω εικόνα φαίνεται η υλοποίηση της μεθόδου *getAllUsers*. Η μέθοδος αυτή δέχεται ένα αντικείμενο τύπου *User*, διότι καλείται από τον *UserController* ο οποίος χρησιμοποιεί αυτόν τον τύπο για τα αιτήματα που δέχεται. Επίσης, βλέπουμε ότι επιστρέφει ένα αντικείμενο τύπου *ExtendedObject* και, πιο συγκεκριμένα, επιστρέφει τη μεταβλητή *response*, την οποία αρχικοποιεί στην αρχή της μεθόδου ώστε να φαίνεται σε όλο το σώμα της μεθόδου.

Μέσα στον βρόγχο *try* παρατηρούμε ότι απλά θέτει στο πεδίο *object* της *response* την τιμή που επιστρέφει η ομόνυμη μέθοδος του *UserDAO.* Μέσα στον βρόγχο *catch*, δηλαδή σε περίπτωση που υπάρξει σφάλμα, θέτει τα πεδία *object* και *errorMessage* σε *null* και ένα μήνυμα σφάλματος, αντίστοιχα. Τέλος, επιστρέφει τη μεταβλητή *response* στον *REST Controller*.

Όλες οι μέθοδοι που χρησιμοποιούνται από τους *REST Controllers* ακολουθούν αυτό το πρότυπο. Εννοείται ότι πολυπλοκότερες μέθοδοι έχουν παραπάνω λογική στο βρόγχο *try*, ανάλογα πάντα με τις ανάγκες της λειτουργίας για την οποία αναπτύχθηκαν.

3.3.2.5 Πακέτο DAO

Το πακέτο αυτό περιέχει κλάσεις οι οποίες πραγματοποιούν *hql queries*, τα *Data Access Objects* ή *DAOs*. Όπως προαναφέραμε, τεχνικά δεν υπάρχει κάποιος περιορισμός που να μας αναγκάζει να πραγματοποιούμε *queries* μόνο μέσω των κλάσεων αυτού του πακέτου. Παρ'όλα αυτά, υπάρχει μία λεπτή διαφορά μεταξύ των κλάσεων στο πακέτο *@Service* από τις κλάσεις που ορίζονται εδώ, η οποία φαίνεται στην παρακάτω εικόνα:

```
18  @Repository
19  @Transactional
20  @SuppressWarnings("CallToPrintStackTrace")
21  public class UserDAOImpl implements UserDAO{
```

Εικόνα 14. Ορισμός ενός DAO

Για τον ορισμό μιας κλάσης ως DAO χρησιμοποιούμε το annotation *@Repository*. Αρχικά, να αναφέρουμε ότι το annotation αυτό όπως και το *@Service* αποτελούν εξειδικεύσεις του

@Component. Τα δύο τελευταία, όμως, απλά ορίζουν τις κλάσεις στις οποίες χρησιμοποιούνται ως *Spring Beans.* Το annotation *@Repository*, όμως, επιτρέπει στο Spring να μεταφράσει σε *DataAccessException* οποιαδήποτε εξαίρεση (exception) προκληθεί από μια τέτοια κλάση και δεν ελέγχεται ή ορίζεται από κάποιον μηχανισμό (unchecked). Επίσης, να σημειωθεί ότι είναι προτιμότερο να χρησιμοποιούνται τα πιο συγκεκριμένα annotations γιατί καθορίζουν πιο ξεκάθαρα το λόγο ύπαρξης μιας κλάσης.

Να σημειωθεί, επίσης, ότι και τα *DAOs* έχουν δικά τους *interfaces*.

Όλες οι μέθοδοι που υλοποιούνται σε ένα *DAO* έχουν σαν σκοπό είτε την ανάκτηση δεδομένων από την βάση είτε την τροποποίηση των δεδομένων της βάσης. Για να γίνει αυτό χρησιμοποιείται το *interface Session* του Hibernate. Για να έχουμε πρόσβαση σε αυτό, χρησιμοποιούμε το *bean sessionFactory*, που είδαμε σε προηγούμενη ενότητα:

```
25      @Autowired
26      private SessionFactory sessionFactory;
```

Εικόνα 15. Ορισμός του sessionFactory σε ένα DAO

Στην επόμενη εικόνα, βλέπουμε την υλοποίηση της *getAllUsers*:

```
142
143     @Override
        public ArrayList<User> getAllUsers() throws Exception {
145         try {
146             Query query = sessionFactory.getCurrentSession().createQuery("from User u where u.isActive = true");
                return new ArrayList<User>(query.list());
148         } catch(Exception e){
149             LOGGER.error("Caught exception! Details: " + e.getMessage());
150             e.printStackTrace();
151             throw new Exception(e);
152         }
153     }
```

Εικόνα 16. Η μέθοδος getAllUsers του UserDAO

Γενικά, παρατηρούμε ότι η μέθοδος αποτελείται από έναν βρόγχο *try – catch*: μέσα στο *try*, ουσιαστικά, υπάρχει όλη η λογική με το *catch* απλά να χειρίζεται το *exception*.

```
Query query = sessionFactory.getCurrentSession().createQuery("from User u where u.isActive = true");
```

Εικόνα 17. Κατασκευή ενός αντικειμένου query

Όπως φαίνεται στην παραπάνω εικόνα (Εικόνα 17), αρχικά καλούμε την μέθοδο *getCurrentSession* του *sessionFactory*. Η μέθοδος αυτή μας επιστρέφει το αντικείμενο *Session*. Πιο συγκεκριμένα, ελέγχει αν υπάρχει ήδη κάποιο τέτοιο αντικείμενο και, αν δεν υπάρχει, δημιουργεί ένα καινούργιο και το επιστρέφει. Σε άλλη περίπτωση, επιστρέφει το υπάρχον.

Το αντικείμενο *Session* αποτελεί διεπαφή σε άλλες βιβλιοθήκες του Hibernate. Οι μέθοδοι που μας ενδιαφέρουν είναι οι ακόλουθες:

- createQuery: η μέθοδος αυτή παίρνει σαν όρισμα μια πρόταση (statement) *HQL* και επιστρέφει ένα αντικείμενο τύπου *Query*.
- createNativeQuery: η μέθοδος αυτή παίρνει σαν όρισμα ένα *SQL statement* και επιστρέφει ένα αντικείμενο τύπου *Query*.
- save: η μέθοδος αυτή παίρνει σαν όρισμα ένα αντικείμενο σημειωμένο με το annotation *@Entity* και δοκιμάζει να γράψει μια νέα εγγραφή στη βάση. Ελέγχει αν υπάρχει ψάχνοντας στον αντίστοιχο πίνακα της βάσης εγγραφή με τιμή στο *primary*

key όμοια με την τιμή του πεδίου που έχει σημειωθεί με το annotation *@id*. Αν βρεθεί εγγραφή, προκαλεί *exception*.

- saveOrUpdate: όπως η save με τη διαφορά ότι αν υπάρχει εγγραφή, την ανανεώνει με τα στοιχεία του ορίσματος.
- update: η μέθοδος αυτή παίρνει σαν όρισμα ένα αντικείμενο σημειωμένο με το annotation *@Entity* και δοκιμάζει να βρει εγγραφή με το *@id*. Αν βρει, ανανεώνει την εγγραφή με την πληροφορία που περιέχει το όρισμα. Αλλιώς, προκαλεί *exception*.
- delete: η μέθοδος αυτή παίρνει σαν όρισμα ένα αντικείμενο σημειωμένο με το annotation *@Entity* και δοκιμάζει να βρει εγγραφή με το *@id*. Αν βρει, διαγράφει την εγγραφή. Αλλιώς, προκαλεί *exception*.

Από τις παραπάνω μεθόδους, οι *save, saveOrUpdate, update* και *delete* ουσιαστικά παράγουν *SQL statements* με την βοήθεια των annotations πάνω στα αντικείμενα που τους δίνονται ως ορίσματα. Οι μέθοδοι createQuery και createNativeQuery, αντιθετα, δημιουργούν αντικείμενα τύπου *Query*.

Στο παράδειγμά μας, λοιπόν, χρησιμοποιούμε την μέθοδο *createQuery* με όρισμα ένα HQL statement:

```
"from User u where u.isActive = true"
```

Εικόνα 18. Μία πρόταση HQL

Το παραπάνω *HQL statement* είναι αντίστοιχο του εξής *SQL statement*:

```
"from SYS_USERS u where u.IS_ACTIVE = 1"
```

Εικόνα 19. Η αντίστοιχη πρόταση SQL

Τα αντικείμενα τύπου *Query* αποτελούν αναπαραστάσεις ενός SQL statement. Έχουν μεθόδους που θέτουν παραμέτρους μέσα στα statements αυτά καθώς και μεθόδους όπως η *uniqueResult*, η οποία επιστρέφει ένα μοναδικό αποτέλεσμα και προκαλεί *exception* αν βρει παραπάνω από ένα.

Στο δικό μας παράδειγμα, χρησιμοποιούμε τη μέθοδο *list* η οποία επιστρέφει μια λίστα με όλες τις εγγραφές που επέστρεψε το *HQL statement* και την οποία μετατρέπουμε σε λίστα από αντικείμενα τύπου *User.* Την λίστα αυτή την επιστρέφουμε στο *Service*.

Η περίπτωση της μεθόδου *getProjectById*, είναι λίγο διαφορετική:

```
130     @Override
        public Project getProjectById(int projectId) throws Exception {
132         try {
133             Query query = sessionFactory.getCurrentSession().createQuery("from Project p where p.projectId = :projectId");
134             query.setParameter("projectId", projectId);
135             return (Project)query.uniqueResult();
136         } catch(Exception e){
137             LOGGER.error("Caught exception! Details: " + e.getMessage());
138             e.printStackTrace();
139             throw new Exception(e);
140         }
141     }
```

Εικόνα 20. Η μέθοδος getProjectById

Στη μέθοδο αυτή θέλουμε να πάρουμε ένα συγκεκριμένο αντικείμενο τύπου *Project*. Για αυτόν τον λόγο, η μέθοδος έχει σαν όρισμα έναν ακέραιο ο οποίος είναι και η τιμή του κύριου κλειδιού του πίνακα.

Το *HQL statement*, στην περίπτωση αυτή, χρειάζεται επίσης μια παράμετρο, σημειωμένη με το λεκτικό «:projectId». Ο χαρακτήρας «:» υποδηλώνει την παράμετρο (αντίστοιχο με το «?» σε *SQL statements*) και τον ακολουθεί το όνομα της παραμέτρου. Για να θέσουμε τιμή στην παράμετρο αυτή, καλούμε την μέθοδο *setParameter* του αντικειμένου *Query*, η οποία δέχεται δύο ορίσματα: το όνομα της παραμέτρου και την τιμή.

Τέλος, για να πάρουμε μοναδικό αποτέλεσμα χρησιμοποιούμε την μέθοδο *uniqueResult*, που αναφέραμε πιο πάνω, και μετατρέπουμε το αποτέλεσμα της σε αντικείμενο τύπου *Project*.

3.3.2.6 Logging

Σε κάθε κλάση που περιέχει κάποια λογική έχουμε ορίσει δικό της ξεχωριστό logger, με την βοήθεια της βιβλιοθήκης Log4J. Για να γίνει αυτό, πρώτα πρέπει να έχουμε ορίσει τον Logger της κλάσης αυτής στο αρχείο log4j.properties:

```
30 log4j.logger.com.plato.dissertation.controller.UserController=DEBUG, UserControllerAppender
31 log4j.appender.UserControllerAppender=org.apache.log4j.RollingFileAppender
32 log4j.appender.UserControllerAppender.File=${catalina.home}/logs/dissertation/controller/UserController.log
33 log4j.appender.UserControllerAppender.MaxFileSize=400MB
34 log4j.appender.UserControllerAppender.MaxBackupIndex=7
35 log4j.appender.UserControllerAppender.layout=org.apache.log4j.PatternLayout
36 log4j.appender.UserControllerAppender.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p:(%M:%L) - %m%n
```

Εικόνα 21. Ορισμός logger στο αρχείο log4j.properties

Στην πρώτη γραμμή, ορίζεται ο *logger* του *UserController*. Στον ορισμό του δίνονται δύο στοιχεία: το επίπεδο, δηλαδή μέχρι και τι σημαντικότητας μηνύματα θα καταγράφονται, και το όνομα του *appender* του. Όσον αφορά στο επίπεδο, εμείς επιλέξαμε *DEBUG* που περιλαμβάνει όλα τα χρήσιμα μηνύματα.

Appenders είναι αντικείμενα που δηλώνουν τον προορισμό των μηνυμάτων και δέχονται παραμέτρους όπως μέγιστο μέγεθος αρχείου, μοτίβο καταγραφής κλπ. Εν συντομία, έχουμε ορίσει ως αρχείο καταγραφής το «/logs/dissertation/controller/UserController.log», με μέγιστο μέγεθος τα 400MB, να κρατάει τα 7 τελευταία αρχεία και να γράφει logs στην εξής μορφή:

%d{yyyy-MM-dd HH:mm:ss} %-5p:(%M:%L) - %m%n

Με τον χαρακτήρα «%» ξεκινάνε ιδιότητες του μοτίβου οι οποίες αντικαθίστανται με τιμές ως εξής:

- %d{μοτίβο}: γράφει την ημερομηνία που δημιουργήθηκε η εγγραφή με βάση το μοτίβο που του δίδεται
- %-5p: προτεραιότητα της εγγραφής, δηλαδή *DEBUG, ERROR, INFO, WARN* κλπ.
- %M: το όνομα της μεθόδου η οποία καταγράφει μήνυμα
- %L: η γραμμή της κλάσης στην οποία είναι η εντολή καταγραφής
- %m: το μήνυμα το οποίο παρέχει η εφαρμογή
- %n: αντικαθίσταται με τον χαρακτήρα αλλαγής γραμμής του συστήματος

Έχοντας ορίσει στο αρχείο log4j.properties τον *logger* για την συγκεκριμένη κλάση ή πακέτο, μπορούμε να το χρησιμοποίησουμε μέσα στον κώδικα, αφού ορίσουμε ένα αντικείμενο τύπου *Logger*:

```
23
24      private final static Logger LOGGER = Logger.getLogger(UserController.class.getName());
25
```

Εικόνα 22. Δημιουργία αντικειμένου Logger μέσα σε κλάση (UserControllerImpl.java)

Μέσω του αντικειμένου αυτού, μπορούμε να γράψουμε στο αρχείο καταγραφής με διάφορες προτεραιότητες χρησιμοποιώντα τις αντίστοιχες μεθόδους.

```
42          LOGGER.debug("Entered with parameters " + request.toString());
```

Εικόνα 23. Παράδειγμα χρήσης του αντικειμένου Logger για καταγραφή σε προτεραιότητα DEBUG

Στην παραπάνω εικόνα, βλέπουμε ένα παράδειγμα χρήσης του αντικειμένου *Logger* για καταγραφή με προτεραιότητα *DEBUG*. Για καταγραφή σε άλλες προτεραιότητες, υπάρχουν άλλες μέθοδοι όπως π.χ. η μεθοδος *error()* για την προτεραιότητα *ERROR*.

3.3.2.7 Κύκλος ζωής ενός αιτήματος

Έχοντας περιγράψει με συντομία τις διάφορες κλάσεις και τεχνικές οι οποίες αποτελούν το server-side τμήμα της εφαρμογής, θα παρουσιάσουμε τον κύκλο ζωής ενός αιτήματος. Σαν παράδειγμα, θα χρησιμοποιήσουμε το αίτημα σύνδεσης στην εφαρμογή.

Όπως είπαμε σε προηγούμενη ενότητα, η πρώτη σελίδα της εφαρμογής είναι το index.jsp όπου ο χρήστης δίνει τα στοιχεία σύνδεσης του. Το αίτημα που παράγεται έχει σαν URL το εξής:

http://domain:8080/Dissertation/user

Στο σώμα του αιτήματος θα υπάρχουν τρεις παράμετροι: η παράμετρος action, η οποία θα έχει το λεκτικό που υποδηλώνει τη σύνδεση με το σύστημα, η παράμετρος username, που θα περιέχει το όνομα του χρήστη ο οποίος προσπαθεί να συνδεθεί, και η παράμετρος password, δηλαδή το password που έδωσε ο χρήστης.

Το αίτημα αυτό θα το λάβει το *DispatcherServlet* και θα το προωθήσει στον *UserController*, διότι το URL περιέχει το λεκτικό «/user» το οποίο έχει αντιστοιχηθεί με αυτόν τον *REST Controller*. Ο *UserController* θα διαβάσει την παράμετρο action και θα καλέσει την μέθοδο *validateUserCredentials* του *UserService*.

Η μέθοδος αυτή θα πραγματοποιήσει τους παρακάτω ελέγχους:

1. Θα ελέγξει ότι όλα τα απαραίτητα στοιχεία έχουν δωθεί. Για το συγκεκριμένο αίτημα όλες οι παράμετροι είναι υποχρεωτικές.
2. Θα ελέγξει ότι στο σύστημα υπάρχει χρήστης με το username που δώθηκε στο αίτημα, χρησιμοποιώντας την μέθοδο *getUser* του *UserDAO*. Η μέθοδος αυτή παίρνει δύο ορίσματα, το username και μία *Boolean* τιμή που δείχνει αν θέλουμε να επιστρέψει μόνο ενεργούς χρήστες, και επιστρέφει την πληροφορία που υπάρχει στη βάση για τον χρήστη με username το πρώτο όρισμα σε μορφή αντικειμένου τύπου *User*. Αν δεν βρει πληροφορία, επιστρέφει κενό.

3. Θα ελέγξει ότι ο χρήστης αυτός είναι ενεργός τη δεδομένη στιγμή (Αυτό το αναφέρουμε ξεχωριστά αλλά ο έλεγχος στην πραγματικότητα γίνεται μέσω της μεθόδου *getUser* και, πιο συγκεκριμένα, του δεύτερου ορίσματος που της δίνουμε).
4. Θα ελέγξει ότι το password που έχει δωθεί στο αίτημα είναι ίδιο με το password της υπάρχουσας εγγραφής.

Αν τα στοιχεία που υπάρχουν πάνω στο αίτημα θεωρηθούν έγκυρα, τότε η μέθοδος αυτή επιστρέφει ένα αντικειμένο τύπου *ExtendedObject* του οποίου το πεδίο «object» έχει σαν τιμή ένα αντικείμενο τύπου *User* (συγκεκριμένα, αυτό που επέστρεψε η *getUser*). Αλλιώς, επιστρέφει ένα ίδιου τύπου αντικείμενο, με τη διαφορά ότι το πεδίο «object» θα είναι κενό ενώ το πεδίο «errorMessage» θα περιέχει ένα κατάλληλο μήνυμα λάθους.

Ο *UserController* θα διαβάσει το αντικείμενο που επέστρεψε το *UserService*, θα διαμορφώσει ένα αντικείμενο τύπου *ResponseEntity* και, αναλόγως με τα περιεχόμενα του, είτε θα θέσει στο σώμα της απάντησης μια αναπαράσταση του αντικειμένου *User* σε json είτε θα θέσει ένα μήνυμα σφάλματος.

3.3.3 Client-side

Όπως αναφέραμε σε προηγούμενη ενότητα, η βιβλιοθήκη Angular εισάγει το μοντέλο MVC στα client-side τμήματα των εφαρμογών. Συνεπώς, και στην δική μας περίπτωση, το τμήμα αυτό της εφαρμογής είναι οργανωμένο με τέτοιο τρόπο:

- Η βάση του client-side τμήματος της εφαρμογής είναι τρία αρχεία.jsp, τα οποία χρησιμεύουν σαν βάση για όλες τις οθόνες της εφαρμογής. Πάνω στα αρχεία αυτά συμπληρώνονται οι οθόνες με περαιτέρω αρχεία .html, τα οποία φορτώνονται ανάλογα με τις επιλογές του χρήστη. Αυτά και τα αρχεία .jsp αποτελούν τα *views*.
- Η λογική που χρειάζεται κάθε *view* για να λειτουργήσει, είτε είναι διαχείριση κάποιων τιμών είτε αίτηση για πληροφορίες από το *REST API*, υλοποιείται στους *angular controllers*. Κάθε *controller* έχει υλοποιηθεί για να διαχειρίζεται ένα *view*.
- Ό,τι λογική είναι πιο γενική ή ανεξάρτητη από τα *views* υλοποιείται σε συναρτήσεις σε *angular factories*. Τα αρχεία αυτά περιέχουν συναρτήσεις συνήθως σχετικές με κάποια συγκεκριμένα εργαλεία ή βιβλιοθήκες.
- Το αρχείο properties.js περιέχει σταθερές που χρησιμοποιούνται στην εφαρμογή.

Στις επόμενες ενότητες θα περιγράψουμε με συντομία τον ρόλο των διαφορετικών τύπων αρχείων καθώς και το πώς συνδέονται μεταξύ τους.

3.3.3.1 Angular module

Το *module* μιας εφαρμογής angular είναι ένας *container* όλων των αντικειμένων – *services, controllers, directives* – που ορίζονται για την εφαρμογή. Είναι ένα βασικό δομικό στοιχείο και είναι απαραίτητο για μια εφαρμογή που χρησιμοποιεί angular, άσχετα με το μέγεθος της.

Στην εικόνα που ακολουθεί φαίνεται ο ορισμός του *module* της εφαρμογής:

```
3   var App = angular.module('myApp',
4       [
5           'ngCookies',
6           'ngAnimate',
7           'ngMessages',
8           'ui.bootstrap',
9           'ui.select',
10          'ngSanitize'
11      ]);
```

Εικόνα 24. Ορισμός ενός module

Για να ορίσουμε ένα *module* καλούμε τη συνάρτηση *module* της angular με δύο ορίσματα: το όνομα του *module* και μία λίστα με τα ονόματα των *modules* από τα οποία εξαρτάται ή, αλλιώς, χρησιμοποιεί. Το αποτέλεσμα αποθηκεύεται στη μεταβλητή «App», η οποία υπάρχει σε όλη την εφαρμογή (global) και την χρησιμοποιούμε για να ορίσουμε άλλα στοιχεία.

3.3.3.2 Angular factories

Τα *angular factories* είναι *singletons* και, όπως αναφέραμε, υλοποιούν λογική ανεξάρτητη από κάποιο συγκεκριμένο *view*. Σαν οντότητες αποτελούνται από μία συνάρτηση που επιστρέφει (return) ένα αντικείμενο με διάφορα πεδία όπως άλλα αντικείμενα, javascript συναρτήσεις, απλές μεταβλητές και δικά τους πεδία, συναρτήσεις κ.ο.κ. αν έχουν.

Συνήθως, κάθε *factory* περιέχει πεδία σχετικά με μια συγκεκριμένη λογική ή βιβλιοθήκη. Παραδείγματος χάρην, το *CookieService*, το οποίο ορίζεται στο αρχείο cookie_service.js, παρέχει τρεις javascript συναρτήσεις: την *saveObject*, η οποία αποθηκεύει ένα αντικείμενο στα *cookies*, την retrieveObject, η οποία παίρνει ένα αντικείμενο από τα *cookies* και, τέλος, την *deleteObject*, η οποία σβήνει ένα αντικείμενο από τα *cookies*.

```
App.factory('CookieService', ['$cookies', 'Properties', function($cookies, Properties){
4       return {
5           saveObject: function(name, object){
6               $cookies.putObject(name, object);
7           },
8
9           retrieveObject: function(name){
10              return $cookies.getObject(name);
11          },
12
13          deleteObject: function(name){
14              $cookies.remove(name);
15          }
16      }
17  }]);
```

Εικόνα 25. Το CookieService factory

Όπως φαίνεται στην παραπάνω εικόνα, για να ορίσουμε ένα factory καλούμε τη συνάρτηση *factory* του *module*. Στην συνάρτηση αυτή, δίνουμε δύο ορίσματα: το πρώτο είναι το όνομα του *factory* ενώ το δεύτερο είναι μία λίστα javascript στην οποία πρώτα αναφέρουμε όλα τα άλλα αντικείμενα (*services, factories* κλπ.) τα οποία χρησιμοποιεί και μετά τη συνάρτηση ορισμού του. Η συνάρτηση ορισμού του έχει σαν ορίσματα τα υπόλοιπα αντικείμενα που δηλώσαμε στη λίστα και στο σώμα της την υλοποίηση του *factory*.

Στην εικόνα 24, το σώμα του *factory* αποτελείται μόνο από ένα *return statement* το οποίο επιστρέφει ένα αντικείμενο με τις τρεις συναρτήσεις που αναφέραμε προηγουμένως. Παρ'όλα αυτά, πριν το *return*, επιτρέπεται να ορίζονται και άλλα εσωτερικά πεδία και συναρτήσεις καθώς και να υπάρχει λογική. Όμως, επειδή ένα *factory* θα υπάρχει μία φορά στο *scope* της εφαρμογής, ο κώδικας πριν το *return* θα εκτελεστεί μία φορά μόνο όταν το *factory* θα πρωτοδημιουργηθεί.

Παρακάτω, παρατίθενται σύντομες περιγραφές των διαφόρων *factories* που χρησιμοποιούνται στην εφαρμογή:

UhttpService

Το *UhttpService* παρέχει συναρτήσεις για την πραγματοποίηση αιτημάτων προς το *REST*. Πιο συγκεκριμένα, έχει δικές του συναρτήσεις που πραγματοποιούν τις http κλήσεις και διαχειρίζονται τις απαντήσεις. Αυτές τις συναρτήσεις τις χρησιμοποιεί εσωτερικά στις συναρτήσεις που παρέχει προς τους *controllers* ή και σε όποιο *factory* το χρησιμοποιεί.

Για τις κλήσεις στο *REST* χρησιμοποιούμε την υπηρεσία *$http* της Angular.js. Η υπηρεσία αυτή παρέχει έτοιμες συναρτήσεις όπως *get, put, delete* κ.ο.κ. για την πραγματοποίηση κλήσεων ή την δυνατότητα δημιουργίας προσαρμοσμένων αιτημάτων. Είτε με τις έτοιμες συναρτήσεις είτε με προσαρμοσμένο αίτημα, η υπηρεσία αυτή επιστρέφει ένα αντικείμενο *promise*.

Το αντικείμενο *promise* αναπαριστά το τελικό αποτέλεσμα μια ασύγχρονης διαδικασίας και ο κύριος τρόπος αλληλεπίδρασης μαζί του είναι μέσω της συνάρτησης *then* η οποία εκτελείται όταν η ασύγχρονη διαδικασία τελειώσει. Η συνάρτηση αυτή παίρνει ως ορίσματα δύο συναρτήσεις: η πρώτη θα εκτελεστεί όταν και εφόσον η ασύγχρονη διαδικασία τελειώσει με επιτυχία ενώ η δεύτερη θα εκτελεστεί σε οποιαδήποτε περίπτωση αποτυχίας. Ο όρος *promise* προτάθηκε από τους Daniel Friedman και David Wise [4] ενώ η υλοποίηση της λογικής αυτής στην Angular.js παρέχεται από την υπηρεσία της *$q*.

Όπως φαίνεται στην παρακάτω εικόνα:

```
27      self.performAjaxCall = function (serviceUrl, serviceData) {
28          return $http({
29              url: serviceUrl,
30              method: Properties.CallParameters.pMethod,
31              headers: {
32                  'Content-Type': Properties.CallParameters.jsonContent
33              },
34              data: serviceData
35          }).then(
36              function (response) {
37                  return response.data;
38              },
39              function (errResponse) {
40                  console.error(Properties.Messages.ajaxCallError);
41                  return $q.reject(errResponse);
42              }
43          );
44      };
```

Εικόνα 26. Κλήση http προς το REST

Στην εφαρμογή μας χρησιμοποιούμε τον προσαρμοσμένο τρόπο να κάνουμε κλήσεις στο *REST API* της εφαρμογής. Χρησιμοποιώντας την εντολή *return $http()* και δίνοντας ένα αντικείμενο αιτήματος σαν όρισμα παράγουμε ένα αντικείμενο *promise* στου οποίου τη συνάρτηση *then* δίνουμε δύο συναρτήσεις ως ορίσματα: η πρώτη απλά επιστρέφει τα δεδομένα από την απάντηση – τα *headers* του *http response* δεν αφορούν τον *controller* ή οποιοδήποτε αντικείμενο την κάλεσε. Η δεύτερη συνάρτηση, η οποία εκτελείται σε περίπτωση σφάλματος, απλά επιστρέφει το σφάλμα στον *controller* ώστε να το διαχειριστεί όπως πρέπει.

CookieService

Το factory αυτό χρησιμοποιεί την υπηρεσία της Angular.js $cookies για να διαχειρίζεται τα cookies ανάλογα με τις ανάγκες της εφαρμογής. Συγκεκριμένα, υποστηρίζει τρεις λειτουργίες:

- saveObject: η συνάρτηση αυτή αποθηκεύει στα *cookies* το δεύτερο όρισμα της με όνομα το πρώτο όρισμα.
- retrieveObject: η συνάρτηση αυτή διαβάζει από τα *cookies* και επιστρέφει το αντικείμενο με όνομα το όρισμα της.
- deleteObject: η συνάρτηση αυτή σβήνει από τα *cookies* το αντικείμενο με όνομα το όρισμα της.

MessageService

Το *factory* αυτό περιέχει δύο συναρτήσεις οι οποίες πυροδοτούν *events* στέλνοντας τα ονόματα τους.

Για να περιγράψουμε περισσότερο το *factory* αυτό, πρέπει να περιγράψουμε λίγο τα *events* στην Angular.js. Όπως είχαμε αναφέρει σε προηγούμενη ενότητα, η Angular.js δημιουργεί μια ιεραρχική δομή από αντικείμενα *Scope* η οποία καθρεπτίζει την δομή του DOM, με την έννοια του ότι κάθε στοιχείο του DOM έχει και ένα δικό του αντικείμενο *Scope*, με αρχικό στοιχείο της δομής το *$rootScope*. Προφανώς, για δικές τις ανάγκες, η Angular.js και διάφορες υπηρεσίες ή αντικείμενα της προσθέτουν και άλλα τέτοια αντικείμενα στη δομή.

Κάθε αντικείμενο *Scope*, συμπεριλαμβανομένου και του *$rootScope*, έχει την λειτουργία *$on*. Ένα παράδειγμα φαίνετα στην παρακάτω εικόνα:

```
109     $rootScope.$on('logout', function(event, args){
110         self.logout();
111     });
```

Εικόνα 27. Παράδειγμα event listener στον angular controller UserController

Ο παραπάνω κώδικας ουσιαστικά παρακολουθεί ή, πιο σωστά, ακούει για το *event* με όνομα «logout», το πρώτο όρισμα του *$on*. Όταν αυτό το *event* πυροδοτηθεί, εκτελείται η συνάρτηση που του δίνεται σαν δεύτερο όρισμα.

Για να πυροδοτηθεί ένα *event*, πρέπει να κληθεί μία εκ των συναρτήσεων *$broadcast* και *$emit*. Η *$broadcast* στέλνει το όνομα ενός *event* καθώς και άλλη πληροφορία της επιλογής του προγραμματιστή αναδρομικά προς τα κάτω στη δομή των *Scopes*, δηλαδή στα παιδιά, στα παιδιά των παιδιών κλπ. Αντίθετα, η *$emit* στέλνει την ίδια πληροφορία προς τα πάνω.

Όταν εκτελείται οποιαδήποτε από τις δύο αυτές συναρτήσεις, όλοι οι *listeners* του κάθε *Scope* ειδοποιούνται και ελέγχονται με το όνομα που έχει δωθεί σαν όρισμα ξεκινώντας με τους *listeners* του *Scope* που τις εκτελεί.

Το *factory MessageService*, λοιπόν, απλά παρέχει συναρτήσεις προς την $*broadcast* και *$emit*, αντίστοιχα. Ο λόγος για τον οποίο δεν καλούμε κατευθείαν τις συναρτήσεις αυτές είναι ώστε, στην περίπτωση που θα πρέπει μελλοντικά να υπάρξει κάποια λογική πριν πυροδοτηθεί το *event*, να μην χρειάζεται να αλλάξουν όλα τα σημεία όπου καλούνται αυτές οι συναρτήσεις αλλά μόνο οι συναρτήσεις του *MessageService*.

UtilityService

Το *factory* αυτό περιέχει εργαλεία και συναρτήσεις που χρησιμοποιούνται σε διάφορα σημεία της εφαρμογής αλλά δεν έχουν κάποιο κοινό χαρακτηριστικό ή δεν υλοποιούν κάποια συγκεκριμένη λογική.

PopupService

Το *PopupService* παρέχει υλοποίηση και συναρτήσεις για παράθυρα *modal (popups)* με χρήση της εξωτερικής βιβλιοθήκης ui.modal. Εδώ να αναφέρουμε ότι, για λόγους συνοχής, κάποιοι *controllers* που χρειάζονται για τα παράθυρα αυτά έχουν οριστεί στο ίδιο αρχείο μιας και χρησιμοποιούνται μόνο για αυτόν τον σκοπό.

3.3.3.3 Angular Controllers

Οι *angular controllers* υλοποιούν τη λογική ενός *view*. Σε αντίθεση με τα *factories*, οι *controllers* δεν είναι *singletons*, γεγονός το οποίο σημαίνει ότι ένας *controller* μπορεί να υπάρχει παραπάνω από μία φορές.

Παρ'όλα αυτά, ο ορισμός τους είναι παρόμοιος με τα *factories*:

```
App.controller('UserController',
    ['$scope', '$rootScope', '$window', 'MessageService', 'UhttpService', 'CookieService', 'PopupService', 'Properties', 'UtilityService',
        function($scope, $rootScope, $window, MessageService, UhttpService, CookieService, PopupService, Properties, UtilityService){
```

Εικόνα 28. Ορισμός του angular controller UserController

Όπως φαίνεται στην Εικόνα 26, για να ορίσουμε έναν *angular controller* καλούμε την συνάρτηση *controller* του *module* μας (App) με ορίσματα το όνομα του *controller* και μία λίστα με τα ονόματα όλων των αντικειμένων που χρησιμοποιεί και, στο τέλος, μια μέθοδο με ορίσματα τα αντικείμενα αυτά.

Στο σημείο αυτό, όμως, οι ομοιότητες των *controllers* και τον *factories* τελειώνουν. Από τη στιγμή που οι *controllers* δεν είναι *singletons*, δεν μπορούν να χρησιμοποιηθούν από άλλους *controllers* ή *factories*. Για αυτό το λόγο και δεν υπάρχει κάποιο *return statement* στην συνάρτηση ορισμού τους. Το αποτέλεσμα είναι ότι ένας *controller* ορίζει ένα *Scope* (περιγράφηκε σε προηγούμενη ενότητα) με λογική, πληροφορία και συμπεριφορά.

Στο σώμα, δηλαδή μέσα στη συνάρτηση ορισμού του *controller*, λοιπόν, ορίζονται μεταβλητές, συναρτήσεις ή μπορεί να υπάρχουν και εντολές που θα εκτελούνται κατά τη δημιουργία ενός στιγμιότυπου (instance) του *controller*. Τεχνικά, και ο ορισμός μεταβλητών και συναρτήσεων

είναι τέτοιες εντολές. Το πώς αξιοποιούνται οι εντολές αυτές θα περιγραφεί στην επόμενη ενότητα.

3.3.3.4 Views

Με τον όρο *views* αναφερόμαστε στα .jsp και .html αρχεία τα οποία αποτελούν τις όψεις της εφαρμογής. Όπως προαναφέραμε, δύο είναι οι βασικές όψεις της εφαρμογής: το *index*, το οποίο περιέχεται στο index.jsp, και στο *dashboard*, το οποίο περιέχεται στο dashboard.jsp. Το includes.jsp περιέχει τις εισαγωγές των διαφόρων βιβλιοθηκών και λοιπών κοινών αρχείων που χρειάζονται τα άλλα δύο .jsp αρχεία.

Η όψη *index* περιέχει μόνο τη σελίδα σύνδεσης του χρήστη. Ο χρήστης δίνει τα στοιχεία του και, αν αυτά είναι έγκυρα, τότε μεταβαίνει στην όψη *dashboard* η οποία είναι και η κύρια όψη της εφαρμογής. Η όψη αυτή περιέχει το βασικό μενού και, ανάλογα με τις επιλογές του χρήστη, φορτώνει κάποιο εκ των επιμέρους .html αρχείων, τα οποία περιέχουν πιο συγκεκριμένες όψεις. Για παράδειγμα, το αρχείο about.html περιέχει πληροφορίες για την εφαρμογή.

Από το HTML μιας όψης δεν μπορούμε να έχουμε πρόσβαση απευθείας σε κάποιον *controller* ή *factory*. Για να γίνει αυτό αλλά και για να ενισχυθεί η HTML συνολικά με την βοήθεια της Angular.js, υπάρχουν τα *directives.* Όπως αναφέραμε σε προηγούμενη ενότητα, τα *directives* είναι ειδική κατηγορία στοιχείων της Angular.js η οποία μεταφράζεται σε ετικέτες HTML (html tags) ή σε χαρακτηριστικά (attributes). Αυτά, χρησιμοποιούνται κατά κόρον σε εφαρμογές Angular.js καθώς μπορούν να αλλάξουν δραστικά την συμπεριφορά της HTML ακόμα και χωρίς τη χρήση *angular controllers*.

Για να ενεργοποιηθεί, λοιπόν, η Angular.js σε μία σελίδα, αρχικά πρέπει να χρησιμοποιήσουμε το *directive ng-app*.

```
18    <body ng-app="myApp">
19        <...26 lines />
45        <c:import url="includes.jsp" />
46    </body>
```

Εικόνα 29. Χρήση της οδηγίας ng-app στο index.jsp

Δηλώνοντας το όνομα του *angular module* στο *directive ng-app*, μπορούμε να χρησιμοποιήσουμε στην εφαρμογή μας τα αντικείμενα που έχουμε ορίσει σε αυτό το *module*. Αλλά από μόνο του, το *directive* αυτό δεν προσφέρει λογική στην εφαρμογή, παρά σε πολύ περιορισμένο βαθμό.

Όπως αναφέραμε προηγουμένως, ένα *view* πάντα έχει κάποιον *angular controller* ώστε να έχει πρόσβαση σε δυναμικά στοιχεία και λογική. Για να χρησιμοποιήσουμε έναν *controller* σε κάποιο *view*, χρησιμοποιούμε το *directive ng-controller*:

```
19    <div class="container text-center" id="root_container" ng-controller="UserController as ctrl">
20        <...24 lines />
44    </div>
```

Εικόνα 30. Χρήση του ng-controller για δέσιμο ενός view με ένα angular controller

Εδώ η σύνταξη διαφέρει λίγο από προηγουμένως, με την έννοια ότι αντί να δώσουμε μόνο το όνομα του *controller*, στην περίπτωση μας *UserController*, δίνουμε μια έκφραση. Η έκφραση αυτή ουσιαστικά θέτει ένα αντιπροσωπευτικό όνομα για τον *controller* (alias), με το οποίο η

εφαρμογή μπορεί να αναφέρεται σε αυτό. Παραδείγματος χάρην, αν θέλουμε να αποκτήσουμε πρόσβαση στο πεδίο «a» του *UserController*, μπορούμε απλά να πούμε «ctrl.a».

Το *directive ng-controller* δίνει πρόσβαση στα στοιχεία του *controller* από την HTML ή άλλα *directives*. Επίσης, όταν εκτελείται το *directive ng-controller*, δημιουργείται ένα στιγμιότυπο του *controller* και εκτελούνται όλες οι εντολές που βρίσκονται στο σώμα του.

```
<label class="sr-only">Username</label>
<input type="text" name="username" class="form-control" placeholder="Username" ng-model="ctrl.credentials.username" required autofocus>
```

Εικόνα 31. Χρήση του directive ng-model

Στην Εικόνα 31 βλέπουμε το πεδίο στο οποίο εισάγει ο χρήστης το όνομα χρήστη του. Ουσιαστικά έχουμε μία ετικέτα (label) η οποία δηλώνει στο χρήστη τι βλέπει και ένα πλαίσιο εισαγωγής κειμένου. Πέρα από τα κλασσικά χαρακτηριστικά της HTML βλέπουμε και ένα *directive*, το *ng-model*.

Το *ng-model* τοποθετείται σε HTML ετικέτες οι οποίες έχουν κάποια τιμή όπως π.χ. οι ετικέτες *input*. Αυτό που κάνει είναι να δένει την τιμή της HTML ετικέτας (δηλαδή το χαρακτηριστικό *value* μιας HTML ετικέτας *input*) με την μεταβλητή που ορίζει ο προγραμματιστής μέσα. Το «δέσιμο» αυτό είναι αμφίδρομο (two-way binding), με την έννοια του ότι αν αλλάξει η τιμή της μεταβλητής, η αλλαγή θα φανεί σε πραγματικό χρόνο και στην ετικέτα HTML.

Στο δικό μας παράδειγμα δένουμε την τιμή του *input* με όνομα «username» στην μεταβλητή *ctrl.credentials.username*. Αρχικά, η μεταβλητή αυτή ανήκει στον *controller* γιατί ξεκινάει με το λεκτικό *ctrl*. Άρα, η τιμή του *input* «username» δένεται στο πεδίο «username» του αντικειμένου «credentials» του *UserController*.

```
5     self.credentials = {username:'',password:'', passwordRet:''};
```

Εικόνα 32. Το javascript αντικείμενο credentials όπως ορίζεται στο σώμα του UserController

Αντίστοιχα, όπως μπορούμε να έχουμε πρόσβαση σε αντικείμενα ορισμένα σε κάποιον *controller*, μπορούμε να έχουμε πρόσβαση αλλά και να εκτελέσουμε συναρτήσεις ορισμένες σε κάποιον *controller*.

Για παράδειγμα, όταν ο χρήστης πατάει το κουμπί «Sign in» για να δοκιμάσει να συνδεθεί αφού έχει συμπληρώσει τα στοιχεία του στη φόρμα, εκτελείται η συνάρτηση *login* του *UserController*:

```
43          <button class="btn btn-lg btn-primary btn-block"
44                  ng-click="ctrl.login(login_form.$valid, login_form.$dirty);login_form.$setPristine();">Sign in</button>
```

Εικόνα 33. Ορισμός του κουμπιού "Sign in" της εφαρμογής

Εδώ χρησιμοποιούμε το *directive ng-click*, το οποίο συμπεριφέρεται με τον ίδιο τρόπο όπως το αντίστοιχο του *onClick*. Η διαφορά είναι ότι η *onClick* του DOM δεν έχει πρόσβαση σε δομές της Angular.js οπότε σε εφαρμογές που την χρησιμοποιούμε προτιμάται το *ng-click*. Στο *ng-click* δίνεται σαν όρισμα μια έκφραση ακριβώς με τον ίδιο τρόπο όπως θα γραφόταν και σε ένα αρχείο javascript.

Τα *$valid*, *$dirty* και *$setPristine* είναι πεδία και μέθοδοι που ανήκουν στο *scope* των στοιχείων της HTML. Μέσω *directives* είναι ο μόνος τρόπος που μπορούμε να διαβάσουμε και να περάσουμε σε *controller* τις τιμές και τα πεδία αυτά. Αυτό είναι και ένα ακόμα προτέρημα της λογικής αυτής.

3.3.3.5 ContentController

Εξαίρεση του κανόνα αποτελεί ο *ContentController* ο οποίος χρησιμοποιείται στο dashboard.jsp. Ο *controller* αυτός δεν έχει να κάνει με κανένα *view*. Ουσιαστικά, όταν δημιουργείται ένα στιγμιότυπό του ζητάει από το *REST API* κάποιους πόρους που χρησιμοποιούνται πολύ συχνά και βρίσκονται στη βάση και τους αποθηκεύει στα *cookies*, ώστε να μην χρειάζεται να γίνονται ξεχωριστές κλήσεις αργότερα για να τους λαμβάνουμε.

3.3.4 Η βάση δεδομένων

Το πιο βασικό κομμάτι μιας εφαρμογής είναι η πληροφορία που διαχειρίζεται. Όλη η λογική βασίζεται και στρέφεται γύρω από την υλοποίηση της βάσης δεδομένων. Η λύση που έχουμε προτείνει για το υπό μελέτη πρόβλημα είναι η ενσωμάτωση ενός SLA στην υλοποίηση του συστήματος. Αυτό, όμως, δεν θα το καταφέρουμε μέσω κώδικά.

Το SLA ορίζει διαδικασίες αλλά ορίζει και οντότητες, μεγέθη και σταθερές. Και αυτά τα στοιχεία μπορούν να αποτυπωθούν μόνο στη βάση ενώ οι διαδικασίες θα υλοποιηθούν μέσω ενός συνδυασμού κώδικα και παραμετροποίησης. Εμείς στοχεύσαμε σε μια πιο ελεύθερη υλοποίηση σε κώδικα, ώστε η εφαρμογή να είναι όσο πιο γενική είναι δυνατό.

Για τον λόγο αυτό έχουμε κάνει την υπόθεση ότι τα μεγέθη που ορίζονται στο εκάστοτε SLA δεν θα αλλάζουν συχνά οπότε θεωρούνται τμήμα της εγκατάστασης της εφαρμογής. Δεν θα υποστηρίζεται οθόνη με διαχείριση των στοιχείων αυτών. Για την περίπτωση, όμως, που γίνουν αλλαγές, τα βασικά στοιχεία θα αποτυπώνονται ως πληροφορία στη βάση με αποτέλεσμα να είναι εύκολη η αλλαγή τους.

Όπως αναφέρθηκε παραπάνω, στην εφαρμογή χρησιμοποιήσαμε τον MySQL Community Server. Ο ορισμός των πινάκων και του σχήματος της βάσης περιλαμβάνεται σε τέσσερα αρχεία sql: *schema_creation.sql, create_billing_tables.sql, create_sys_tables.sql* και *create_ticket_tables.sql*. Το πέμπτο αρχείο, *populate_reference_tables.sql*, περιέχει την αρχικοποίηση κάποιων από τους πίνακες με αρχική πληροφορία και, πιο συγκεκριμένα, την πληροφορία που περιμένουμε να ορίζεται ή να βασίζεται σε ένα SLA.

Παρακάτω παρουσιάζουμε τους βασικούς πίνακες και τους ορισμούς τους.

Σύμφωνα με τις απαιτήσεις που περιγράψαμε σε προηγούμενη ενότητα, οι αιτήσεις υποστήριξης θα έχουν ένα χαρακτηριστικό που θα ονομάζεται προτεραιότητα. Το χαρακτηριστικό αυτό θα ορίζεται στη βάση ως ένας πίνακας «TIC_TICKET_PRIORITY» με τις εξής κολώνες:

- «PRIORITY_ID», το κύριο κλειδί του πίνακα
- «PRIORITY_DES», το λεκτικό της προτεραιότητας, όπως π.χ. Υψηλή
- «TARGET_TIME», αριθμός ημερών μέσα στις οποίες πρέπει η αίτηση να έχει διευθετηθεί

Ο πίνακας «TIC_FAULT_TYPE» περιέχει πληροφορία για τις κατηγορίες σφαλμάτων. Οι κολώνες που ορίζονται είναι οι εξής:

- «FAULT_TYPE_ID», το κύριο κλειδί του πίνακα
- «FAULT_TYPE_DES», η περιγραφή της κατηγορίας

- «SCORE_MODIFIER», το κύριο κλειδί του πίνακα «TIC_TICKET_PRIORITY» που αντιστοιχεί στην προτεραιότητα με την οποία συνδέεται η κάθε κατηγορία
- «IS_INITIAL», δείκτης που δείχνει αν αυτή η κατηγορία μπορεί να επιλεχθεί κατά την αρχική υποβολή της αίτησης.

Αντίστοιχα ορίζονται και οι πίνακες «BIL_CUSTOMER_TYPES» και «TIC_AFFECTED_CASES_CATEGORIES» για τους τύπους πελάτη και τις κατηγορίες πλήθους «κρουσμάτων» αντίστοιχα, με τη διαφορά ότι δεν χρειάζονται κολώνα «IS_INITIAL» διότι τίθενται μόνο μια φορά κατά την αρχική υποβολή της αίτησης.

Όσον αφορά στους τύπους πελατών, αναφέραμε και πιο πάνω ότι επιτρέπουμε στον χρήστη να εισάγει συγκεκριμένο πελάτη αντί για κάποιον από τους υπάρχοντες τύπους. Η πληροφορία πελατών υπάρχει στον πίνακα «BIL_CUSTOMER», με τις παρακάτω κολώνες:

- «CUSTOMER_ID», κύριο κλειδί του πίνακα
- «CUSTOMER_NAME», ονομασία του πελάτη
- «CUSTOMER_TYPE», εξωτερικό κλειδί στον πίνακα «BIL_CUSTOMER_TYPES»
- «CUSTOMER_SCORE», αριθμητικό που αντιστοιχεί στο κύριο κλειδί του πίνακα «TIC_TICKET_PRIORITY»

Όσον αφορά στις κατηγορίες αριθμού «κρουσμάτων», ο χρήστης έχει την επιλογή να εισάγει ακριβή αριθμό. Όπως και το SLA, έτσι και η εφαρμογή πρέπει να το διαχειρίζεται αυτό. Στην δικιά μας περίπτωση, έχουμε ορίσει έναν ακόμα πίνακα «TIC_AFFECTED_CASES_CONFIGURATION» με τις εξής κολώνες:

- «CONFIGURATION_ID», κύριο κλειδί του πίνακα
- «CASES_FROM» και «CASES_TO» είναι κολώνες που θέτουν ένα πάνω και κάτω κατώφλι πλήθους στο οποίο αντιστοιχείται μια προτεραιότητα
- «SCORE_MODIFIER», τιμή που αντιστοιχεί σε κύριο κλειδί του πίνακα «TIC_TICKET_PRIORITY» και εκφράζει την προτεραιότητα που αντιστοιχεί στα πλήθη που ορίζονται από τις παραπάνω κολώνες

Στη βάση δεδομένων, επίσης, υπάρχουν καταχωρημένες οι καταστάσεις μια αίτησης στο πίνακα «TIC_TICKET_STATE», με δύο κολώνες για το κύριο κλειδί και το λεκτικό της, καθώς και οι έγκυρες μεταβάσεις από κατάσταση σε κατάσταση στον πίνακα «TIC_STATE_TRANSITIONS».

Ο πίνακας «TIC_STATE_TRANSITIONS» δομείται ως εξής:

- «ID», το κύριο κλειδί του πίνακα
- «STATE_FROM», εξωτερικό κλειδί στον «TIC_TICKET_STATE» που υποδηλώνει την παρούσα κατάσταση της αίτησης
- «STATE_TO», εξωτερικό κλειδί στον «TIC_TICKET_STATE» που υποδηλώνει την κατάσταση στην οποία θέλουμε να μεταβεί η κατάσταση
- «GROUP_VISIBILITY_TYPE», ορίζει τον τύπο ομάδας που μπορεί να εκτελέσει τη μετάβαση αυτή

- «REQUIRES_ASSIGNMENT», ορίζει το κατά πόσο πρέπει η αίτηση να έχει ανατεθεί σε χρήστη
- «REQUIRES_SUBMITTER», ορίζει το κατά πόσο η μετάβαση αυτή επιτρέπεται μόνο από χρήστες *ticket submitters*
- «REQUIRES_GROUP_ASSIGNMENT», ορίζει το κατά πόσο χρειάζεται να έχει ανατεθεί η αίτηση σε *support* ομάδα

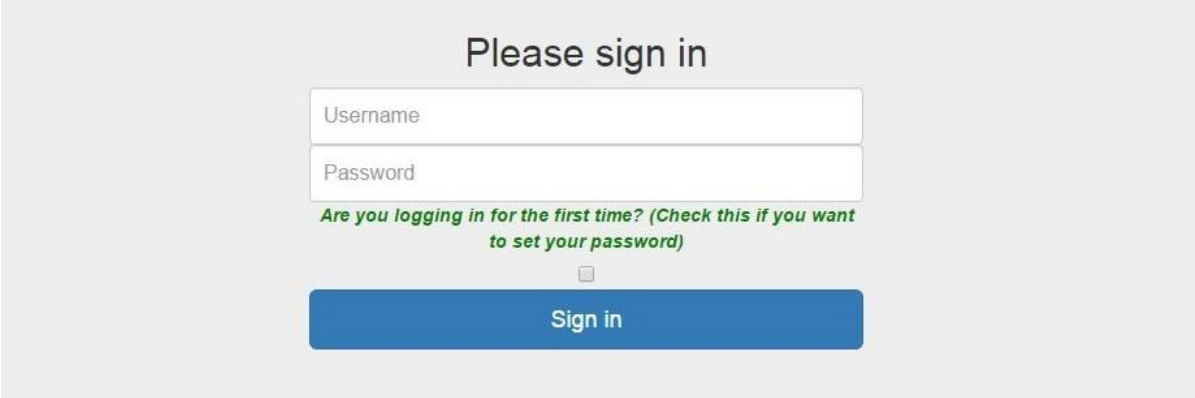
Ο πίνακας «TIC_STATE_TRANSITIONS» έχει σαν ρόλο να ορίζει τον κύκλο ζωής μιας αίτησης και, κατ'επέκταση, αποτελεί την υλοποίηση των διαδικασιών υποστήριξης της εφαρμογής.

4 Παρουσίαση της SLASUP

Στην ενότητα αυτή θα παρουσιάσουμε την SLASUP και τη λειτουργία της. Η ενότητα είναι χωρισμένη σε υποενότητες, ανάλογα με τα θεματικά τμήματα της εφαρμογής.

Σύνδεση στην εφαρμογή

Πρώτη όψη της εφαρμογής είναι η φόρμα σύνδεσης όπου ο χρήστης καλείται να συμπληρώσει τα στοιχεία του:



Εικόνα 34. Φόρμα σύνδεσης στην εφαρμογή

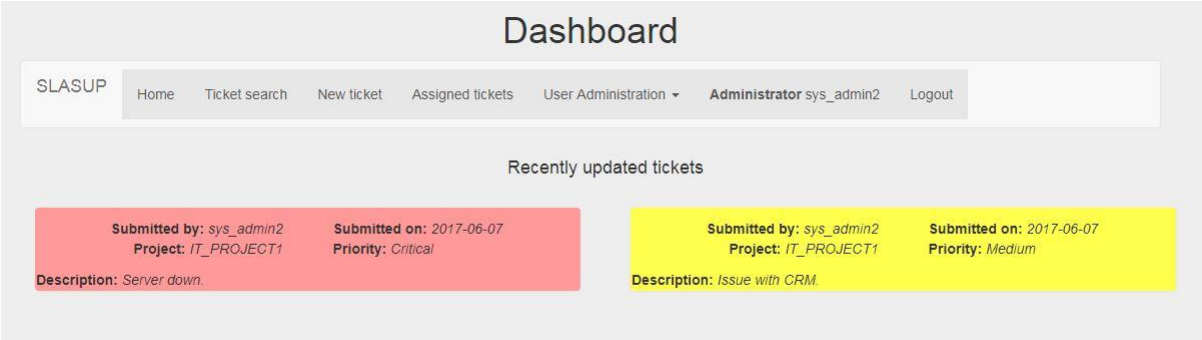
Στο πεδίο «Username» ο χρήστης συμπληρώνει το όνομα χρήστη του ενώ στο «Password» τον μυστικό κωδικό του. Το κουτάκι κάτω από τα πράσινα γράμματα αναφέρεται σε χρήστες που συνδέονται για πρώτη φορά στην εφαρμογή. Η λειτουργία του θα εξηγηθεί παρακάτω στην ενότητα «Δημιουργία νέου χρήστη».

Έπειτα, ο χρήστης πατάει το κουμπί «Sign in», και αν τα στοιχεία που εισήγαγε είναι σωστά, μεταφέρεται στην κύρια όψη της εφαρμογής.

Dashboard

Η όψη «Dashboard» είναι η κύρια όψη της εφαρμογής και, όπως φαίνεται στην Εικόνα 36, χωρίζεται σε δύο βασικά τμήματα: το βασικό μενού (Εικόνα 35) και την υπόλοιπη σελίδα. Να σημειωθεί ότι το «Dashboard» δεν αποτελεί όψη αλλά ένα πλαίσιο μέσα στο οποίο φορτώνονται οι υπόλοιπες όψεις και παρέχει και συνεχή πρόσβαση στο μενού. Για παράδειγμα, όταν συνδέεται ένας χρήστης, μεταφέρεται αυτόματα στην όψη «Home», η οποία φαίνεται στην Εικόνα 36.

Εικόνα 35. Βασικό μενού της εφαρμογής



Εικόνα 36. Dashboard χρήστη που ανήκει σε ομάδα ticket submitters

Για λόγους πληρότητας, συνδεθήκαμε στην εφαρμογή με τον χρήστη «sys_admin2», ο οποίος είναι χρήστης ομάδας *ticket submitters* με δικαιώματα διαχείριστη. Το παραπάνω μενού είναι πανομοιότυπο, παρ'όλα αυτά, για όλους τους χρήστης με ελάχιστες εξαιρέσεις. Παρακάτω, εξηγούμε τι σημαίνει το κάθε αντικείμενο του μενού:

- Home: είναι η βασική όψη της εφαρμογής και δείχνει τις πιο πρόσφατες αιτήσεις – είτε αυτές που υποβλήθηκαν πιο πρόσφατα είτε αιτήσεις που αλλάξανε κατάσταση πιο πρόσφατα.
- Ticket search: στην όψη αυτή παρέχεται μια διεπαφή για αναζήτηση αιτήσεων.
- New ticket: στην όψη αυτή ο χρήστης μπορεί να υποβάλει νέα αίτηση υποστήριξης. Η επιλογή αυτή υπάρχει μόνο για χρήστες που ανήκουν σε ομάδα *ticket submitters*.
- Assigned tickets: όψη που περέχει λίστα με όλες τις αιτήσεις που είναι αυτή τη στιγμή ανατεθειμένες στην συγκεκριμένη ομάδα ή/και χρήστη.
- User Administration: Υπομενού με τις ακόλουθες επιλογές διαχείρισης χρηστών:
    - User details: Όψη η οποία επιτρέπει στον χρήστη την αλλαγή ενός ή παραπάνω από τα στοιχεία του.
    - New user: διαθέσιμη μόνο για διαχειριστές, η όψη αυτή επιτρέπει στον χρήστη να δημιουργήσει έναν νέο χρήστη για την ομάδα του.
    - Delete user: η όψη αυτή δίνει τη δυνατότητα στους διαχειριστές να απενεργοποιήσουν κάποιον χρήστη.
    - Modify a user: η όψη αυτή επιτρέπει την επεξεργασία και αλλαγή ενός ή παραπάνω στοιχείων κάποιου άλλου χρήστη. Επιτρέπει, επίσης, την επαναενεργοποίηση ενός χρήστη. Είναι διαθέσιμη μόνο στους διαχειριστές.
- Logout: Αποσύνδεση από την εφαρμογή

Home

Η όψη «Home» είναι η αρχική σελίδα κάθε χρήστη της εφαρμογής και, όπως προαναφέραμε, παρέχει μια λίστα από τις πιο πρόσφατες αιτήσεις:

Εικόνα 37. Λίστα από πρόσφατες αιτήσεις

Οι αιτήσεις της λίστας εμφανίζονται σαν πλαίσια χρωματισμένα κατάλληλα ανάλογα με τη σοβαρότητα της αίτησης. Για παράδειγμα, στην Εικόνα 37 βλέπουμε μία αίτηση ύψιστης προτεραιότητας επισημασμένη με το σκούρο κόκκινο χρώμα, μία αίτηση μέτριας προτεραιότητας με κίτρινο χρώμα και μία πληροφοριακή αίτηση με πράσινο χρώμα.

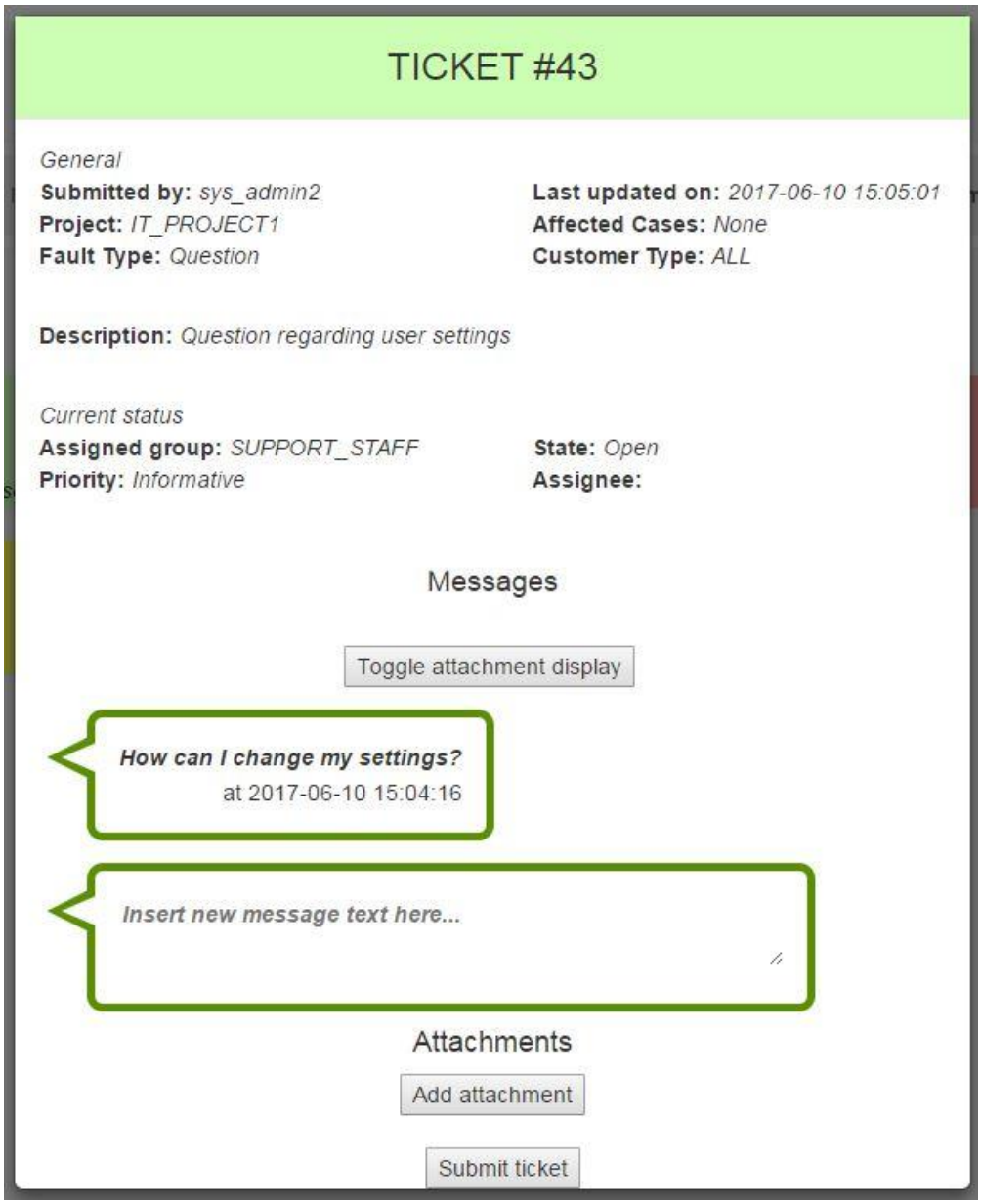
Η πληροφορία που περιέχουν τα στιγμιότυπα αυτά είναι η εξής:

- Το όνομα του χρήστη που υπέβαλλε την αίτηση
- Η ημερομηνία υποβολής
- Το έργο (project) στο οποίο αναφέρεται
- Η προτεραιότητα
- Η περιγραφή της αίτησης

Ο χρήστης μπορεί να δει περισσότερες επιλογές για μια συγκεκριμένη αίτηση πατώντας πάνω σε ένα στιγμιότυπο. Με τον τρόπο αυτό ανοίγει ένα αναδυόμενο παράθυρο με περισσότερες λεπτομέρειες για την αίτηση.

Στην Εικόνα 38 φαίνεται ένα αναδυόμενο παράθυρο με τα πλήρη στοιχεία της αίτησης. Βλέπουμε ότι το παράθυρο αυτό χωρίζεται σε τρία τμήματα. Το πρώτο τμήμα, το οποίο αναγράφεται ως «General», έχει την σταθερή πληροφορία της αίτησης:

- Όνομα χρήστη του χρήστη που υπέβαλλε την αίτηση
- Ημερομηνία τελευταίας αλλαγής της αίτησης
- Το έργο στο οποίο αναφέρεται η αίτηση
- Η κατηγορία ή ο αριθμός, αν είχε δωθεί, χρηστών που επηρεάζονται
- Κατηγορία σφάλματος
- Τύπος πελάτη ή ονομασία πελάτη, αν επιλέχθηκε συγκεκριμένος πελάτης
- Περιγραφή του σφάλματος ή της ερώτησης



Εικόνα 38. Αναδυόμενο παράθυρο αίτησης υποστήριξης

Το επόμενο τμήμα (Current status) είναι τα στοιχεία που αφορούν στην συγκεκριμένη κατάσταση της αίτησης:

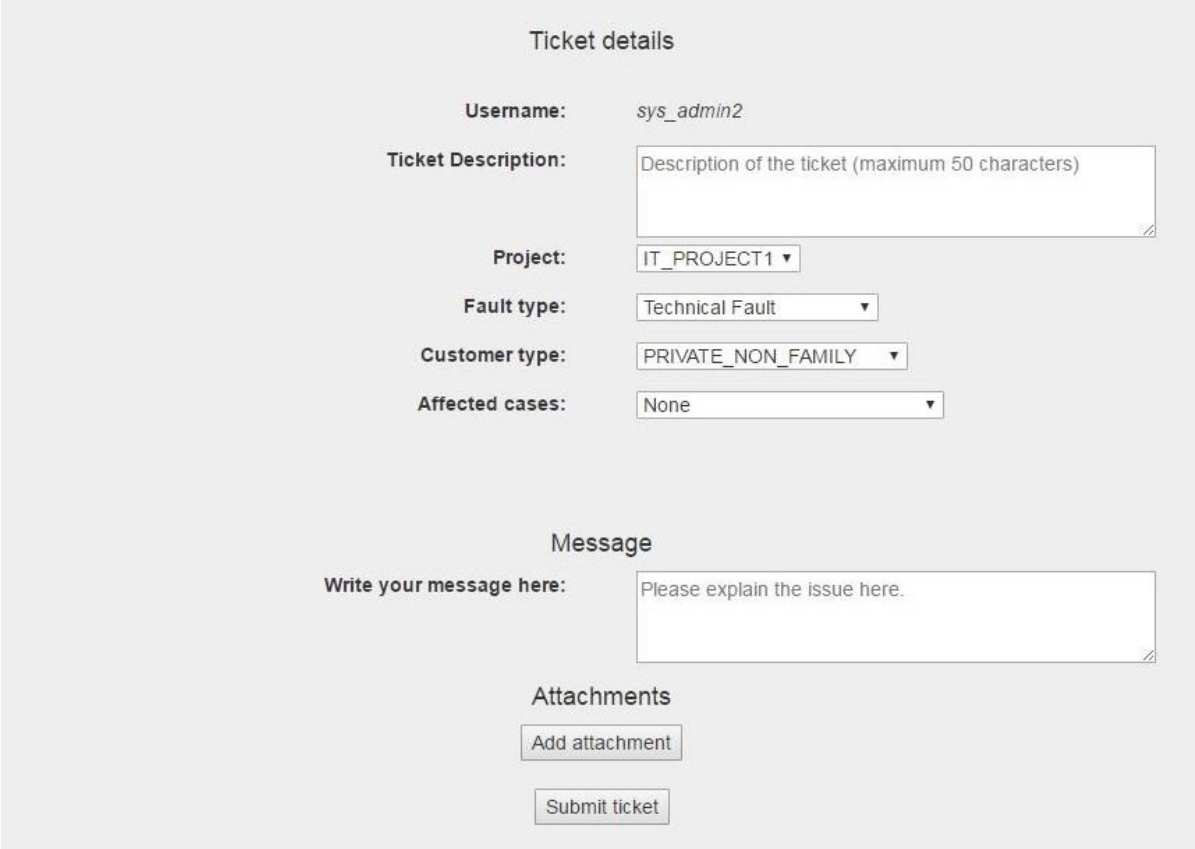
- Η ομάδα στην οποία είναι ανατεθειμένη η αίτηση αυτή τη στιγμή
- Η κατάσταση της αίτησης
- Η προτεραιότητα της αίτησης (μπορεί να αλλάξει από την ομάδα *support*)
- Ο χρήστης στον οποίο έχει ανατεθεί η αίτηση

Τέλος, υπάρχει το τμήμα των μηνυμάτων, στο οποίο αναγράφονται τα μηνύματα μαζί με την ημερομηνία και ώρα που γράφτηκαν. Το κουμπί «Toggle attachment display» εμφανίζει τυχόν συνημμένα σε κάποιο μήνυμα αρχεία.

Κάθε χρήστης μπορεί οποιαδήποτε στιγμή να προσθέσει κάποιο μήνυμα ή/και συνημμένο αρχείο σε μία ενεργή αίτηση. Όμως, δεν μπορεί να αλλάξει την κατάσταση της παρά μόνο αν έχει ανατεθεί στην ομάδα του και, ακόμα και τότε, υπάρχουν κάποιες προϋποθέσεις που πρέπει να πληρούνται.

New ticket

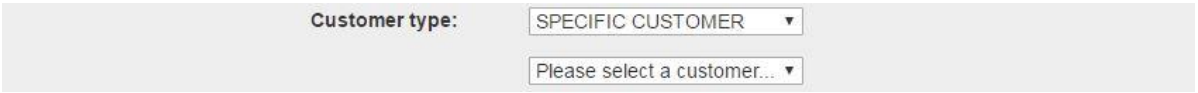
Στην όψη αυτή, ένας χρήστης που ανήκει σε ομάδα *ticket submitters* μπορεί να υποβάλλει μια νέα αίτηση:



Εικόνα 39. Όψη υποβολής νέας αίτησης

Όπως αναφέραμε και στην ενότητα 4.1, ο χρήστης στην αρχή της διαδικασίας μπορεί να συμπληρώσει συγκεκριμένα πεδία, τα οποία φαίνονται παραπάνω. Η προτεραιότητα δεν τίθεται από τον χρήστη σε αυτό το βήμα αλλά παράγεται δυναμικά από την εφαρμογή με βάση τα υπόλοιπα στοιχεία (πιο συγκεκριμένα τον τύπο σφάλματος, τον τύπο πελάτη και τον αριθμό των πελατών που επηρεάζονται).

Ο χρήστης έχει την επιλογή να επιλέξει μια κατηγορία πελάτη ή να επιλέξει συγκεκριμένο πελάτη. Αυτό γίνεται μέσω της επιλογής «SPECIFIC CUSTOMER», η οποία εμφανίζει ένα ακόμα *drop-down* κάτω από το υπάρχον με όλους τους πελάτες:

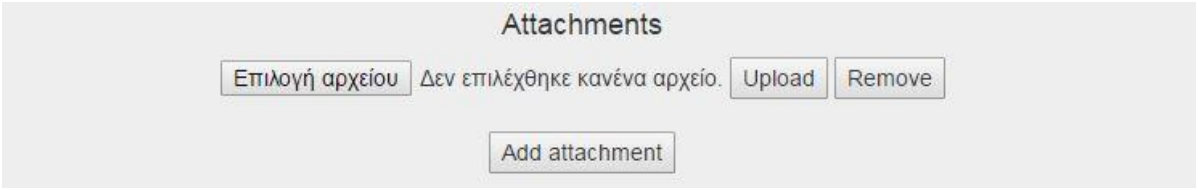


Εικόνα 40. Επιλογή συγκεκριμένου πελάτη

Αντίστοιχα, ο χρήστης έχει την επιλογή να γράψει συγκεκριμένο αριθμό «κρουσμάτων» αντί να επιλέξει μια κατηγορία. Για να γίνει αυτό, μπορεί να επιλέξει «Number inserted» και να γράψει ένα νούμερο:

Εικόνα 41. Εισαγωγή αριθμού κρουσμάτων

Τέλος, ο χρήστης έχει τη δυνατότητα να συνάψει μέχρι και 5 αρχεία. Για να προσθέσει ένα αρχείο, πρέπει πρώτα να πατήσει το κουμπί «Add attachment».



Εικόνα 42. Προσθήκη συνημμένων αρχείων σε μήνυμα

Στη συνέχεια, προσθέτει το αρχείο με το κουμπί «Επιλογή αρχείου» και το ανεβάζει με το κουμπί «Upload».

Επειδή η βάση μας είναι σχεσιακή βάση δεδομένων, αποφασίσαμε να υλοποιήσουμε την διαχείριση αρχείων με τον εξής τρόπο. Όταν ο χρήστης προσπαθεί να προσθέσει ένα αρχείο για πρώτη φορά, η εφαρμογή παράγει ένα τυχαίο αλφαριθμητικό το οποίο ονομάζει «file session». Αυτό το αλφαριθμητικό χρησιμοποιείται για να φτιαχτεί ένας φάκελος κάτω από τον κατάλογο «__ticketfiles» του εξυπηρετητή. Τα αρχεία που προσθέτει ο χρήστης ανεβαίνουν ξεχωριστά στην τοποθεσία αυτή (μέσω του κουμπιού «Upload») και όταν ο χρήστης υποβάλλει την αίτηση, τα μονοπάτια των αρχείων που ανέβηκαν αποθηκεύονται στη βάση μαζί με το μήνυμα.

Η διαδικασία αυτή έχει πολλά προτερήματα, κύριο εκ των οποίων είναι ότι μπορούν να υπάρχουν στην εφαρμογή πολλά αρχεία με το ίδιο όνομα. Επίσης, είναι πολύ εύκολο να σβηστούν τα αρχεία ή να τα βρει κάποιος χρήστης στο μηχάνημα (δηλαδή με ftp).

Αφού ο χρήστης έχει συμπληρώσει όλα τα στοιχεία, μπορεί να υποβάλει την αίτηση πατώντας το «Submit ticket». Τα στοιχεία θα ελεγχθούν και θα παραχθεί μια προτεραιότητα για την αίτηση. Αφού γίνει αυτό, θα ανοίξει ένα αναδυόμενο παράθυρο με όλα τα στοιχεία της αίτησης.

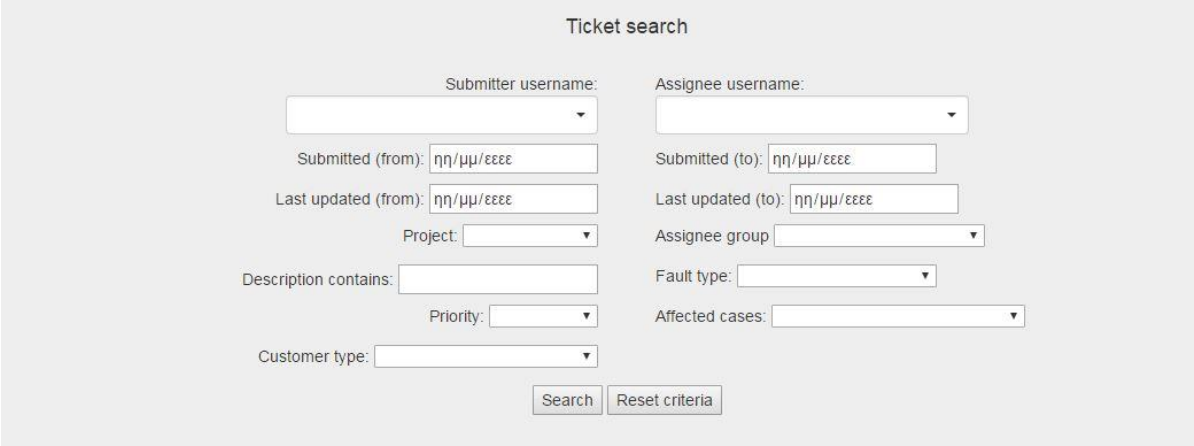
Assigned tickets

Στην περίπτωση που ο χρήστης ανήκει σε ομάδα τύπου *support*, τότε η όψη αυτή παρέχει μια λίστα από ενεργές αιτήσεις που έχουν ανατεθεί στην ομάδα του. Αν ο χρήστης είναι *ticket submitter*, τότε οι αιτήσεις που παρέχονται είναι αιτήσεις που έχει υποβάλει αυτός και έχουν

ανατεθεί πάλι στην ομάδα του. Τέλος, αν ο χρήστης ανήκει σε ομάδα *IT*, στην όψη αυτή θα δει ενεργές αιτήσεις που έχουν ανατεθεί στην ομάδα και στον ίδιο ή στην ομάδα του αλλά σε κανέναν συγκεκριμένο χρήστη. Κατα τα άλλα, η όψη αυτή είναι παρόμοια με την όψη «Home».

Ticket search

Η όψη αυτή επιτρέπει στον χρήστη να κάνει αναζήτηση αιτήσεων με διάφορα κριτήρια.



Εικόνα 43. Αναζήτηση αιτήσεων

Πατώντας «Search» η εφαρμογή αναζητεί αιτήσεις που πληρούν τα κριτήρια του χρήστη και επιστρέφει μια λίστα, στην ίδια μορφή με την όψη «Home».

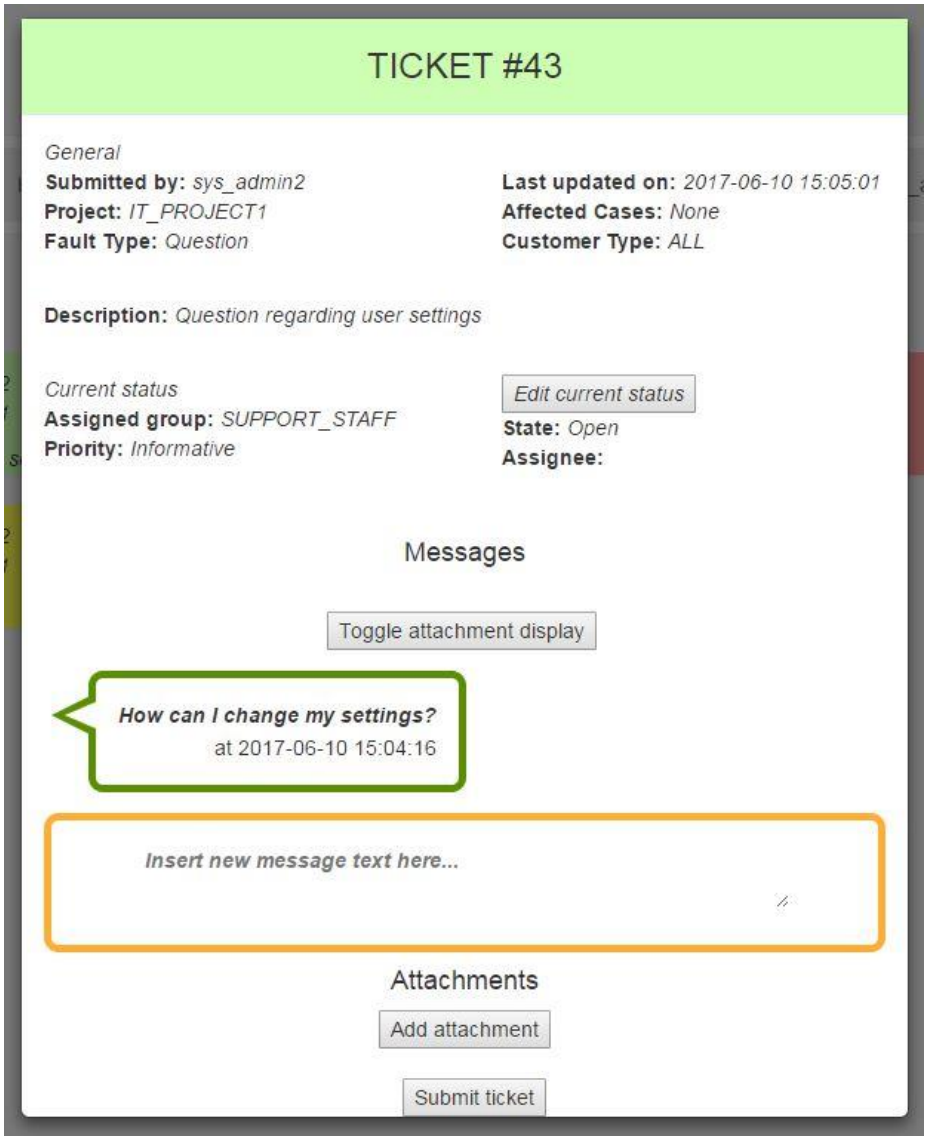
Στο σημείο αυτό είναι πολύ σημαντικό να δείξουμε πώς ο κύκλος ζωής μια αίτησης υλοποιείται στην εφαρμογή. Αρχικά, όπως περιγράψαμε σε προηγούμενη ενότητα, ο κύκλος ζωής μιας αίτησης αποτελείται από τα εξής βήματα:

1. Η αίτηση υποβάλλεται από έναν χρήστη.
2. Η αίτηση προωθείται στην ομάδα υπεύθυνη για την υποστήριξη του έργου (IT) για διερεύνηση.
3. Η ομάδα *IT* προτείνει μια λύση.
4. Ο χρήστης που υπέβαλε αρχικά την αίτηση δέχεται την λύση και η αίτηση κλείνει ή απορρίπτει τη λύση και η αίτηση ανατίθεται ξανά στην ομάδα *IT*.

Η ομάδα *support*, υπεύθυνη για την διαδικασία και την υποστήριξη των ομάδων που συμμετέχουν, ελέγχει την επικοινωνία των υπόλοιπων ομάδων και παρακολουθεί την εξέλιξη των αιτήσεων.

Όταν παρουσιάσαμε το αναδυόμενο παράθυρο των αιτήσεων, αναφέραμε ότι υπό προϋποθέσεις οι ομάδες μπορούν να αλλάξουν την κατάσταση της αίτησης. Όταν, λοιπόν, οι *ticket submitters* υποβάλλουν μια άιτηση, η αίτηση αυτή είναι αρχικά σε κατάσταση «Ανοικτή» και ανατίθεται στην ομάδα *support*.

Η ομάδα *support* μπορεί να δει την αίτηση είτε στην όψη «Assigned tickets» είτε στην όψη «Home» είτε μπορεί να την αναζητήσει. Στο αναδυόμενο παράθυρο, έχει το δικαίωμα να τροποποιήσει την αίτηση ως εξής:



Εικόνα 44. Το αναδυόμενο παράθυρο αίτησης με επιλογή επεξεργασίας

Το κουμπί «Edit current status» επιτρέπει την αλλαγή ενός ή παραπάνω στοιχείων της ενότητας «Current status»:

Εικόνα 45. Αλλαγή τρέχουσας κατάστασης αίτησης

Όπως φαίνεται στην Εικόνα 45, ο χρήστης μπορεί να αλλάξει την ομάδα στην οποία έχει ανατεθεί αυτή τη στιγμή η αίτηση, την κατάστασή της αλλά και την προτεραιότητα της. Το

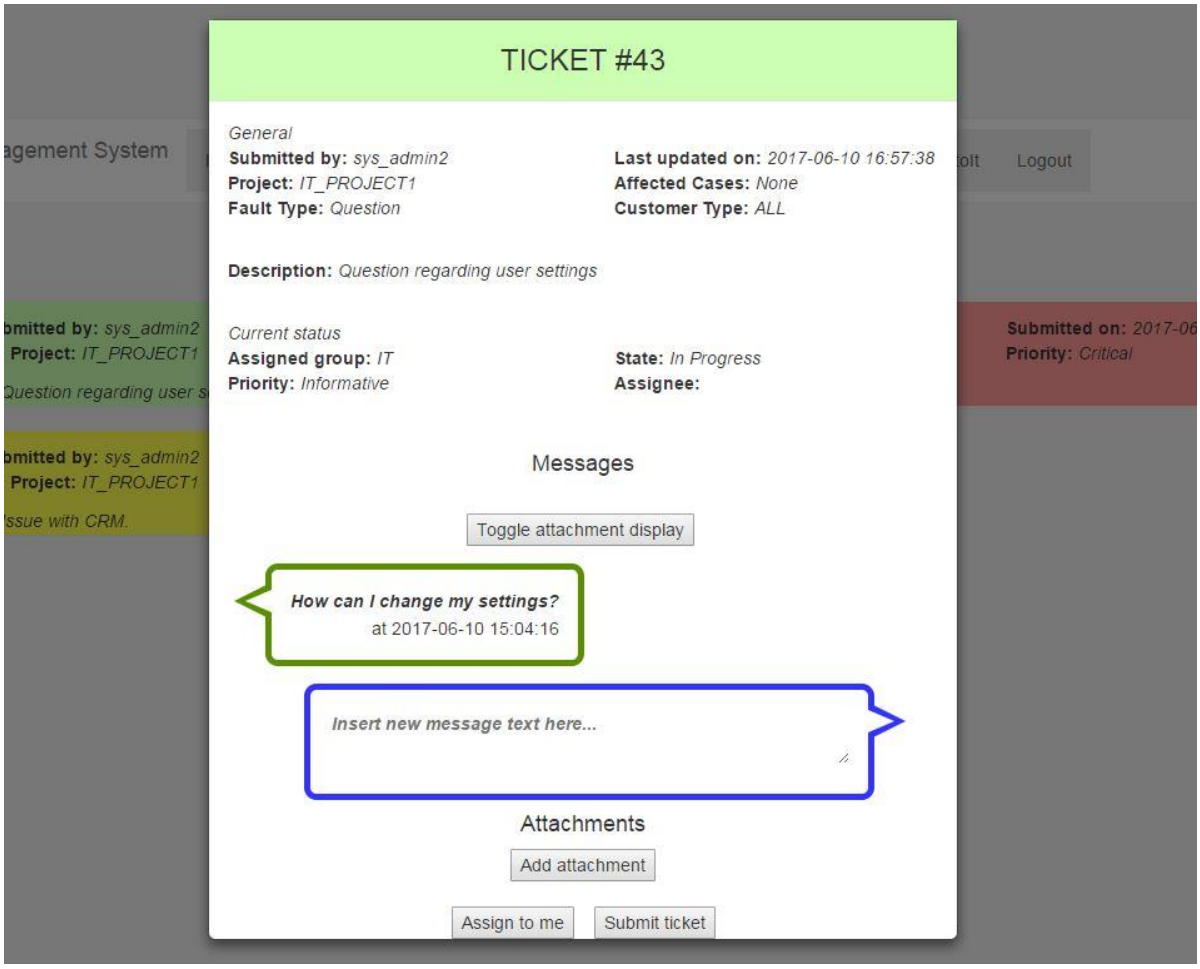
τελευταίο, βέβαια, ισχύει μόνο για χρήστης ομάδας *support* και υπάρχει καθαρά και μόνο για λόγους διόρθωσης λαθών.

Για να προχωρήσουμε στο επόμενο βήμα, η ομάδα *support* θα αναθέσει την αίτηση στην ομάδα *IT* για διερεύνηση και θα αλλάξει την κατάσταση της αίτησης σε «Υπό επεξεργασία»:

Εικόνα 46. Αλλαγή κατάστασης αίτησης σε "Υπό επεξεργασία" και προώθηση στην ομάδα "IT"

Οι αλλαγές αποθηκεύονται πατώντας το κουμπί «Submit ticket» στο κάτω μέρος του αναδυόμενου παράθυρου. Φυσικά, οι χρήστες του *support* μπορούν να προσθέσουν και κάποιο σχόλιο.

Για να προχωρήσει η ομάδα *IT* σε διερεύνηση θα πρέπει πρώτα να αναθέσει την αίτηση σε κάποιο συγκεκριμένο άτομο. Οπότε, αφού ο χρήστης της ομάδας *IT* βρει και ανοίξει την αίτηση από μία από τις όψεις που υπάρχουν για αυτόν τον σκοπό, προτού μπορέσει να την επεξεργαστεί, πρέπει να πατήσει το κουμπί «Assign to me», που βρίσκεται δίπλα στο κουμπί «Submit ticket».



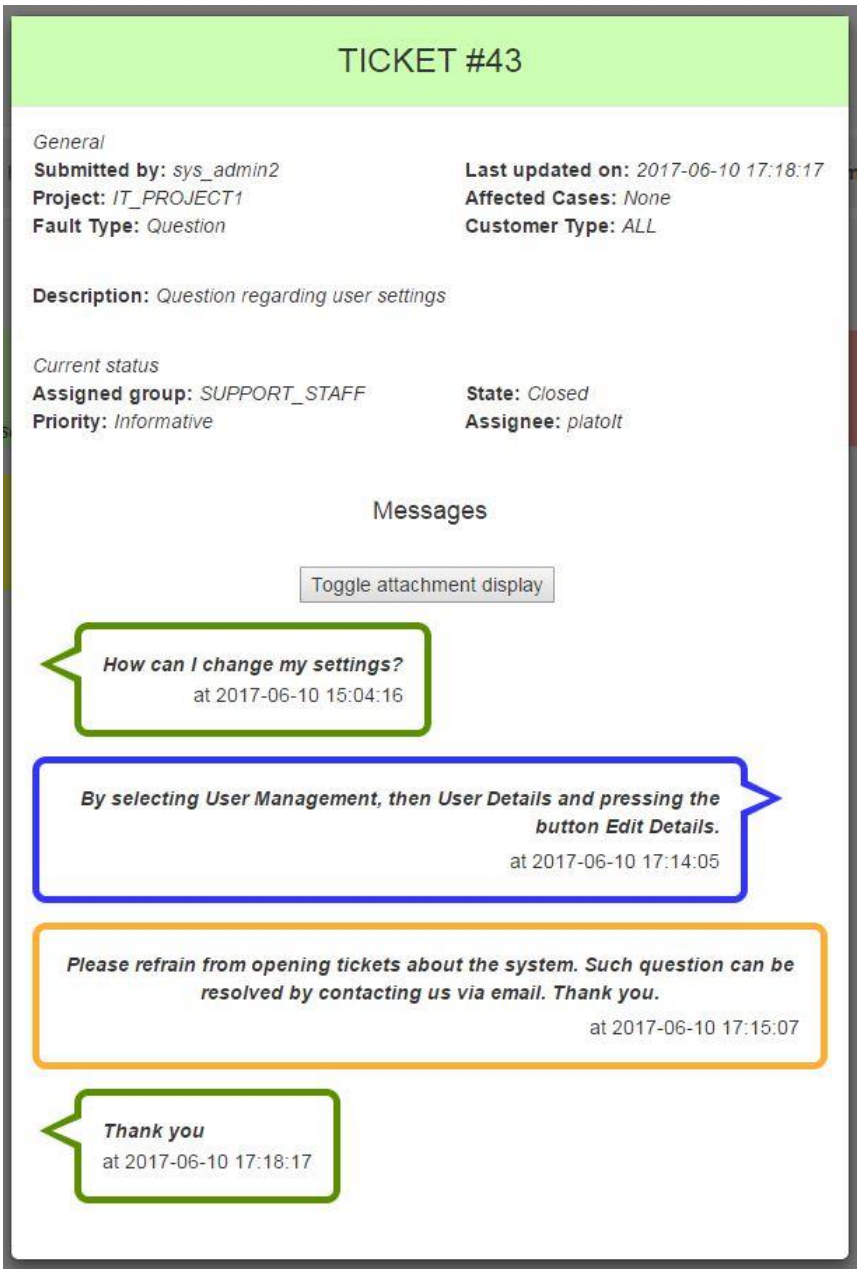
Εικόνα 47. Αναδυόμενο παράθυρο αίτησης με χρήστη ομάδας τύπου "IT"

Αφού πατήσει το κουμπί αυτό, έχει πλέον πρόσβαση στην επεξεργασία της κατάστασης της αίτησης. Επίσης, το όνομα χρήστη του φαίνεται δίπλα στην ετικέτα «Assignee».

Όταν ο χρήστης έχει να προτείνει κάποια λύση (στο παράδειγμα μας η αίτηση είναι πληροφοριακού χαρακτήρα άρα ο χρήστης απλά χρειάζεται να απαντήσει), γράφει την απάντηση στο μπλε σύννεφο και αλλάζει την κατάσταση της αίτησης σε «Ολοκληρωμένη». Έτσι, η αίτηση προωθείται (δηλαδή ανατίθεται) στην ομάδα *support* για έλεγχο των στοιχείων προτού προωθηθεί στην ομάδα του χρήστη που την υπέβαλε αρχικά.

Ο χρήστης *IT* έχει επίσης το δικαίωμα να ξανααναθέσει την αίτηση σε κάποια άλλη ομάδα, χωρίς να αλλάξει την κατάσταση της, στην περίπτωση που χρειάζεται κάποιες παραπάνω διευκρινίσεις. Η επιλογή αυτή έχει να κάνει και με τις εκάστοτε διαδικασίες που θα ορίσουν οι ομάδες μεταξύ τους για την καλύτερη τους συνεργασία.

Τέλος, άμα ο χρήστης που υπέβαλε την αίτηση θεωρήσει ότι η απάντηση που έλαβε ήταν επαρκής, αλλάζει την κατάσταση της αίτησης σε «Κλειστή» και η διαδικασία λαμβάνει τέλος. Σε αντίθετη περίπτωση, μπορεί να επαναφέρει την κατάσταση της αίτησης σε «Υπό επεξεργασία» μέχρι να βρεθεί καλύτερη λύση ή να λυθεί η απορία του. Παρακάτω, βλέπουμε την αίτηση του παραδείγματος κλειστή:



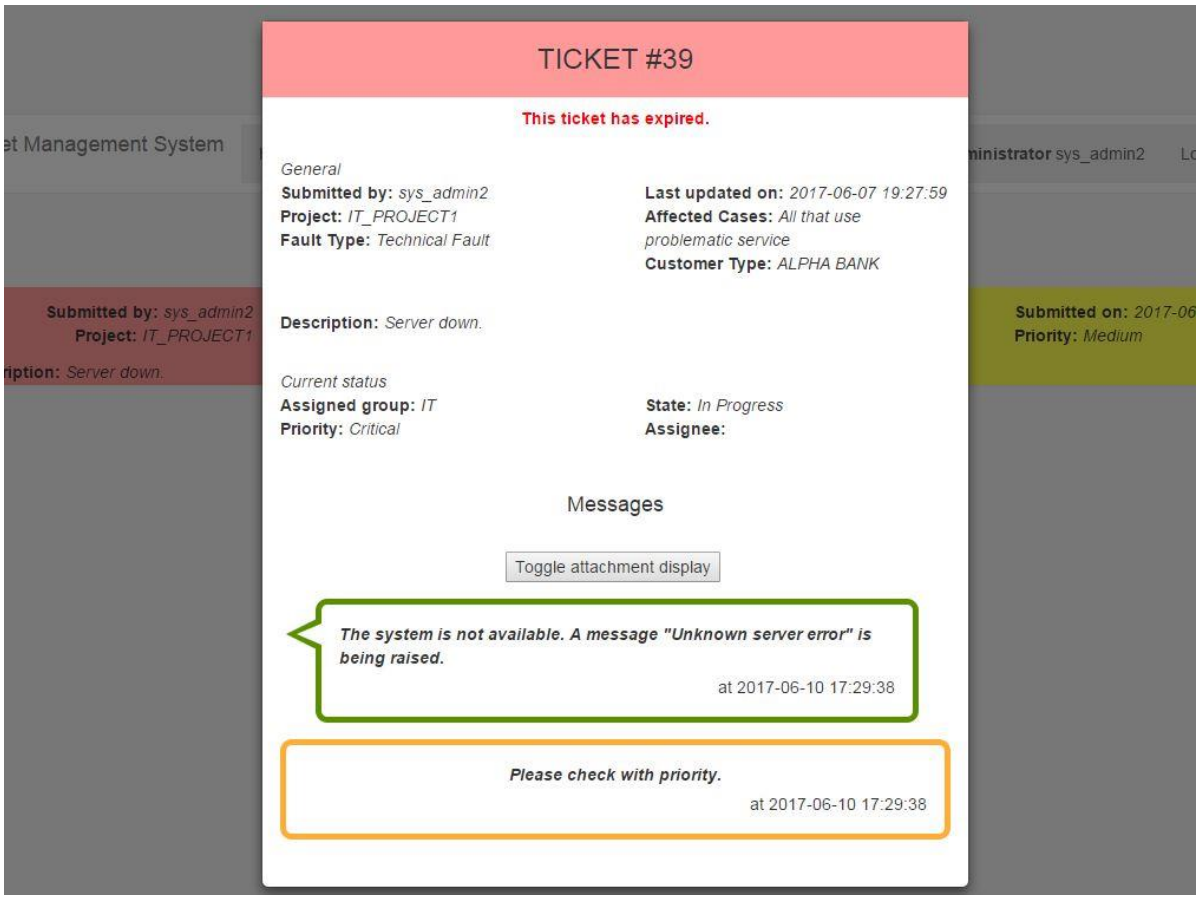
Εικόνα 48. Μια κλειστή αίτηση

Παρατηρούμε ότι κανένας χρήστης δεν μπορεί να αλλάξει την κατάσταση της ή να προσθέσει μήνυμα σε μια κλειστή αίτηση.

Επίσης, όπως φαίνεται και στην Εικόνα 48, τα μηνύματα έχουν χρωματισμούς για να ξεχωρίζουν οι ομάδες που τα πρόσθεσαν. Πιο συγκεκριμένα, τα πράσινα μηνύματα με βελάκι προς τα αριστερά είναι μηνύματα από *ticket submitters*, τα κίτρινα μηνύματα χωρίς βελάκι είναι από *support* και, τέλος, τα μπλε μηνύματα με βελάκι προς τα δεξιά είναι από *IT*.

Όπως είχαμε αναφέρει, ανάλογα με την προτεραιότητα μιας αίτησης ορίζεται και ένα χρονικό όριο μέσα στο οποίο μια αίτηση πρέπει να έχει κλήσει. Αν ξεπεραστεί το όριο αυτό, τότε η αίτηση δεν μπορεί να επεξεργαστεί από καμία ομάδα και «παγώνει» στην τρέχουσα

κατάσταση της. Ένα μήνυμα, το οποίο φαίνεται στην παρακάτω εικόνα, ειδοποιεί τους χρήστες:



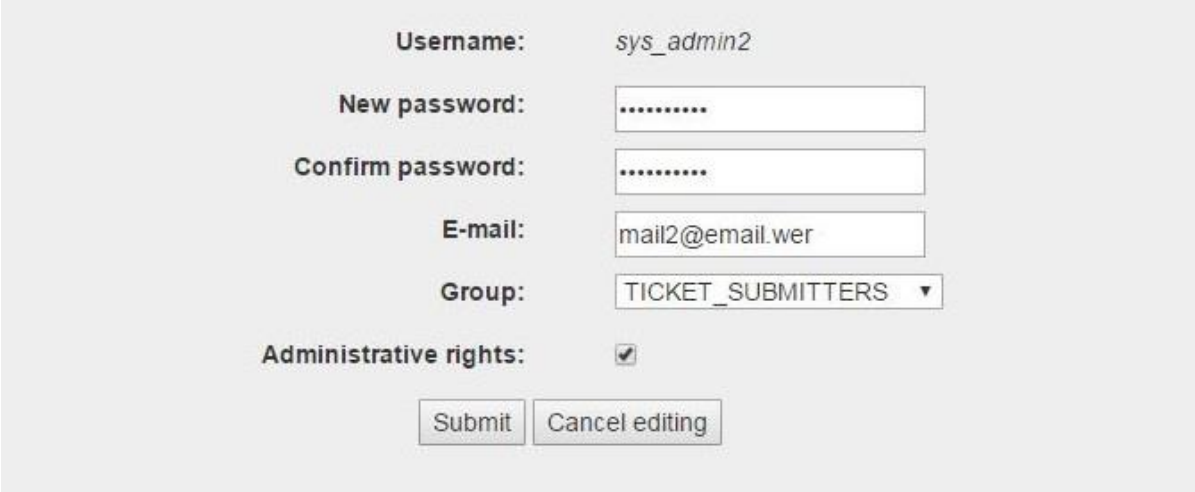
Εικόνα 49. Εκπρόθεσμη αίτηση

User Details

Το μενού αυτό (η πρώτη επιλογή του «User Administration») είναι διαθέσιμη για όλους τους χρήστες και εμφανίζει τα στοιχεία τους.

Εικόνα 50. Στοιχεία του χρήστη όπως εμφανίζονται στο μενού "User Details"

Αν ο χρήστης επιθυμεί να αλλάξει κάποια από τα στοιχεία του, αρκεί να πατήσει στο κουμπί «Edit» και η οθόνη αλλάζει κατάλληλα ώστε να επιτρέψει τις αλλαγές, όπως φαίνετα στην Εικόνα 51.



Εικόνα 51. Επεξεργασία στοιχείων χρήστη στην όψη "User Details"

Το κουμπί «Submit» υποβάλλει τις αλλαγές ενώ το κουμπί «Cancel Editing» διακόπτει την διαδικασία και επαναφέρει την όψη στην αρχική της κατάσταση.

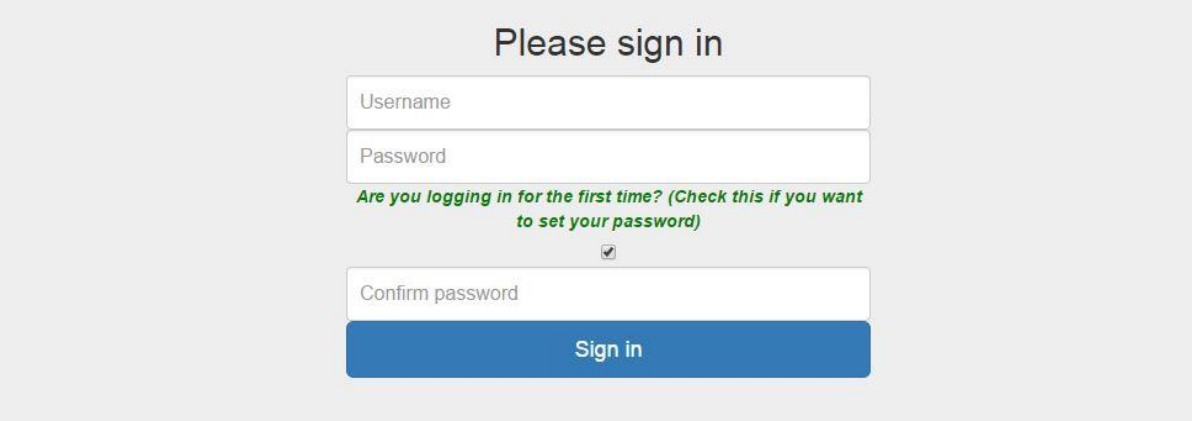
New User

Διαθέσιμη μόνο σε χρήστες με δικαιώματα διαχειριστή, η όψη αυτή επιτρέπει την δημιουργία νέου χρήστη. Η διαδικασία αυτή ολοκληρώνεται σε δύο βήματα.

Εικόνα 52. Εισαγωγή νέου χρήστη

Στο πρώτο βήμα ο διαχειριστής προσθέτει νέο χρήστη μέσω του μενού «New User». Στο μενού αυτό εισάγονται μόνο το όνομα χρήστη του νέου χρήστη, η διεύθυνση ηλεκτρονικού ταχυδρομίου του και αν έχει δικαιώματα διαχειριστή. Η ομάδα του νέου χρήστη είναι ίδια με την ομάδα του χρήστη που τον εισάγει.

Στο δεύτερο βήμα, ο νέος χρήστης μεταβαίνει στη σελίδα σύνδεσης στην εφαρμογή και «τσεκάρει» την επιλογή κάτω από το κείμενο που λέει «Are you logging in for the first time? (Check this if you want to set your password)». Αυτό εμφανίζει ένα πεδίο «Confirm password». Αυτό επιτρέπει στον χρήστη και να συνδεθεί στην εφαρμογή αλλά και να ορίσει τον μυστικό κωδικό του. Η διαδικασία αυτή γίνεται μόνο κατά την πρώτη του σύνδεση στην εφαρμογή. Έπειτα, συνδέεται κανονικά.



Εικόνα 53. Φόρμα πρώτης σύνδεσης στην εφαρμογή

Delete User

Η όψη αυτή επιτρέπει σε διαχειριστές να απανεργοποιήσουν έναν χρήστη. Όπως φαίνεται και στην Εικόνα 54, παρέχεται ένα κουτί στο οποίο ο χρήστης συμπληρώνει το όνομα του χρήστη που θέλει να διαγραφεί.

Εικόνα 54. Φόρμα διαγραφής χρήστη

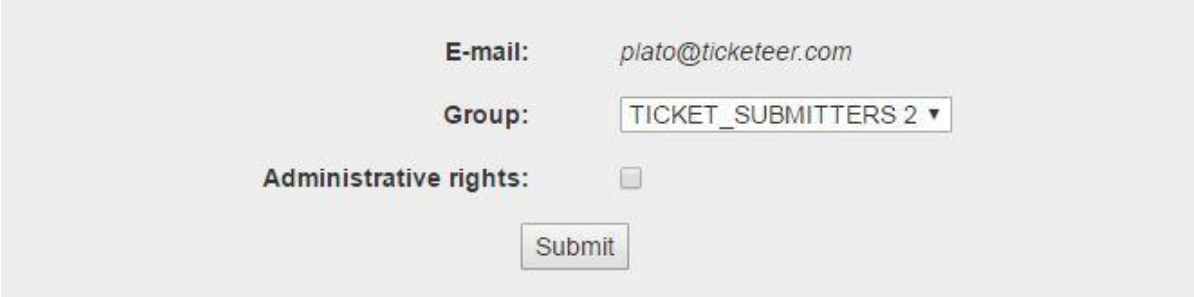
Στη συνέχεια ο χρήστης πρέπει να πατήσει το κουμπί «Search», το οποίο θα εμφανίσει τα στοιχεία του χρήστη.

Εικόνα 55. Τα στοιχεία του χρήστη, όπως φαίνονται στην όψη "Delete User"

Το κουμπί «Delete» απενεργοποιεί τον χρήστη ενώ το κουμπί «Reset» επαναφέρει την όψη στην κατάσταση πριν την αναζήτηση.

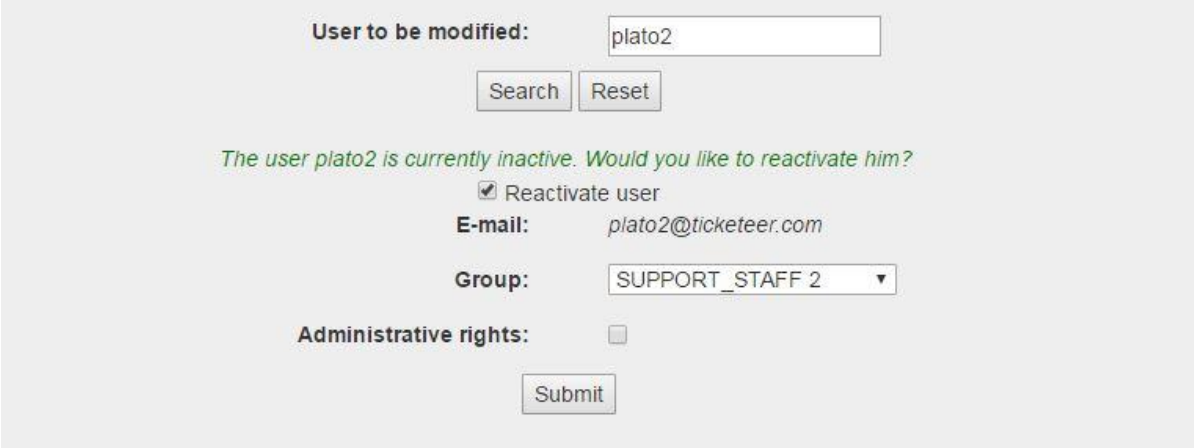
Modify a user

Η όψη αυτή είναι διαθέσιμη μόνο για χρήστες με δικαιώματα διαχειριστή και επιτρέπει την επεξεργασία των στοιχείων ενός χρήστη. Αντίστοιχα με την όψη «Delete User», ο διαχειριστής πρέπει πρώτα να αναζητήσει έναν χρήστη εισάγωντας το όνομα του και πατώντας το κουμπί «Search».



Εικόνα 56. Όψη "Modify a user" μετά την αναζήτηση

Αν ο χρήστης που αναζητάται είναι ανενεργός, τότε για να μπορεί ο διαχειριστής να τροποποιήσει τα στοιχεία του θα πρέπει να τον ενεργοποιήσει.



Εικόνα 57. Αναζήτηση ανενεργού χρήστη

Όπως φαίνεται και στην Εικόνα 57, αρκεί η επιλογή του «Reactivate User» για να μπορέσει ο χρήστης να προχωρήσει.

Σε κάθε μία εκ των παραπάνω περιπτώσεων, το κουμπί «Submit» υποβάλλει τις αλλαγές ενώ το κουμπί «Reset» επαναφέρει την όψη στην κατάσταση πριν την αναζήτηση.

5 Αξιολόγηση

Στην παρούσα εργασία παρουσιάσαμε ένα πρόβλημα στη διαδικασία αναφοράς σφαλμάτων καθώς και, γενικότερα, στη συνεργασία μεταξύ εταιρειών: τα SLA που συμφωνούνται μεταξύ ενός παρόχου υπηρεσιών και του πελάτη του δεν παρέχουν επαρκή πληροφορία για την σωστή επικοινωνία σφαλμάτων από τον πελάτη και τα συστήματα αιτήσεων υποστήριξης υλοποιούν τα SLA μόνο σε ό,τι αφορά σε χρόνους απόκρισης ή λύσης του σφάλματος. Αυτό επιτρέπει στους πελάτες την κατάχρηση των συστημάτων και δυσχαιρένει την διαδικασία υποστήριξης.

Σαν λύση, προτείναμε, αρχικά, τα SLA να παρέχουν ένα σύνολο από χαρακτηριστικά και μεγέθη καθώς και τον τρόπο με τον οποίο επηρεάζουν την βαρύτητα ενός σφάλματος, θέτωντας έτσι ένα πιο κατανοητό πλαίσιο επικοινωνίας. Δεύτερον, προτείναμε συστήματα τα οποία να λαμβάνουν υπόψην τους τα SLAs πιο συνολικά και στοιχεία βασικά για την σωστή λειτουργία της διαδικασίας υποστήριξης των πελατών όπως η προτεραιότητα της αίτησης να προκύπτουν με δυναμικό τρόπο από τα χαρακτηριστικά που έχει θέσει ο χρήστης και με βάση, πάντα, ό,τι έχει οριστεί στο SLA.

Θέτοντας ένα πιο κατανοητό πλαίσιο επικοινωνίας μεταξύ πελάτη και πράκτορα IT βελτιώνει την επικοινωνία των σφαλμάτων διότι θέτει ένα λεξιλόγιο κατανοητό από δύο ομάδες με πιθανώς διαφορετικά τεχνικά προφίλ. Ακόμα και χωρίς να λάβουμε υπόψην τις καταχρήσεις, το γεγονός αυτό από μόνο του βελτιώνει την ταχύτητα με την οποία λύνονται τα σφάλματα.

Για να φανούν, όμως, τα πραγματικά οφέλη ενός πιο κατανοητού και λεπτομερούς SLA πρέπει να το λάβει υπόψην πλήρως και το σύστημα το οποίο χρησιμοποιείται για την διαχείριση των αιτήσεων σφαλμάτων. Το γεγονός ότι ο χρήστης θα επιλέγει μόνο τα χαρακτηριστικά αυτά που έχουν περιγραφεί στο εκάστοτε SLA σημαίνει ότι μειώνονται οι δυνατότητες κατάχρησης του συστήματος. Αντίθετα, ο χρήστης εισάγει στοιχεία με βάση τα οποία ο πράκτορας καταλαβαίνει την βαρύτητα και το μέγεθος τους σφάλματος. Έτσι, οι προτεραιότητες που προκύπτουν από το SLA τηρούνται από όλες τις εμπλεκόμενες ομάδες και η διαδικασία υποστήριξης βελτιώνεται και επιταχύνεται.

Στις επόμενες ενότητες, αξιολογούμε τον αντίκτυπο της λύσης αυτής σε άλλες πτυχές ενός οργανισμού ή μιας εταιρείας και την εφαρμογή που αναπτύξαμε έναντι σε άλλα εμπορικά συστήματα.

5.1 Αξιολόγηση της λύσης

Για την αξιολόγηση της λύσης, δημιουργήσαμε ένα διάγραμμα *Balanced Scorecard*. Προτού προχωρήσουμε στην περιγραφή του διαγράμματος μας θα περιγράψουμε σύντομα τι είναι το *Balanced Scorecard*.

5.1.1 Balanced Scorecard

Το Balanced Scorecard (BSC) είναι ένας δείκτης απόδοσης που χρησιμοποιείται στην Στρατηγική Διεύθυνση (Strategic Management). Με τη βοήθεια δεδομένων που συλλέγονται από τμήματα της εταιρείας, το BSC ταυτοποιεί και βελτιώνει εσωτερικές λειτουργίες της καθώς και τα εξωτερικά αποτελέσματα αυτών. Το BSC προτάθηκε για πρώτη φορά από τους Dr. Rober Kaplan και Dr. David Norton και εκδόθηκε το 1992 [5].

Το BSC χωρίζει τον οργανισμό ή την εταιρεία σε τέσσερις τομείς (ή πόδια): learning and growth, business processes, customers και finance. Μέσω της μεθόδου αυτής, ένας οργανισμός συγκεντρώνει στόχους, μετρήσεις, πρωτοβουλίες και σκοπούς καθώς και τους τρόπους με τους οποίους αυτά αλληλοεπηρεάζονται και μπορεί να ταυτοποιήσει ανασταλτικούς παράγοντες καθώς και παράγοντες που έχουν αξία.

Ο τομέας learning and growth ερευνά πόσο αποδοτικά λαμβάνεται και χρησιμοποιείται η πληροφορία από τους υπαλλήλους για να δώσει ανταγωνιστικό πλεονέκτημα στον οργανισμό.

Ο τομέας business processes αφορά στην ποιότητα της παρασκευής προϊόντων (ή υπηρεσιών).

Ο τομέας customers ελέγχει την ικανοποίηση των πελατών σχετικά με την ποιότητα και διαθεσιμότητα των προϊόντων μέσα από σχόλια τους σχετικά με την ικανοποίηση των αναγκών τους.

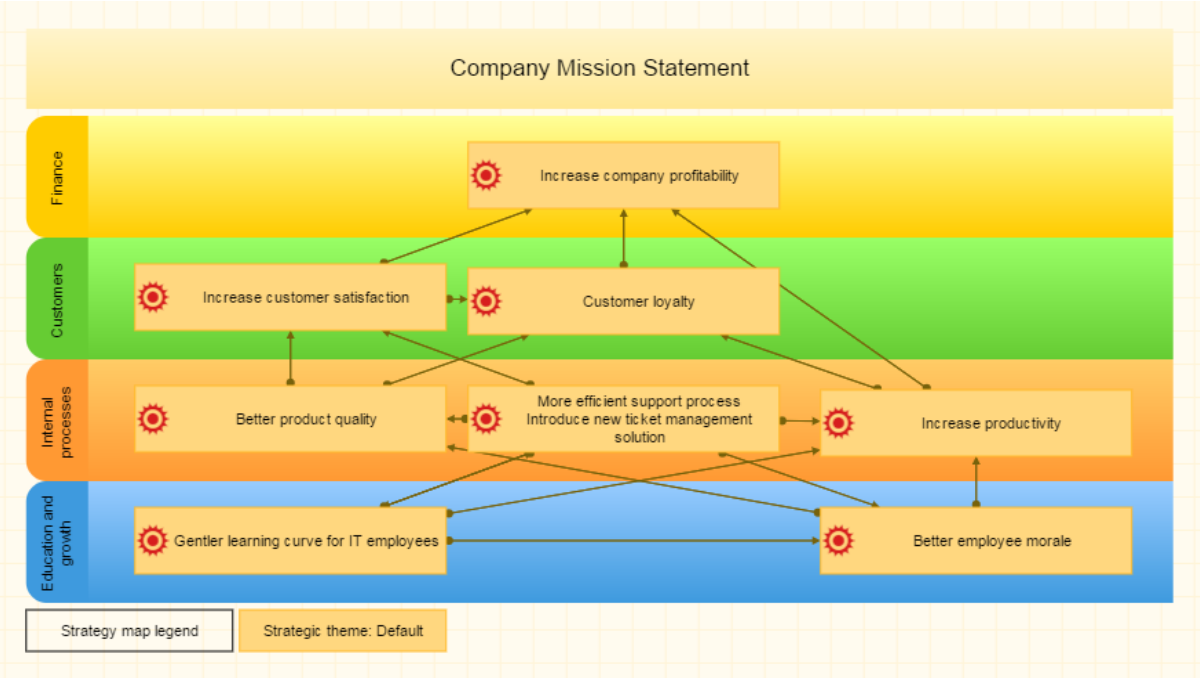
Τέλος, ο τομέας finance παρέχει μετρήσεις σχετικές με τις πωλήσεις, τα έξοδα και τα έσοδα.

Οι τέσσερις αυτοί τομείς, και τα στοιχεία που παρέχουν, αποτελούν τους στόχους και τη στρατηγική του οργανισμού και απαιτούν ανάλυση των δεδομένων που έχουν συλλεχθεί. Για τον λόγο αυτό καθώς και για την επιρροή των πορισμάτων στη στρατηγική του οργανισμού, το BSC χρησιμοποιείται ως εργαλείο διαχείρισης και όχι απλά ως εργαλείο μετρήσεων.

5.1.2 Ανάλυση του BSC

Το διάγραμμα BSC που βλέπουμε στην Εικόνα 58 χωρίζεται στους τέσσερις τομείς που αναφέρθηκαν στην προηγούμενη ενότητα. Οι στόχοι απεικονίζονται ως πλαίσια ενώ βέλη υποδηλώνουν τις αλληλεξαρτήσεις μεταξύ τους. Η λύση που προτάθηκε αποτελεί στόχο στο internal processes τμήμα του BSC. Με αρχή αυτό το πλαίσιο του διαγράμματος θα αναλύσουμε το πώς θα επηρεάσει η λύση μας άλλα στοιχεία και διαδικασίες ενός οργανισμού. Τα βέλη υποδεικνύουν ποιος στόχος επηρεάζει ποιον.

Ο στόχος «More efficient support process» στο πλαίσιο internal processes αντιπροσωπεύει την βελτίωση της διαδικασίας υποστήριξης με την υλοποίηση της λύσης που προτείναμε. Μέσα στο πλαίσιο των internal processes, η αλλαγή αυτή σημαίνει ότι θα βελτιωθούν όλες οι διαδικασίες που βασίζονται στη διαδικασία υποστήριξης όπως, για παράδειγμα, η δοκιμή νέων προϊόντων καθώς και η βελτίωση των υπάρχοντων. Συνεπώς, βελτιώνεται η συνολική ποιότητα των προϊόντων ή υπηρεσιών της εταιρείας. Στο διάγραμμα αυτό φαίνεται μέσω του βέλους που συνδέει τον στόχο «More efficient support process» με τον στόχο «Better product quality». Επίσης, λόγω της καλύτερης απόδοσης της διαδικασίας υποστήριξης, οι εργαζόμενοι στα αρμόδια τμήματα θα χρειάζονται λίγοτερο χρόνο κατά μέσο όρο για κάθε αίτηση οπότε άλλα καθήκοντα θα μπορούν να διευθετούνται πιο γρήγορα. Συνεπώς, βελτιώνεται η παραγωγικότητα της εταιρείας (στόχος «Increase productivity»).



Εικόνα 58. Το διάγραμμα Balanced Scorecard της λύσης

Οι βελτιωμένες διαδικασίες υποστήριξης, επίσης, δημιουργούν καλύτερες συνθήκες για τους εργαζόμενους και, συνεπώς, βελτιώνουν το ηθικό των ομάδων. Στο διάγραμμα, αυτό φαίνεται με τον στόχο «Better employee morale» του πλαισίου Education and growth. Αντίστοιχα, η καλύτερη οργανωσή και διαχείριση των αιτήσεων που προκύπτει από το αναλυτικότερο SLA καθιστά την ένταξη νέων ατόμων στις ομάδες υποστήριξης πιο εύκολη και γρήγορη (στόχος «Gentler learning curve for IT employees»). Το αποτέλεσμα τηε τελευταίας αυτής βελτίωσης είναι ότι τα νεότερα μέλη των ομάδων αυτών γίνονται παραγωγικά πιο άμεσα. Έτσι, βελτιώνει και την παραγωγικότητα (στόχος «Increase productivity») αλλά και την ψυχολογία των εργαζομένων (στόχος «Better employee morale») διότι ο όγκος εργασίας μοιράζεται πιο δίκαια πιο γρήγορα. Έτσι, η παραγωγικότητα της εταιρείας βελτιώνεται από τρεις παράγοντες σαν αποτέλεσμα της βελτίωσης ενός.

Αντίστοιχα, λόγω της καλής ψυχολογίας και του βελτιωμένου ηθικού, οι εργαζόμενοι έχουν ένα παραπάνω κίνητρο να κυνηγάνε την ποιότητα στη δουλειά τους. Έτσι, πέρα από την παραγωγικότητα αυξάνεται και η ποιότητα των προϊόντων ή υπηρεσιών (στόχος «Better product quality»).

Η βελτιώσεις αυτές έχουν άμεσο αντίκτυπο στην εικόνα και σχέση του οργανισμού ή της εταιρείας με τους πελάτες. Η βελτίωση στη διαδικασία υποστήριξης σημαίνει και καλύτερη εξυπηρέτηση των πελατών άρα και καλύτερη ικανοποίηση τους, όπως φαίνεται από τον στόχο «Increase customer satisfaction» στο πλαίσιο Customers. Επίσης, η καλύτερη ποιότητα υπηρεσιών ή προϊόντων βελτιώνει ανάλογα την ικανοποίηση των πελατών αλλά και την πιθανότητα να παραμείνουν πελάτες (στόχος «Customer loyalty»). Τον τελευταίο στόχο τον επηρεάζουν και η ικανοποίηση των πελατών αλλά και η παραγωγικότητα.

Τέλος, στο πλαίσιο *Finance*, έχουμε τον τελευταίο και πιο βασικό στόχο: την κερδοφορία της επιχείρησης (στόχος «Increase company profitability»), όπου με τον όρο επιχείρηση

υποννοούμε και οργανισμούς και εταιρείες. Ο στόχος αυτός είναι ο βασικότερος αν όχι ο μοναδικός. Όπως φαίνεται και στο διάγραμμα μας, η βελτίωση της ικανοποίησης των πελατών καθώς και η πίστη των πελατών βελτιώνουν την κερδοφορία. Αυτό συμβαίνει διότι ο οργανισμός μπορεί να κρατήσει και πιθανώς να πουλήσει περισσότερα προϊόντα στις εδραιομένες αγορές ή στους υπάρχοντες πελάτες του αλλά προσελκύει και άλλους πελάτες χάρην στην καλή φήμη που του παρέχει η ικανοποίηση των υπάρχοντων. Τέλος, η αυξημένη παραγωγικότητα δίνει στην εταιρεία την δυνατότητα παροχής και πώλησης περισσότερων προϊόντων ή υπηρεσιών καθώς και την δυνατότητα βελτίωσης των παραγόμενων της, με αποτέλεσμα την αύξηση των πωλήσεων.

Από το διάγραμμα και την παραπάνω ανάλυση φαίνεται ότι η βελτίωση της διαδικασίας υποστήριξης ενός οργανισμού και, κατ'επέκταση, η λύση που προτείνουμε, επιφέρει βελτίωση τόσο στην κερδοφορία όσο και στην συνεργασία με τους πελάτες του.

5.2 Αξιολόγηση της εφαρμογής

Το δεύτερο βήμα στην αξιολόγηση της εργασίας είναι η αξιολόγηση της εφαρμογής που προκύπτει από την υλοποίηση της λύσης που προτείνουμε. Αυτό θα γίνει συγκρίνοντάς την με δύο αντίστοιχα συστήματα, το *Freshservice* και το *Zendesk*.

Τα κριτήρια τα οποία θα χρησιμοποιηθούν είναι τα εξής:

- Υποστήριξη SLA από το σύστημα
- Κατά πόσο απλοποιεί και επιταχύνει τη διαδικασία υποστήριξης.

Τα κριτήρια που χρησιμοποιούμε αναφέρονται κυρίως στην υποστήριξη της διαδικασίας από την εφαρμογή.

Στις παρακάτω ενότητες, πρώτα περιγράφουμε συνοπτικά το κάθε σύστημα και μετά προχωράμε στην σύγκριση του με την πρότυπη εφαρμογή μας.

5.2.1 Freshservice

Το *Freshservice*[10] της *Freshworks* είναι ένα βραβευμένο σύστημα διαχείρισης αιτήσεων υποστήριξης το οποίο χρησιμοποιείται από πολλές διαφόρων μεγεθών εταιρείες και οργανισμούς. Ένα από τα ιδιαίτερα χαρακτηριστικά του *Freshservice* είναι η ενσωμάτωση παιχνιδιού με στόχους και ορόσημα στο σύστημα ώστε η επίλυση προβλημάτων να γίνει πιο διασκεδαστική για τους πράκτορες.

Το *Freshservice* είναι εύκολο και απλό στην εγκατάσταση καθώς και απλό στη χρήση. Παρέχει μια αρχική σελίδα στην οποία φαίνονται κάποια στοιχεία όπως ο αριθμός ανοιχτών αιτήσεων καθώς και ο αριθμός αιτήσεων που λήγουν την σημερινή μέρα αλλά και μηνύματα με την πρόσφατη δραστηριότητα που αφορά στον πράκτορα. Παρέχει, επίσης, διαφορετικές διεπαφές για αιτήσεις υποστήριξης, αλλαγές, προβλήματα και εκδόσεις αλλά και μια βάση γνώσης στην οποία μπορούν να ανατρέξουν και πράκτορες και πελάτες. Τέλος, παρέχει πολλές επιλογές για αναφορές σχετικές με όλα τα μεγέθη που χρησιμοποιούνται από την εφαρμογή.

Όσον αφορά στο πρόβλημα το οποίο αναλύουμε, το *Freshservice* επιτρέπει στον χρήστη να θέτει μια συγκεκριμένη προτεραιότητα, μη δίνοντας λύση στο υπό συζήτηση πρόβλημα.

Παρ'όλα αυτά, ενώ η προτεραιότητα αυτή είναι υποχρεωτική, δεν είναι το μόνο σχετικό στοιχείο. Ο χρήστης μπορεί προαιρετικά να θέσει και δύο περαιτέρω πεδία: το πεδίο «urgency», δηλαδή πόσο κρίσιμο είναι να λυθεί το συγκεκριμένο πρόβλημα, και το πεδίο «impact», το οποίο είναι αντίστοιχο με το πεδίο «Affected cases», της δικής μας εφαρμογής. Ενώ τα παραπάνω πεδία δεν αποτελούν ακριβώς λύση, δίνουν ένα έναυσμα στον χρήστη να σκεφτεί την πληροφορία που υποβάλλει. Αναλόγως με τις διαδικασίες και την συνεργασία μεταξύ των επιχειρήσεων, το γεγονός αυτό μπορεί και πάλι να βελτιώσει κατά πολύ το πρόβλημα αυτό.

Δυστυχώς, όμως, όπως και στα περισσότερα συστήματα, τα SLA υφίστανται στην εφαρμογή μόνο ως χρόνοι απόκρισης και επίλυσης προβλημάτων καθώς και πολιτικές που εφαρμόζονται όταν τα αιτήματα κοντεύουν να λήξη ή λήγουν.

Επίσης, το *Freshservice* δεν ορίζει κάποιον συγκεκριμένο κύκλο ζωής για τα αιτήματα. Αντίθετα, το κάθε αίτημα μπορεί να το επεξεργαστούν οι εμπλεκόμενοι πράκτορες ελεύθερα και να αλλάξουν όλες τις τιμές συμπεριλαμβανομένης και της προτεραιότητας. Επίσης, σαν συνέπεια των παραπάνω, δεν υπάρχουν τελικές καταστάσεις των αιτημάτων, με αποτέλεσμα ένα αίτημα το οποίο να είναι κλειστό να μπορεί να ανοίξει ξανά. Εδώ υποθέτουμε ότι το σύστημα δεν παίρνει αποφάσεις για τις διαδικασίες, αφήνοντας τις εμπλεκόμενες επιχειρήσεις να τις θέσουν.

Τα παραπάνω μπορεί να μην δίνουν, αφενός, μια γενική λύση στο υπό μελέτη πρόβλημα αλλά καθιστούν το *Freshservice* πολύ εύκολο και απλό στη χρήση. Η διαδικασία υποβολής και επεξεργασίας ενός αιτήματος είναι ευθύς και γρήγορη. Με τη χρήση της βάσης γνώσεων δίνεται η δυνατότητα και στους πράκτορες αλλά και στον πελάτη να μπορούν να αναζητήσουν πληροφορία εκεί, πιθανώς αποφεύγοντας την υποβολή ενός αιτήματος εξ'ολοκλήρου. Το «παιχνίδι», δηλαδή το ενσωματωμένο σύστημα με τους στόχους για τους πράκτορες, κάνει την διαδικασία πιο ενδιαφέρουσα και διασκεδαστική. Τέλος, υποστηρίζει πολλά κανάλια επικοινωνίας (e-mail, τις αιτήσεις, chat κ.λ.π.) με αποτέλεσμα την καλύτερη εξυπηρέτηση του πελάτη.

Εν κατακλείδι, το *Freshservice* αποτελεί ένα αποδοτικό, προσβάσιμο και πλήρες σύστημα υποστήριξης που απαιτεί ελάχιστο ή καθόλου χρόνο στην εκμάθηση, μπορεί να καλύψει τις ανάγκες πολλών διαφορετικών επιχειρήσεων και, ενώ δεν λύνει το υπό μελέτη πρόβλημα, έχει κάνει πολλά βήματα προς την βελτίωση του χώρου ως προς αυτόν τον τομέα.

5.2.2 Zendesk

Το *Zendesk*[11] της ομώνυμης εταιρείας είναι επίσης ένα αρκετά διαδεδομένο σύστημα διαχείριστης αιτήσεων. Ιδιαίτερο χαρακτηριστικό του συγκεκριμένου συστήματος είναι οι αυτοματισμοί που μπορεί να θέσει ο χρήστης για διάφορες λειτουργίες πάνω στις αιτήσεις για τους οποίους θα πούμε μερικά λόγια παρακάτω.

Στόχοι του *Zendesk* είναι μια καλύτερη εμπειρία για τον χρήστη, είτε είναι υπάλληλος IT είτε πελάτης, και η καλύτερη εξυπηρέτηση των δεύτερων. Τόσο η διεπαφή χρήστη όσο και το σύστημα το ίδιο είναι απλό στον σχεδιασμό, με αποτέλεσμα να μην χρειάζεται κάποια εκμάθηση. Οι περισσότερες ρυθμίσεις και παραμετροποιήσεις είτε είναι προφανείς σαν χρήση είτε έχουν ξεκάθαρες οδηγίες που καθοδηγούν τον χρήστη.

Όπως και το *Freshservice*, το *Zendesk* επιτρέπει πολλαπλά κανάλια επικοινωνίας με τους πελάτες όπως *chat* ή ομιλία. Είναι πολύ ενδιαφέρον το γεγονός ότι επιτρέπει τη χρήση των υπηρεσιών του *facebook* καθώς και του *twitter*. Επίσης, προσφέρει βάση γνώσεων, SDK για χρήση σε εφαρμογές *android* καθώς και *widget* που δίνει γρήγορη πρόσβαση σε λειτουργίες.

Όσον αφορά στην αποδοτικότητα της διαδικασίας, το *Zendesk* υπερτερεί αρκετά έναντι πολλών συστημάτων. Όπως αναφέραμε πιο πάνω, το *Zendesk* προσφέρει πολλούς αυτοματισμούς για την διαχείριση των αιτήσεων. Αρχικά, επιτρέπει την ανάπτυξη αυτόματων απαντήσεων για αιτήσεις ώστε να μην χρειάζεται οι πράκτορες να ασχολούνται με τα πιο συνηθισμένα ή γνωστά προβλήματα. Δεύτερον, επιτρέπει την ρύθμιση αυτοματισμών οι οποίοι κάνουν συγκεκριμένες ενέργειες σε αιτήσεις που ικανοποιούν συγκεκριμένες προϋποθέσεις όπως, για παράδειγμα, να κλείσουν όλες οι αιτήσεις που έχουν απαντηθεί επαρκώς μετά από τρεις μέρες. Τέλος, επιτρέπει την δημιουργία λειτουργιών που «αντιδρούν» σε συγκεκριμένες ενέργειες με περαιτέρω ενέργεις. Παραδείγματος χάρην, όταν μία αίτηση ανοίγει για πρώτη φορά, το σύστημα μπορεί να ρυθμιστεί να στέλνει *e-mail* σε κάποιο συγκεκριμένο χρήστη ή ομάδα.

Ενώ το *Zendesk* σαν υλοποίηση, εμπειρία χρήστη και πλατφόρμα παρέχει πολλές λειτουργίες σε πολύ υψηλή ποιότητα, δεν λύνει το υπό μελέτη πρόβλημα. Η προτεραιότητα τίθεται από τον χρήστη που εισάγει την αίτηση και μία αίτηση που δεν έχει κλήσει μπορεί να αλλάζει ελεύθερα καταστάσεις καθώς και προτεραιότητα. Επίσης, τα SLAs, όπως και στο *Freshservice*, αποτελούν μόνο χρονικούς περιορισμούς για την διαχείριση των προβλημάτων.

Το *Zendesk* αποτελεί ένα πολύ ισχυρό εργαλείο στο χώρο της υποστήριξης, δίνοντας στον χρήστη επιλογές και για την επικοινωνία με κάποιον πράκτορα ή πελάτη αλλά και πολλά εργαλεία διαχείρισης των αιτήσεων και εισαγωγής αυτοματισμών. Παρ'όλα αυτά, δεν πραγματοποιεί τα απαραίτητα βήματα προς την επίλυση του υπό μελέτη προβλήματος.

5.2.3 Συνολική αξιολόγηση

Στον Πίνακας 1 βλέπουμε μια περίληψη της σύγκρισης των δύο εμπορικών συστημάτων *Zendesk* και *Freshservice* έναντι της πειραματικής εφαρμογής που αναπτύξαμε σαν κομμάτι της εργασίας αυτής.

Τα κριτήρια με βάση τα οποία συγκρίνουμε τις τρεις εφαρμογές είναι το πλήθος και η ποικιλία των λειτουργιών (γραμμή «Λειτουργίες»), το πώς αξιοποιείται ένα εκάστοτε SLA μέσα στην εφαρμογή (γραμμή «Αξιοποίηση SLA») και, τέλος, ο βαθμός στον οποίο κάνουν πιο αποδοτική και εύκολη την διαδικασία διαχείρισης των αιτήσεων υποστήριξης (τρίτη και τελευταία γραμμή).

Όσον αφορά στις υποστηριζόμενες λειτουργίες, το *Zendesk* παρέχει από μόνο του πολλούς δίαυλους επικοινωνίας και λειτουργίες επί των αιτήσεων. Επίσης, υποστηρίζει και εξωτερικές εφαρμογές, *widget* με λειτουργίες του καθώς και SDK για ενταξη σε εφαρμογές *Android.* Το *Freshservice*, πέρα από πολλά κανάλια επικοινωνίας και εργαλεία για αναφορές, παρέχει ένα σύστημα με επιβραβεύσεις και ορόσημα για τους πράκτορες το οποίο κάνει τη διαδικασία πιο ενδιαφέρουσα. Σε αντίθεση με τα εμπορικά συστήματα, η εφαρμογή που αναπτύξαμε σαν τμήμα της εργασίας είναι περιορισμένη στις βασικές σελίδες διαχείρισης των αιτήσεων.

Όσον αφορά στην αξιοποίηση ενός SLA, οι εμπορικές εφαρμογές δυστυχώς περιορίζονται στους χρονικούς περιορισμούς διαχείρισης της αίτησης, δηλαδή ρυθμίσεις όπως μέχρι πότε πρέπει να έχει δωθεί μια πρώτη απάντηση στην αίτηση ή μέχρι πότε πρέπει να έχει λυθεί το πρόβλημα. Αντίθετα, στην δική μας εφαρμογή επιτρέπουμε τη χρήση πιο λεπτομερών SLA με τον θεσμό κατηγοριών σφαλμάτων, πελατών ή και τον θεσμό συγκεκριμένων πελατών καθώς και την σύνδεση τους με προτεραιότητες ώστε το σύστημα να είναι πιο δυναμικό και ευφυές.

Τέλος, και οι τρεις εφαρμογές έχουν σαν στόχο την απλότητα και την καλύτερη εμπειρία χρήστη – τόσο για τους πράκτορες όσο για τους πελάτες. Και το *Freshservice* αλλά και το *Zendesk* είναι δομημένα με τέτοιο τρόπο που δεν απαιτούν ιδιαίτερη εκμάθηση από τους χρήστες. Τα πιο περίπλοκα σημεία παρέχουν ξεκάθαρες οδηγίες για την καθοδήγη του χρήστη. Σε αντίθεση με την δική μας εφαρμογή, δεν έχουν αύστηρά καθορισμένο κύκλο ζωής των αιτήσεων, που σημαίνει ότι κάθε κατάσταση έπιτρέπει διαφορετικές λειτουργίες πάνω στη αίτηση και κάνει πιο απλή τη διαδικασία διαχείρισης τους. Το *Zendesk,* ιδιαίτερα, παρέχει αυτοματισμούς για να μειώσει ακόμα περισσότερο τις δράσεις που πρέπει να πραγματοποιούν οι πράκτορες. Η δική μας εφαρμογή, από την άλλη, χρειάζεται κάποια εκμάθηση γιατί υποστηρίζει έναν συγκεκριμένο κύκλο ζωής των αιτήσεων. Τα βασικά, όμως, βήματα γίνονται αυτόματα από το σύστημα. Τα στοιχεία, επίσης, που καλείται να εισάγει ο πελάτης δεν είναι τόσο τεχνικά όσο περιγραφικά, διευκολύνοντας έτσι τη διαδικασία για αυτόν.

Συνεπώς, η SLASUP λύνει το πρόβλημα που θέσαμε αξιοποιώντας ένα πιο λεπτομερές SLA, όπως προτείναμε στην εργασία αυτή, στην βάση δεδομένων της ώστε οι προτεραιότητες των αιτήσεων να προκύπτουν από αυτό. Παρ'όλα αυτά, το επιτυγχάνει αυτό σε ένα σύστημα το οποίο χρειάζεται κάποια εκμάθηση και ειδική παραμετροποίηση στη βάση για κάθε διαφορετικό SLA. Επίσης, από άποψη λειτουργιών, τα εμπορικά συστήματα υποστηρίζουν πολύ περισσότερες λειτουργίες και κανάλια επικοινωνίας.

Κριτήριο\Σύστημα	Freshservice	Zendesk	SLASUP
Λειτουργίες	Πολλές λειτουργίες, τόσο όσον αφορά στα αιτήματα όσο και σε αναφορές, επικοινωνία κ.λ.π.	Υποστήριξη πολλών εξωτερικών εφαρμογών, λειτουργιών και διαύλων επικοινωνίας με τον πελάτη.	Περιορισμένες στα απολύτως απαραίτητα
Αξιοποίηση SLA	Μόνο όσον αφορά σε χρόνους απόκρισης/επίλυσης και πολιτικές λήξης.	Μόνο όσον αφορά σε χρόνους απόκρισης/επίλυσης.	Χρήση του SLA για ορισμό μεγεθών και βαρύτητας τους, χρόνουν επίλυσης καθώς και ρύθμιση των διαδικασιών μέσω του κύκλου ζωής των αιτήσεων
Διαδικασία υποστήριξης	Απλή, γρήγορη και πλήρης. Δεν απαιτείται κανένας χρόνος για την εκμάθηση του συστήματος.	Απλή και γρήγορη. Δεν απαιτείται χρόνος εκμάθησης του συστήματος. Επιτρέπει την εισαγωγή αυτοματισμών για ακόμα μεγαλύτερη απόδοση.	Απλή και γρήγορη αλλά απαιτείται κάποιος χρόνος για εκμάθηση των διαδικασιών.

Πίνακας 1. Συνολική σύγκριση των συτημάτων

6 Συμπεράσματα

Τα SLA είναι ένα βασικό τμήμα στη συνεργασία μεταξύ επιχειρήσεων διότι όχι μόνο θέτουν μία βάση για την επικοινωνία μεταξύ των εμπλεκόμενων ομάδων, των οποίων τα μέλη συνήθως έχουν διαφορετικά γνωσιακά πλαίσια, αλλά δηλώνουν δικαιώματα και υποχρεώσεις που διατηρούν οι δύο πλευρές κατά τη διάρκεια της συνεργασίας τους. Τα SLA, όμως, όπως ορίζονται συνήθως, εστιάζουν περισσότερο στις υποχρεώσεις του παρόχου και λιγότερο στο να θέσουν μια καλή βάση επικοινωνίας μεταξύ των εμπλεκόμενων ομάδων.

Το γεγονός αυτό έχει ιδιαίτερη σημασία στη διαδικασία υποστήριξης πελατών, μια σημαντική πτυχή της εργολαβίας έργων. Η διαδικασία αυτή στα σημερινά χρόνια γίνεται συνήθως μέσω κατάλληλων πληροφοριακών συστημάτων, τα οποία δέχονται τις αιτήσεις σφαλμάτων που υποβάλλουν οι πελάτες και, ανάλογα με την υποβληθήσα προτεραιότητα, θέτουν κάποιον χρονικό περιορισμό για την επίλυση του προβλήματος, πάντα σύμφωνα με κάποιο SLA. Ο τρόπος αυτός, όμως, σε συνδυασμό με τη γενική φύση των SLA, δημιουργεί περιπτώσεις κατάχρησης ή κακής συνεννόησης, δημιουργώντας αντίξοες συνθήκες για τις ομάδες που έχουν την ευθύνη διόρθωσης των βλαβών.

Με την εργασία αυτή, προτείνουμε ότι η προτεραιότητα των αιτήσεων υποστήριξης δεν θα πρέπει να δηλώνεται από τους πελάτες αλλά θα πρέπει να προκύπτει από τα χαρακτηριστικά του σφάλματος. Αυτό σημαίνει ότι στο εκάστοτε SLA θα πρέπει να τίθεται ένα σύνολο χαρακτηριστικών σφάλματος με δικές τους βαρύτητες από τα οποία θα προκύπτει η προτεραιότητα του σφάλματος και, κατ'επέκταση, της αίτησης. Τα χαρακτηριστικά αυτά θα πρέπει, προφανώς, να τα λαμβάνουν υπόψην και τα συστήματα υποστήριξης πελατών με την έννοια ότι ο πελάτης δεν θα εισάγει την προτεραιότητα αλλά θα εισάγει τιμές για τα υπόλοιπα χαρακτηριστικά και το σύστημα θα υπολογίζει την προτεραιότητα της αίτησης με βάση αυτά. Επίσης, το σύστημα θα παρέχει τη δυνατότητα αλλαγής της προτεραιότητας σε συγκεκριμένες ομάδες ή χρήστες.

Ως τμήμα της εργασίας αυτής, αναπτύξαμε μια εφαρμογή, τη SLASUP, η οποία υλοποιεί τα παραπάνω χαρακτηριστικά. Η εφαρμογή υποστηρίζει ένα SLA όπως το περιγράψαμε παραπάνω, το οποίο υλοποιείται στη βάση δεδομένων της εφαρμογής ώστε να είναι εύκολη η αλλαγή ή η διόρθωση τιμών. Επίσης, υποστηρίζει έναν παραμετροποιήσιμο κύκλο ζωής αίτησης μέσω ενός πίνακα στη βάση ο οποίος περιορίζει τις εναλλαγές κατάστασης της αίτησης ανάλογα με την ομάδα στην οποία ανήκει ο χρήστης που την πραγματοποιεί. Η προτεραιότητα προκύπτει βάση τριών χαρακτηριστικών: τον τύπο σφάλματος, το μέγεθος του αντίκτυπου που είχε το σφάλμα καθώς και τι τύπου πελάτες επηρεάστηκαν.

Πρώτη συνέπεια των παραπάνω είναι ένα πλήρες SLA, μέσα στο οποίο ορίζονται κατανοητοί και μετρήσιμοι όροι. Αυτό σημαίνει ότι οι πελάτες θα μπορούν να επικοινωνήσουν καλύτερα τα σφάλματα στους πράκτορες IT. Επίσης, τα συστήματα που προκύπτουν από την πρόταση μας περιορίζουν τη δυνατότητα κατάχρησης τους από τους πελάτες αλλά και επιτρέπουν την υποβολή αιτήσεων με πιο ακριβείς πληροφορίες ώστε τα πιο καίρια σφάλματα να αντιμετωπίζονται γρηγορότερα και πιο εύκολα. Έτσι, η διαδικασία υποστήριξης του πελάτη

γίνεται πιο αποδοτική με άμεση συνέπεια την καλύτερη ικανοποίηση του και, πιο έμμεσα, την μεγαλύτερη κερδοφορία της επιχείρησης.

Επίσης, η αξιολόγηση της εφαρμογής μας έναντι σε εμπορικά συστήματα αιτήσεων υποστήριξης έδειξε ότι ενώ λύνει το υπό μελέτη πρόβλημα, είναι πιο περίπλοκη ώστε να χρειάζεται κάποιο χρόνο εκμάθησης και υστερεί σε λειτουργίες και κανάλια επικοινωνίας.

Πρωταρχικός στόχος της εργασίας αυτής είναι η πρόταση της δικής μας λύσης στο πρόβλημα που περιγράψαμε. Επόμενο λογικό βήμα, συνεπώς, πρέπει να αποτελεί η πρόταση ενός μοντέλου πιο λεπτομερούς SLA, που θα πληρεί της προϋποθέσεις που περιγράψαμε, καθώς και η πιθανή τυποποίηση του από τους αρμόδιους οργανισμούς. Η εφαρμογή που αναπτύξαμε θα μπορούσε στο μέλλον να εξελιχθεί σε ένα πλήρες σύστημα διαχείρισης αιτήσεων υποστήριξης με πολλαπλά κανάλια επικοινωνίας, επιλογές για αναφορές και άλλες λειτουργίες ή σύγχρονους αυτοματισμούς ή θα μπορούσε να αποτελέσει πηγή έμπνευσης για άλλα συστήματα που θέλουν να κινηθούν προς την κατεύθυνση αυτή.

7 Βιβλιογραφικές Πηγές

1. Green, B., Seshadri, S. *AngularJS*. O'Reilly Media, 2013

2. *The Service Level Agreement Zone*. SLA Information Zone, 2015

3. Walls, C., Breidenback, R. *Spring in Action*. Manning

4. Friedman, D., Wise D. *The Impact of Applicative Programming on Multiprocessing*. International Conference on Parallel Processing, 1976

5. Kaplan, R. S, Norton, D. P. *The Balanced Scorecard – Measures That Drive Performance*. Harvard Business Review, 1992

6. Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000

7. Johnson, R. E., Foote B. *Designing Reusable Classes*. Journal of Object-Oriented Programming, Volume 1, Number 2

8. *"Εξωτερική ανάθεση"*, Ευρετήριο Οικονομικών Όρων

9. "What is Object/Relational Mapping", *Hibernate Overview*, JBOSS Hibernate

10. *Freshservice*, Freshworks

11. *Zendesk*, Zendesk