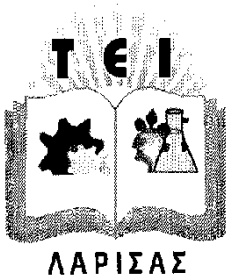


ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ
ΛΑΡΙΣΑΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποιήση εφαρμογής οικονομικής διαχείρησης καταστήματος στο cloud της Google με την χρήστη της Python

Καραγιάννης Χρήστος Τ-1435

ΕΠΙΒΛΕΠΩΝ: Σωμαράς Χρήστος, τίτλος, βαθμίδα

ΛΑΡΙΣΑ 2012

«Δηλώνω υπεύθυνα ότι το παρόν κείμενο αποτελεί προϊόν προσωπικής μελέτης και εργασίας και πως όλες οι πηγές που χρησιμοποιήθηκαν για τη συγγραφή της δηλώνονται σαφώς είτε στις παραπομπές είτε στη βιβλιογραφία. Γνωρίζω πως η λογοκλοπή αποτελεί σοβαρότατο παράπτωμα και είμαι ενήμερος/η για **την** ***επέλευση των νομίμων συνεπειών»***

ΠΑΡΑΡΤΗΜΑ Δ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή

Τόπος, Ημερομηνία

ΕΠΙΤΡΟΠΗ ΑΞΙΟΛΟΓΗΣΗΣ

1. Ονοματεπώνυμο, Υπογραφή

2. Ονοματεπώνυμο, Υπογραφή

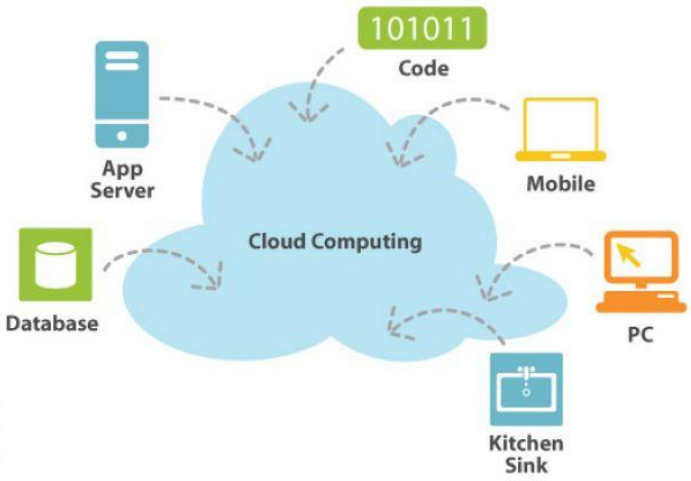
3. Ονοματεπώνυμο, Υπογραφή

Περιεχόμενα

Πίνακας περιεχομένων εικόνων και κώδικα

Μέρος Ι - Εισαγωγή

Με τον όρο cloud computing αναφερόμαστε στην παροχή υπολογιστικής ισχύος, χώρου αποθήκευσης, δικτυακού εξοπλισμού και όλων των μέσων όπου μπορούν επιχειρήσεις ή και ομάδες ανθρώπων να χρησιμοποιήσουν χωρίς να χρειάζεται να κατέχουν ανάλογο εξοπλισμό. Αποτελεί μια υπηρεσία για την παροχή πόρων, λογισμικού και δεδομένων μέσα από ένα δίκτυο, συνήθως το διαδίκτυο.



Το κύριο μοντέλο λειτουργίας του cloud computing απαρτίζεται από τον συνδυασμό υπολογιστικού , δικτυακού και αποθηκευτικού εξοπλισμού για την απόδοση των παραπάνω δυνατοτήτων της πληροφορικής ως υπηρεσία στον τελικό χρήστη. Ο χρήστης δεν χρειάζεται πλέον να κατέχει τεχνικές γνώσεις πάνω στα υπολογιστικά συστήματα για να δημιουργήσει μια εφαρμογή. Επίσης δεν είναι απαραίτητο να διαθέτει χώρους εγκατάστασης εξοπλισμού για την εύρυθμη λειτουργία της εφαρμογής του.

Μια επιχείρηση για παράδειγμα θα έπρεπε να αγοράζει άδειες για τις εφαρμογές των υπαλλήλων της και να απασχολεί επιπλέον τεχνικό προσωπικό για την παρακολούθηση και συντήρηση του εξοπλισμού. Επίσης να διαθέτει ένα χώρο εγκατάστασης των εξυπηρετητών, των συσκευών αποθήκευσης, των δικτυακών συσκευών και επαρκή ηλεκτρολογικό εξοπλισμό για την λειτουργία του πληροφοριακού συστήματος. Ο χώρος αυτός θα πρέπει να είναι προστατευμένος και από άλλες φυσικές καταστροφές όπως σεισμοί και πυρκαγιές ώστε να μην δημιουργηθεί πρόβλημα στον εξοπλισμό. Αυτό ως αποτέλεσμα έχει την αύξηση του κόστους των υπηρεσιών της κάθε επιχείρησης.

Το cloud computing επιτρέπει την πρόσβαση και χρησιμοποίηση υπολογιστικών συστημάτων και πόρων, που βρίσκονται σε απομακρυσμένες τοποθεσίες από τον τελικό χρήστη, μέσω του διαδικτύου. Η πρόσβαση αυτή επιτυγχάνεται μέσω των cloud υπηρεσιών τις οποίες και μπορούμε να τις διακρίνουμε σε τρείς κατηγορίες.

Κατηγορίες υπηρεσιών:

- Η υποδομή του υπολογιστικού συστήματος ως υπηρεσία όπου μπορούν να χρησιμοποιηθούν οι υποδομές-εξοπλισμός για την ανάπτυξη ενός συνόλου εφαρμογών. Ένα τέτοιο παράδειγμα είναι το Amazon elastic compute cloud (Amazon EC2) όπου ο τελικός χρήστης χρησιμοποιεί την υπηρεσία της Amazon για να εγκαταστήσει εικονικούς server και τους οποίους να παραμετροποιήσει όπως θέλει.
- Πλατφόρμες ως υπηρεσία όπου μπορούν να αναπτυχθούν εφαρμογές έτοιμες για να χρησιμοποιηθούν χωρίς κάποια εγκατάσταση ή παραμετροποίηση των υπολογιστικών συστημάτων που θα χρησιμοποιηθούν. Ένα τέτοιο παράδειγμα είναι το app engine της Google, όπου ο τελικός χρήστης θα πρέπει να αναπτύξει στις τρείς από τις υποστηριζόμενες γλώσσες (Python, Java, Go) την cloud εφαρμογή του.
- Λογισμικό ως υπηρεσία όπου ο τελικός χρήστης μπορεί να χρησιμοποιεί λογισμικό που δεν χρειάζεται να εγκαταστήσει στον υπολογιστή του, και μπορεί να έχει πρόσβαση σε αυτό μέσω ενός φυλλομετρητή (Browser).

Το cloud computing μπορεί να χρησιμοποιηθεί από οποιονδήποτε έχει πρόσβαση σε αυτό και θα πρέπει να είναι διαθέσιμο συνεχώς χωρίς να δημιουργούνται διακοπές. Επίσης τα δεδομένα που αποθηκεύονται στις εφαρμογές αυτές θα πρέπει να είναι ασφαλή και να μην μπορούν άλλοι μη εξουσιοδοτημένοι χρήστες να τα χρησιμοποιήσουν.

Ένα ακόμη πλεονέκτημα των cloud εφαρμογών είναι ότι είναι ελαστικές, δηλαδή ανάλογα με την ζήτηση που έχουν από τους χρήστες να χρησιμοποιούν δυναμικά και περισσότερους πόρους από το υπολογιστικό σύστημα στο οποίο φιλοξενούνται. Έτσι μια εφαρμογή με μικρή ζήτηση δεν θα χρησιμοποιεί το υπολογιστικό σύστημα για μεγάλα χρονικά διαστήματα, ενώ μια εφαρμογή με μεγάλη ζήτηση, θα έχει στην διάθεση της επαρκή υπολογιστική ισχύ και αποθηκευτικό χώρο, για την διεκπεραίωση των αιτημάτων που της ζητούν οι χρήστες.

Επίσης διατίθεται ένα περιβάλλον ελέγχου της cloud υπηρεσίας. Μέσω αυτού του περιβάλλοντος ο διαχειριστής μπορεί να ελέγχει την εφαρμογή του, να παρακολουθεί την κίνηση και των ρυθμό των αιτημάτων, να βλέπει πως συμπεριφέρεται η εφαρμογή και πόση υπολογιστική ισχύ χρησιμοποιεί αλλά και το αν δημιουργούνται σφάλματα από τα αιτήματα προς την εφαρμογή. Εάν του δίνεται η δυνατότητα μπορεί επίσης να θέτει όρια χρήσης της εφαρμογής ώστε να επιτύχει το επιθυμητό αποτέλεσμα όσον αφορά την κοστολόγηση της χρήσης της υπηρεσίας. Μπορεί να θέσει όριο ως προς τον χρόνο εκτέλεσης, το πόσες ταυτόχρονες διεργασίες θα δημιουργούνται για να καλύψουν την ζήτηση καθώς και την δικτυακή κίνηση που δημιουργούν τα αιτήματα. Χρησιμοποιώντας κάποιος τις cloud υπηρεσίες θα πρέπει να καταβάλει κάποιο χρηματικό ποσό. Συνήθως η χρέωση υπολογίζεται από τον χρόνο που χρησιμοποίησε η εφαρμογή το υπολογιστικό σύστημα όπως τον χρόνο που χρειάστηκε για την επεξεργασία δομένων και την ποσότητα δικτυακής κίνησης που δημιούργησε.

App engine

Για την πτυχιακή μου επέλεξα να χρησιμοποιήσω την cloud υπηρεσία της Google, το app engine. Η εφαρμογή μου βρίσκεται στο http://storeroomapp.appspot.com. Κυκλοφόρησε για πρώτη φορά σε δοκιμαστική έκδοση τον Απρίλιο του 2008 και βγήκε από την δοκιμαστική της λειτουργία μόλις τον Σεπτέμβριο του 2011. Ακολουθεί την κατηγόρια πλατφόρμα ως υπηρεσία για την ανάπτυξη και φιλοξενία των δικτυακών εφαρμογών. Με τον όρο δικτυακή εφαρμογή εννοούμε μία εφαρμογή ή υπηρεσία η οποία είναι προσβάσιμη μέσω του διαδικτύου και μπορούμε να την χρησιμοποιήσουμε συνήθως με έναν φυλλομετρητή.

Η πλατφόρμα που χρησιμοποιείται είναι ένα περιβάλλον στο οποίο μπορεί κάποιος να αναπτύξει μια εφαρμογή στις τρείς από τις υποστηριζόμενες γλώσσες (Python, Java, Go) και να χρησιμοποιήσει το app engine για την λειτουργία της. Έχει πρόσβαση με αυτό τον τρόπο στο cloud δίκτυο της Google για τη φιλοξενία της εφαρμογής, για την εκτέλεση της και για την αποθήκευση των δεδομένων.

Το app engine μπορεί να φιλοξενήσει από μία ιστοσελίδα με απλό στατικό περιεχόμενο όπως κείμενο με εικόνες μέχρι και πολυπλοκότερες εφαρμογές που αλληλεπιδρούν με δίκτυα κοινωνικής δικτύωσης ή με εφαρμογές κινητών τηλεφώνων. Αναπτύχθηκε ώστε να μπορεί να εξυπηρετήσει εφαρμογές πραγματικού χρόνου, όπου η απόκριση παίζει σημαντικό ρόλο για την προσπέλαση δεδομένων. Είναι ένα δυναμικό

περιβάλλον το οποίο μπορεί να εξυπηρετεί ταυτόχρονα πολλούς χρήστες χωρίς να χάνεται η απόδοση του συστήματος. Όσο περισσότερους χρήστες χρειάζεται να εξυπηρετήσει τόσο περισσότερους πόρους χρησιμοποιεί και διαχειρίζεται το app engine χωρίς η εφαρμογή να επηρεάζεται.

Οι εφαρμογές που αναπτύσσονται για το app engine μπορούν να είναι άμεσα διαθέσιμες σε όλο τον κόσμο. Με το που αποσταλεί η εφαρμογή από τον προγραμματιστή, το σύστημα αναλαμβάνει να αντιγράψει την εφαρμογή σε όλα τα data center της Google ανά τον κόσμο ώστε να ελαχιστοποιούνται οι χρόνοι χρήσης της υπηρεσίας, ανεξάρτητα από την γεωγραφική τοποθεσία του τελικού χρήστη. Επίσης τα δεδομένα για την κάθε εφαρμογή που δημιουργούνται, αποθηκεύονται ή και διαγράφονται είναι άμεσα διαθέσιμα για προσπέλαση από όλους, μιας και τα αιτήματα αυτά εκτελούνται σε όλα τα data center που φιλοξενούν την εφαρμογή.

Το app engine μπορεί να χωριστεί σε τρία μέρη:

- Το runtime περιβάλλον
- Το Datastore
- και τις επεκτάσιμες υπηρεσίες (scalable services)

Το runtime περιβάλλον

Οι εφαρμογές που βρίσκονται στο app engine δεν τρέχουν συνέχεια πάνω στους εξυπηρετητές. Για να ξεκινήσει μία εφαρμογή θα πρέπει να γίνει ένα αίτημα προς το app engine συνήθως από έναν φυλλομετρητή. Το οποίο έπειτα αναλαμβάνει, σύμφωνα με το domain name της διεύθυνσης που ζητήθηκε, να βρει την αντίστοιχη εφαρμογή και να επιλέξει έναν εξυπηρετητή για να διεκπεραιώσει το αίτημα, με βάση ποιός είναι πιθανότερο να επεξεργαστεί το αίτημα γρηγορότερα.

Μέσα στον κώδικα της εφαρμογής ορίζουμε τους χειριστές - handlers που θα επεξεργάζονται τα αιτήματα. Ο χειριστής ανάλογα με το περιεχόμενο που του ζητείται ενεργοποιεί και το αντίστοιχο κώδικα της εφαρμογής για την εξυπηρέτηση τους αιτήματος. Το runtime περιβάλλον ξεκινάει όταν κάποιος από αυτούς τους χειριστές κληθεί να επεξεργαστεί το αίτημα και σταματάει με τον τερματισμό του χειριστή. Το περιβάλλον αυτό εγγυάται ότι κάθε εφαρμογή δεν θα μπορεί να επηρεάσει κάποια άλλη η οποία τρέχει πάνω στον ίδιο εξυπηρετητή. Επίσης επιτυγχάνεται η δυνατότητα να εκτελούνται περισσότερες της μίας εφαρμογές χωρίς να δημιουργούνται προβλήματα.

Το app engine δεν επιτρέπει στις εφαρμογές να κάνουν άμεση χρήση του υλικού του εξυπηρετητή στον οποίο τρέχουν. Μια εφαρμογή μπορεί να διαβάσει μόνο τα αρχεία που αφορούν την ίδια αλλά δεν έχει πρόσβαση σε δεδομένα άλλων εφαρμογών. Επίσης δεν δίνεται η δυνατότητα για εγγραφή αρχείων πάνω στο σύστημα αρχείων (file system) του εξυπηρετητή. Μια εφαρμογή δεν μπορεί να χρησιμοποιήσει άμεσα το υλικό, όπως για παράδειγμα της κάρτας δικτύου, αλλά μπορεί να κάνει αιτήματα και να λάβει απαντήσεις κάνοντας χρήση υπηρεσιών που αναλαμβάνουν τον ρόλο του διαμεσολαβητή.

Επιπλέον εκτός των περιορισμών πρόσβασης το runtime περιβάλλον θέτει περιορισμούς και ως προς την χρήση της κεντρικής μονάδας επεξεργασίας αλλά και της μνήμης. Αν για παράδειγμα κάποια εφαρμογή χρησιμοποιεί για πολύ ώρα, περισσότερους πόρους από τις υπόλοιπες το app engine αναλαμβάνει να την περιορίσει, ώστε να υπάρχει μια δίκαιη κατανομή πόρων σε όλες τις εφαρμογές.

Το python runtime περιβάλλον τρέχει εφαρμογές που είναι γραμμένες στην έκδοση python 2.5 και δοκιμαστικά στην έκδοση python 2.7 . Το app engine βασίζεται στο CGI για την εκκίνηση των εφαρμογών του, οι οποίες μπορούν έπειτα να κάνουν χρήση των βιβλιοθηκών της Python καθώς και των API και των υπηρεσιών για την εκτέλεση τους. Πολλά ανοιχτού κώδικα framework όπως το Django, web2py και το Pylons υποστηρίζονται από το app engine, όπως επίσης δίνεται και η δυνατότητα στους προγραμματιστές να δημιουργήσουν τις δικές τους βιβλιοθήκες και να τις ενσωματώσουν μέσα στον κώδικα τους της εφαρμογής τους.

Το app engine γνωρίζει ποιό περιβάλλον να χρησιμοποιήσει μιας και με το που ξεκινάει μια οντότητα της εφαρμογής διαβάζει το αρχείο app.yaml . Μέσα στο αρχείο αυτό μπορούμε να δηλώσουμε ποιο περιβάλλον να χρησιμοποιήσει το app engine με τον εξής τρόπο:

```
runtime: python
api_version: 1
...
```

Αρχικά δηλώνουμε στην μεταβλητή runtime ποια γλώσσα - περιβάλλον θα χρησιμοποιήσουμε για την ανάπτυξη της εφαρμογής μας. Υπάρχουν τέσσερις επιλογές μέχρι στιγμής για τις τρείς γλώσσες που υποστηρίζονται:

- python
- python27
- java
- go

Στο api_version επιλέγουμε ποιά έκδοση του περιβάλλοντος θα χρησιμοποιήσουμε. Μέχρι στιγμής η Google χρησιμοποιεί μία έκδοση, την πρώτη. Εάν στο μέλλον υπάρξουν και άλλες εκδόσεις τότε θα πρέπει ο προγραμματιστής να δηλώσει ποιά πρέπει να χρησιμοποιεί το app engine στην εκκίνηση των οντοτήτων ώστε να μην υπάρχουν προβλήματα με τον κώδικα της εφαρμογής.

Όταν ένας server του app engine λάβει ένα αίτημα, συγκρίνει το ζητούμενο URL με τα URL πρότυπα (patterns) που έχουν δηλωθεί μέσα στο app.yaml αρχείο. Ο προγραμματιστής μπορεί να συνδέσει το κάθε πρότυπο με ένα python script που θα αναλαμβάνει να εκτελέσει το αίτημα. Ο server χρησιμοποιεί το CGI ως διεπαφή για την επικοινωνία του με τον python κώδικα, μέσα από μια διεργασία όπου χρησιμοποιούνται μεταβλητές περιβάλλοντος (environment variables) και η είσοδος - έξοδος της εφαρμογής για να διεκπεραιώνεται η επικοινωνία. Ο προγραμματιστής μπορεί να χρησιμοποιήσει διάφορα application framework, όπως το Django που είναι το προκαθορισμένο, ή και να αναπτύξει της δικιά του διεπαφή για την εκτέλεση των παραπάνω διεργασιών.

Το app engine έχει σχεδιαστεί ώστε να κρατάει στην μνήμη του τον κώδικα, τα αρχεία και δεδομένα που χρησιμοποιήθηκαν από την εφαρμογή, με σκοπό να επιταχύνουν την εκτέλεση των αιτημάτων. Εάν δέχεται πολλά αιτήματα τότε μπορεί να ξεκινήσει καινούργιες οντότητες της εφαρμογής και σε άλλους server για να διαμοιράσει το φορτίο. Δεν εγγυάται ότι οι οντότητες αυτές θα παραμείνουν για κάποιο χρονικό διάστημα στην μνήμη των εξυπηρετητών, ούτε και ότι οι οντότητες αυτές θα βρίσκονται στον ίδιο εξυπηρετητή.

Εάν μια εφαρμογή δεν δεχτεί αιτήματα για κάποιο χρονικό διάστημα τότε το app engine καθαρίζει την μνήμη από αυτή ώστε να ελευθερώσει πόρους για νέες διεργασίες εφαρμογών. Επειδή μια εφαρμογή μπορεί να διαγραφεί από την μνήμη και τα δεδομένα χρησιμοποιούν μεταβλητές του περιβάλλοντος, για να κρατήσουν τις τιμές τους, δεν θα πρέπει να αποθηκεύονται δεδομένα πάνω σε αυτή. Για αυτό τον σκοπό θα πρέπει να χρησιμοποιούνται το datastore για την μόνιμη ή το memcache για την προσωρινή αποθήκευση.

Εκτός του δυναμικού περιεχομένου, όπου το app engine δημιουργεί μέσα από τον κώδικα της εφαρμογής που τρέχει στο runtime environment, χρησιμοποιείται και ένα πλήθος από ξεχωριστούς εξυπηρετητές ώστε να διαμοιράζεται και το στατικό περιεχόμενο. Μαζί με την εφαρμογή ο developer μπορεί να χρησιμοποιήσει και αρχεία όπως html, css, JavaScript, φωτογραφίες τα οποία δεν χρειάζεται να περάσουν από κάποια επεξεργασία. Έτσι όταν ο χρήστης κάνει κάποιο αίτημα προς την εφαρμογή το στατικό περιεχόμενο στέλνετε γρηγορότερα. Τα αρχεία αυτά μπορούμε να τα αποθηκεύσουμε σε έναν φάκελο μέσα στην εφαρμογή μας.

Στον παρακάτω κώδικα έχουμε ορίσει το στατικό περιεχόμενο μας στον φάκελο static καθώς και ότι όλα τα αιτήματα μας θέλουμε να εξυπηρετούνται από το script mail.py:

```
handlers:
- url: /static
  static_dir: static

- url: .*
  script: main.py
```

Ένα ακόμα από τα σημαντικότερα στοιχεία στο runtime περιβάλλον είναι ότι δεν εμφανίζει το λειτουργικό σύστημα στην εφαρμογή. Η Google έχει αφαιρέσει από την Python μερικές από της δυνατότητες της ώστε να ενισχύσει την ασφάλεια των εφαρμογών αλλά και των εξυπηρετητών της. Κάθε εφαρμογή τρέχει μέσα στο δικό της sandbox περιβάλλον υπό ορισμένους περιορισμούς:

- Δεν μπορεί να δημιουργεί επιπρόσθετες διεργασίες ή επιπλέον νήματα διεργασιών. Η επεξεργασία θα πρέπει να γίνεται από τον χειριστή του κάθε αιτήματος της.
- Δεν της δύνεται η δυνατότητα να δημιουργεί απευθείας συνδέσεις με άλλες ιστοσελίδες ή υπολογιστές. Για αυτό τον σκοπό θα πρέπει να χρησιμοποιηθεί κάποια από τις υπηρεσίες μέσω του API της.
- Μπορεί να διαβάζει από το file system του server μόνο τα δικά της αρχεία και κώδικα. Δεν μπορεί να δημιουργήσει, αλλάξει ή να διαγράψει αρχεία έστω και της ανήκουν. Για να αποθηκεύσει δεδομένα θα πρέπει να χρησιμοποιήσει την υπηρεσία Datastore μέσω του API της.
- Επίσης δεν μπορεί να παρεμβάλλεται σε διεργασίες άλλων εφαρμογών ακόμα και αν ανήκουν στην ίδια εφαρμογή και τρέχουν ταυτόχρονα.

Datastore

Σε ένα περιβάλλον σαν αυτό του app engine υπάρχουν μεγάλες απαιτήσεις για αποθηκευτικούς χώρους, με μικρούς χρόνους απόκρισης ανάκτησης και αποθήκευσης δεδομένων, που θα υποστηρίζει ταυτόχρονα πολλούς χρήστες και θα μπορεί να κλιμακώνεται ανάλογα με την ζήτηση πληροφοριών.

Οι πιο επιτυχημένες έως τώρα εφαρμογές στηριζόντουσαν σε μοντέλα αποθήκευσης όπου ένας κεντρικός εξυπηρετητής (server) αναλάμβανε τον ρόλο αυτό. Συνήθως ένας ή περισσότεροι web servers συνδεόντουσαν πάνω σε αυτόν ώστε να αποθηκεύσουν ή να ανακτήσουν δεδομένα. Το μοντέλο της κεντρικής αποθήκευσης είχε το πρόβλημα ότι δεν μπορούσε εύκολα να κλιμακωθεί. Όταν οι συνδέσεις θα έφταναν στο μέγιστο επιτρεπτό όριο, το οποίο του είχαν ορίσει, τότε δεν θα είχε άλλους διαθέσιμους πόρους να χρησιμοποιήσει ώστε να εξυπηρετήσει την ζήτηση.

Επίσης το μοντέλο αυτό ήταν περιορισμένο και ως προς τους πόρους συστήματος. Εκτός από το λογισμικό που θα αναλάμβανε την αποθήκευση πάνω στο server, θα πρέπει να φορτώνεται και το λειτουργικό σύστημα. Επεξεργαστική ισχύ καθώς και μνήμη θα καταλαμβάνονταν για διεργασίες που είναι απαραίτητες για την εύρυθμη λειτουργία του συστήματος.

Η λύση σε αυτό δόθηκε με την χρησιμοποίηση τεχνολογιών cluster όπου είναι ο πρόδρομος του cloud computing. Αντί για κάποιο κεντρικό server, δόθηκε η δυνατότητα, μια συστοιχία από μηχανήματα να ενοποιηθούν και αναλάβουν το ρόλο αυτό. Το περιβάλλον λειτουργίας του cluster αναλάμβανε να κρύψει τις συστοιχίες αυτές και να τις εμφανίζει ως μία μηχανή. Έτσι ανάλογα με την ζήτηση για επεξεργασία και αποθήκευση μπορούσαν να βρεθούν επιπλέον πόροι, μιας και το φορτίο δεν το αναλάμβανε μία μηχανή αλλά η συστοιχία από αυτές.

Μπορούσαν πλέον να χρησιμοποιηθούν συστοιχίες όπου κάθε μία από αυτές θα αναλάμβανε και διαφορετικό ρόλο όπως την εξυπηρέτηση των web αιτημάτων, την αποθήκευση δεδομένων αλλά και τον υπολογισμό πολύπλοκων διεργασιών. Επιτυγχάνονταν έτσι μια εξισορρόπηση του φορτίου που αναλάμβανε ο κάθε εξυπηρετητής καθώς και αναπτύσσονταν πιο εξειδικευμένα λογισμικά για τον κάθε ρόλο. Επίσης η διαχείριση αυτών των συστοιχιών μπορούσε πλέον να γίνεται με πιο αποδοτικό και γρήγορο τρόπο.

Το πιο επιτυχημένο είδος συστήματος για την αποθήκευση, επεξεργασία και ανάκτηση δεδομένων τα τελευταία χρόνια είναι οι σχεσιακές βάσεις δεδομένων, όπου γραμμές και στήλες πινάκων, χρησιμοποιούνται για την αποδοτική αναπαράσταση πληροφοριών. Ιδιαίτερο πλεονέκτημα αποτελεί η δυνατότητα, που δίνουν αυτά τα συστήματα, ώστε να γίνονται ενώσεις (joins) μεταξύ των πινάκων που κρατούν τα δεδομένα, υπό ορισμένα κριτήρια, με σκοπό να υλοποιούνται πολύπλοκα ερωτήματα, πάνω στην σχεσιακή βάση δεδομένων. Άλλα συστήματα χρησιμοποιούν ιεραρχικές δομές όπως η τεχνολογία xml ή βάσεις αντικειμένων (OODBMS-object databases). Κάθε είδος έχει τα πλεονεκτήματα και τα μειονεκτήματα του και το πιο επιλέγεται κάθε φορά έχει να κάνει με τα δεδομένα τα οποία πρόκειται να αποθηκευτούν και το πώς θα ζητείται πρόσβαση σε αυτά.

Η Google για το app engine επέλεξε ένα σύστημα που μοιάζει με αυτά των βάσεων αντικειμένων. Είναι βασισμένο στην τεχνολογία BigTable και ονομάζεται datastore. Το BigTable είναι ένα ευρείας κλίμακας, κατανεμημένο σύστημα αποθήκευσης και διαχείρισης δομημένων δεδομένων. Αναπτύχθηκε από την ίδια την εταιρία για την εξυπηρέτηση των δικών της αναγκών. Πάνω από 60 υπηρεσίες όπως το web indexing, Google earth, Google finance, Google analytics και το Orkut το χρησιμοποιούν.

Σχεδιάστηκε ώστε να καλύψει ένα εύρος απαιτήσεων από τις υπηρεσίες που το χρησιμοποιούν. Από αποθηκευτικό χώρο για τις πληροφορίες ιστοσελίδων ή δορυφορικές φωτογραφίες, μέχρι και εξυπηρέτηση αιτημάτων όπου ο χρόνος απόκρισης παίζει σημαντικό ρόλο όπως για παράδειγμα εφαρμογές πραγματικού χρόνου.

Το datastore αποθηκεύει τα δεδομένα σαν οντότητες, όπου μπορούν να έχουν μία ή περισσότερες ιδιότητες. Κάθε ιδιότητα έχει ένα όνομα, που την ξεχωρίζει από τις υπόλοιπες, και μία τιμή για τα δεδομένα. Η εφαρμογή χρησιμοποιεί το αντίστοιχο API για να ορίσει το μοντέλο δεδομένων και δημιουργεί τις οντότητες που θα αποθηκευτούν. Το datastore υποστηρίζει κάποιους συγκεκριμένους τύπους για τις ιδιότητες των αντικειμένων, μια ενημερωμένη και πλήρη λίστα των οποίων βρίσκεται στην σελίδα code.google.com/intl/el-GR/appengine/docs/python/datastore/typesandpropertyclasses.html .

Ένα παράδειγμα δημιουργίας ενός μοντέλου κατάλληλου για αποθήκευση στο datastore φαίνεται παρακάτω:

```
from google.appengine.ext import db

class Student(db.Model):
    onoma = db.StringProperty(required=True)
    ar_mitroou = db.StringProperty(required=True)
    im_genisis = db.DateProperty(required=True)
    tmima = db.StringProperty(required=True, choices=set(["cs", "ee", "me"]))
    im_eggrafis = db.DateProperty(auto_now=True)
```

Στην αρχή του κώδικα δηλώνεται η βιβλιοθήκη που θα χρησιμοποιήσουμε. Φτιάχνουμε την κλάση Student και ορίζουμε τις ιδιότητες της κλάσης που θα χρησιμοποιήσει το datastore API για να δημιουργήσει την οντότητα που θα αποθηκευτεί, αλλά και να επικυρώσει τις τιμές των ιδιοτήτων με τους αντίστοιχους τύπους που ορίστηκαν στο μοντέλο. Το onoma, ο ar_mitroou και το tmima είναι τύπου συμβολοσειράς (StringProperty), ενώ τα im_genisis, im_eggrafis είναι για ημερομηνία και ώρα (DateProperty). Επίσης με το required=True δηλώνουμε ποιά από αυτά θέλουμε οπωσδήποτε να έχουν κάποια τιμή ενώ στο tmima με το choices δηλώνουμε ότι οι δυνατές επιλογές για την τιμή του είναι ότι βρίσκεται μέσα στο set. Το auto_now = True δηλώνει ότι η ημερομηνία θα μπαίνει αυτόματα με το που αποθηκεύετε το αντικείμενο στο datastore.

Στον παραπάνω κώδικα το μοντέλο μπορούμε να το συγκρίνουμε με τους πίνακες των σχεσιακών βάσεων δεδομένων όπου σε κάθε γραμμή αποθηκεύουμε τις οντότητες, κάποιας κλάσης, και σε κάθε στήλη τις ιδιότητες τους. Υπάρχουν όμως κάποιες βασικές διαφορές στην αναγωγή της σύγκρισής τους με τις σχεσιακές. Μια οντότητα δεδομένου τύπου δεν είναι αναγκαίο να έχει τις ίδιες ιδιότητες με άλλες του ίδιου τύπου και επίσης δύο οντότητες μπορούν να έχουν δηλωμένο το ίδιο όνομα για κάποια ιδιότητα τους, αλλά ο τύπος δεδομένων που δέχονται να είναι διαφορετικός. Επίσης μπορούμε για κάθε οντότητα να έχουμε περισσότερες από μία τιμές σε μία ιδιότητα. Αυτή η σχεδίαση του datastore το καθιστά ευέλικτο.

Με την αποθήκευση μιας οντότητας το datastore του εκχωρεί και ένα κλειδί που είναι μοναδικό για κάθε εγγραφή μέσα σε αυτό. Το κλειδί αυτό μπορεί να δημιουργηθεί και από την ίδια την εφαρμογή και να δηλωθεί στο αντικείμενο ώστε ο προγραμματιστής να έχει την δυνατότητα να κρατάει ένα πρότυπο για τα κλειδιά που θα εκχωρούνται στις εγγραφές. Η ανάκτηση οντοτήτων γίνεται πολύ γρηγορότερα αν γνωρίζουμε το κλειδί τους καθώς και η εύρεση αντικειμένων με κριτήρια τα κλειδιά τους. Με το που δημιουργηθεί το κλειδί στην

εγγραφή τότε δεν δίνεται η δυνατότητα να αλλάξει. Επίσης χρησιμοποιείται για να καθοριστεί που βρίσκονται αποθηκευμένα τα δεδομένα, ανάμεσα στις συστοιχίες από server.

Επίσης για την διαχείριση και ανάκτηση δεδομένων αναπτύχθηκε η Gql. Η σύνταξη της γλώσσας μοιάζει με αυτή της Sql. Αν και πολλές δυνατότητες έχουν μείνει ίδιες κάποιες όπως το join μεταξύ πινάκων έχουν καταργηθεί, μιας και η ανάκτηση δεδομένων με αυτόν τον τρόπο αποδείχτηκε μη αποδοτικός, όταν το ερώτημα έπρεπε να εκτελεστεί πάνω σε διαφορετικούς server.

Η Google ανακοίνωσε ότι στο τέταρτο τρίμηνο του 2011 θα αρχίσει και η υποστήριξη σχεσιακών βάσεων δεδομένων για την αποθήκευση δεδομένων της app engine εφαρμογής. Ο προγραμματιστής εκτός από το datastore θα μπορεί να χρησιμοποιήσει το Google cloud sql για να δημιουργήσει οντότητες από MySql βάσεις δεδομένων. Επίσης δίνεται η δυνατότητα, στις οντότητες να έχουν πρόσβαση δύο ή και περισσότερες εφαρμογές ταυτόχρονα, ώστε να μοιράζονται κοινούς πόρους.

Scalable services

Πολλές από τις δυνατότητες του app engine δεν χρειάζεται να τις σχεδιάσει και να τις υλοποίηση ο προγραμματιστής. Η Google του δίνει την δυνατότητα να χρησιμοποιεί υπηρεσίες για την εκτέλεση εργασιών με κλήση υπηρεσιών. Ένα τέτοιο παράδειγμα είναι το datastore το οποίο παρέχεται μέσω υπηρεσίας. Η εφαρμογή χρησιμοποιεί το API της κάθε υπηρεσίας ώστε να έχει πρόσβαση στο ξεχωριστό σύστημα του οποίου τις δυνατότητες θέλει να χρησιμοποιήσει.

Επιτυγχάνεται έτσι ελαχιστοποίηση, από την μεριά του προγραμματιστή, του χρόνου ανάπτυξης της εφαρμογής καθώς και της πολυπλοκότητας της. Μπορεί να επικεντρωθεί στην εγγραφή ποιό αποδοτικού κώδικα αλλά και αποσφαλμάτωση της εφαρμογής του. Το πώς θα αναπτύσσονται και θα κλιμακώνονται τα δεδομένα, πάνω στην κάθε υπηρεσία, το αναλαμβάνει το αντίστοιχο σύστημα που την έχει αναλάβει.

Μερικές από τις χρησιμότερες υπηρεσίες παρουσιάζονται παρακάτω:

- Το datastore που χρησιμοποιείτε κυρίως για να αποθηκεύονται δεδομένα της εφαρμογής. Σχεδιάστηκε ώστε να κλιμακώνεται ανάμεσα σε πολλούς server και να

έχει μικρούς χρόνους ανάκτησης και αποθήκευσης δεδομένων. Παρέχει ασφάλεια και συνέπεια στις αποθηκευμένες πληροφορίες.

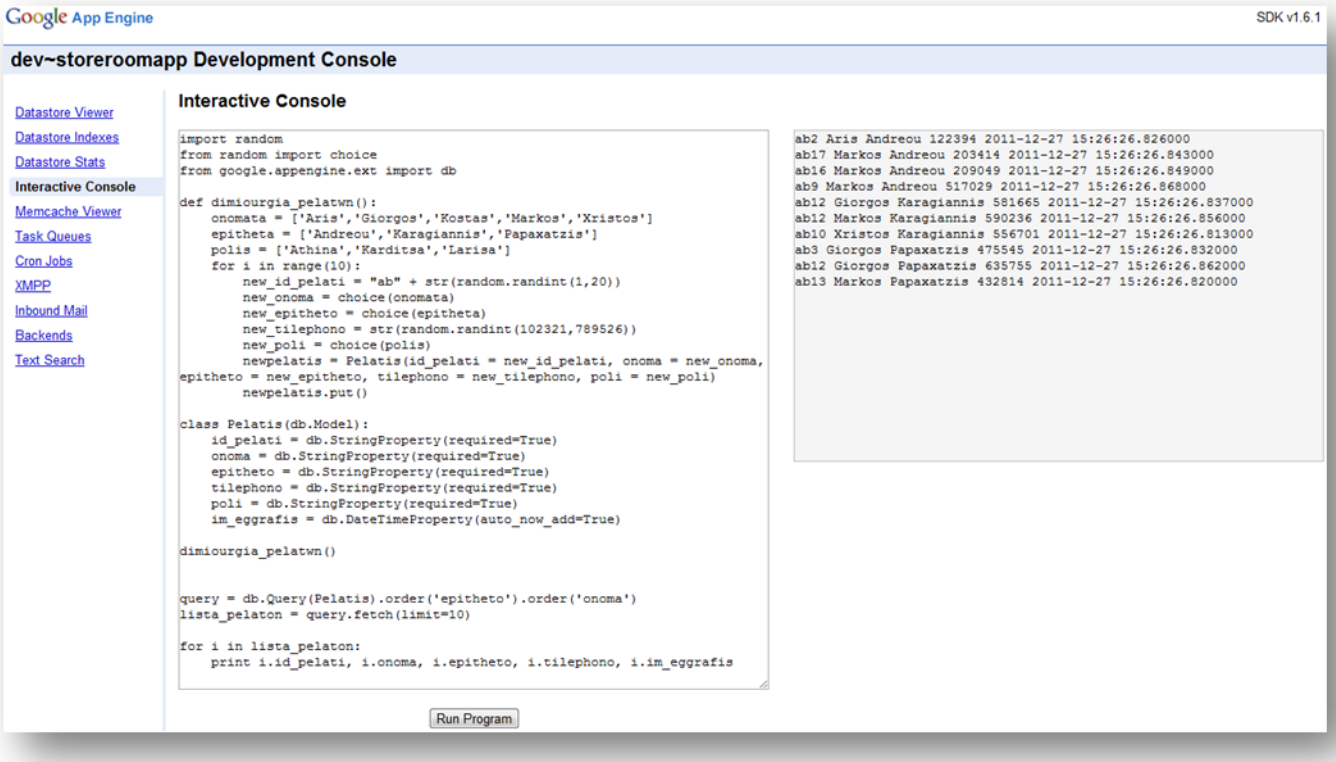
- Για τους ίδιους σκοπούς χρησιμοποιείτε και το memcache που όμως αντί για μόνιμη αποθήκευση παρέχει προσωρινή αποθήκευση σε δεδομένα που δεν είναι πολύ σημαντικά και χρησιμοποιούνται περισσότερο. Είναι γρηγορότερο από το datastore μιας και ως μέσο αποθήκευσης χρησιμοποιεί την μνήμη Ram.
- Το Mail API όπου οι εφαρμογές μπορούν να το χρησιμοποιήσουν για να στέλνουν email είτε για λογαριασμό του διαχειριστή είτε των χρηστών. Επίσης λαμβάνονται email για λογαριασμό της εφαρμογής.
- Με το Users API γίνεται η αυθεντικοποίηση των χρηστών της εφαρμογής. Υποστηρίζονται λογαριασμοί στο domain της, στο Google accounts ή μέσω OpenID. Με αυτόν τον τρόπο γίνεται αναδρομολόγηση του κάθε χρήστη στον χώρο της εφαρμογής που έχει οριστεί για αυτόν καθώς και να είναι ευκολότερος ο διαχωρισμός περιοχών της, για τους διαχειριστές.
- Το URL fetch API χρησιμοποιείτε ώστε να λαμβάνονται πληροφορίες από άλλες ιστοσελίδες μέσω HTTP αιτημάτων. Τα δεδομένα αυτά μπορούμε να τα χρησιμοποιούμε για να παράγουμε δυναμικό περιεχόμενο στην εφαρμογή μας, όπως για παράδειγμα πραγματικού χρόνου πληροφορίες από κοινωνικά δίκτυα.
- Το Blobstore API είναι επίσης μία πολύ σημαντική υπηρεσία. Χρησιμοποιείται για την αποθήκευση δεδομένων πολύ μεγαλύτερων σε μέγεθος από αυτά που μπορεί να διαχειριστεί το datastore. Για παράδειγμα ένα αρχείο μεγάλο σε μέγεθος στέλνετε στο Blobstore όπου και αποθηκεύεται ενώ ένα κλειδί επιστρέφεται για την ανάκτηση του αρχείου.
- XMPP API μέσω του οποίου μπορούν να σταλούν και να ληφθούν άμεσα μηνύματα με άλλες υπηρεσίες που υποστηρίζουν το πρωτόκολλο XMPP.
- Images Processing API το οποίο μια εφαρμογή μπορεί να χρησιμοποιήσει για βασική επεξεργασία φωτογραφιών.

Παρουσίαση χρήσης ορισμένων υπηρεσιών

Datastore

Όπως είδαμε και παραπάνω το datastore αποτελεί την κύρια υπηρεσία για την αποθήκευση μόνιμων δεδομένων της εφαρμογής. Δίνεται ως υπηρεσία ώστε ο προγραμματιστής να μην χρειάζεται να ανησυχεί για το πως τα δεδομένα του θα αναπτύσσονται και θα κλιμακώνονται πάνω στους server του app engine. Η Google αναλαμβάνει να ελέγχει τους server της, να διορθώνει όποια σφάλματα δημιουργούνται και να εγγυάται ότι τα δεδομένα των εφαρμογών θα είναι ασφαλή και σε συνεπή μορφή.

Για να το χρησιμοποιήσουμε δημιουργούμε αντικείμενα στον κώδικα της εφαρμογής, στα οποία γεμίζουμε τις ιδιότητες τους με τις πληροφορίες που θέλουμε και έπειτα καλούμε το api του datastore για να αναλάβει την αποθήκευση τους. Στο παρακάτω παράδειγμα (εικόνα 1) δημιουργούμε ένα αντικείμενο για έναν πελάτη. Συμπληρώνουμε με τα στοιχεία του και έπειτα αποθηκεύουμε την οντότητα στο datastore της εφαρμογής. Τέλος ανακαλούμε το αντικείμενο και εμφανίζουμε τις πληροφορίες του.



εικόνα 1: Αποθήκευση - ανάκτηση από το Datastore

Στην αρχή φορτώνουμε την βιβλιοθήκη random. Από την random επιλέγουμε να φορτωθεί και η choice η οποία θα μας βοηθήσει να επιλέξουμε, παρακάτω στον κώδικα μας, τυχαίες μεταβλητές μέσα από λίστες αλλά και από ένα σύνολο αριθμών. Επίσης από το πακέτο google.appengine.ext φορτώνουμε την db, ή οποία περιέχει όλες τις απαραίτητες συναρτήσεις για την αλληλεπίδραση με το datastore.

Ορίζουμε την συνάρτηση dimiourgia_pelatwn() η οποία θα μας βοηθήσει να φτιάξουμε αυτόματα, δέκα αντικείμενα κλάσης Pelatis και θα τα αποθηκεύσει στο datastore.

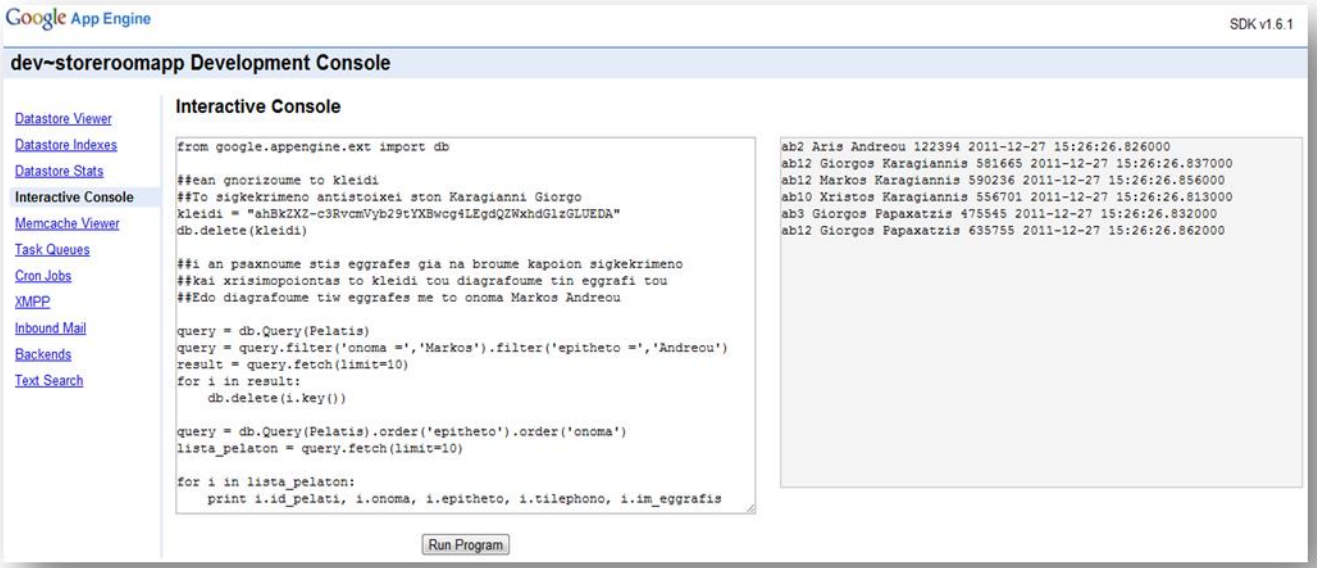
Μέσα στην συνάρτηση έχουμε ορίσει 3 λίστες από τις οποίες η συνάρτηση θα επιλέγει τυχαία ένα στοιχείο. Έχοντας γεμίσει τις ιδιότητες με το πρόθεμα new_ ετοιμάζουμε το αντικείμενο newpelatis που θα είναι τύπου Pelatis. Περνάμε σε κάθε ιδιότητα του Pelatis τα αντίστοιχα, τυχαία επιλεγμένα δεδομένα, και τέλος τρέχουμε την newpelatis.put() που εκτελεί την αποθήκευση στο datastore. Κάτω από την συνάρτηση έχουμε ορίσει και την κλάση που αναπαριστά τον τύπο Pelatis. Μέσα στον κώδικα η κλήση της συνάρτησης γίνεται με το dimiourgia_pelatwn.

Για την ανάκτηση των αποθηκευμένων στο datastore πληροφοριών ετοιμάζουμε ένα ερώτημα (query). Ορίζουμε ότι θέλουμε να εκτελεστεί πάνω στον πίνακα Pelatis και ότι η ταξινόμηση θα γίνει πρώτα σύμφωνα με το επίθετο και έπειτα σύμφωνα με το όνομα. Για τον σκοπό αυτό δημιουργούμε την μεταβλητή query. Τρέχουμε το ερώτημα με το query.fetch(limit=10) και επιστρέφουμε τις τιμές μέσα σε μια καινούργια λίστα την lista_pelaton.

Τέλος με ένα for loop προσπελάζουμε τα δεδομένα της λίστας και εμφανίζουμε το κάθε στοιχείο του πελάτη. Το limit=10 είναι το πλήθος των στοιχείων που θέλουμε να μας επιστρέψει από τον πίνακα. Μιας και μέσα στο for loop της συνάρτησης ορίσαμε ότι θέλουμε να δημιουργήσουμε 10 αντικείμενα, κάτι μεγαλύτερο από δέκα στο limit δεν θα εξυπηρετούσε κάπου.

<u>Διαγραφή αντικειμένων από το datastore</u>

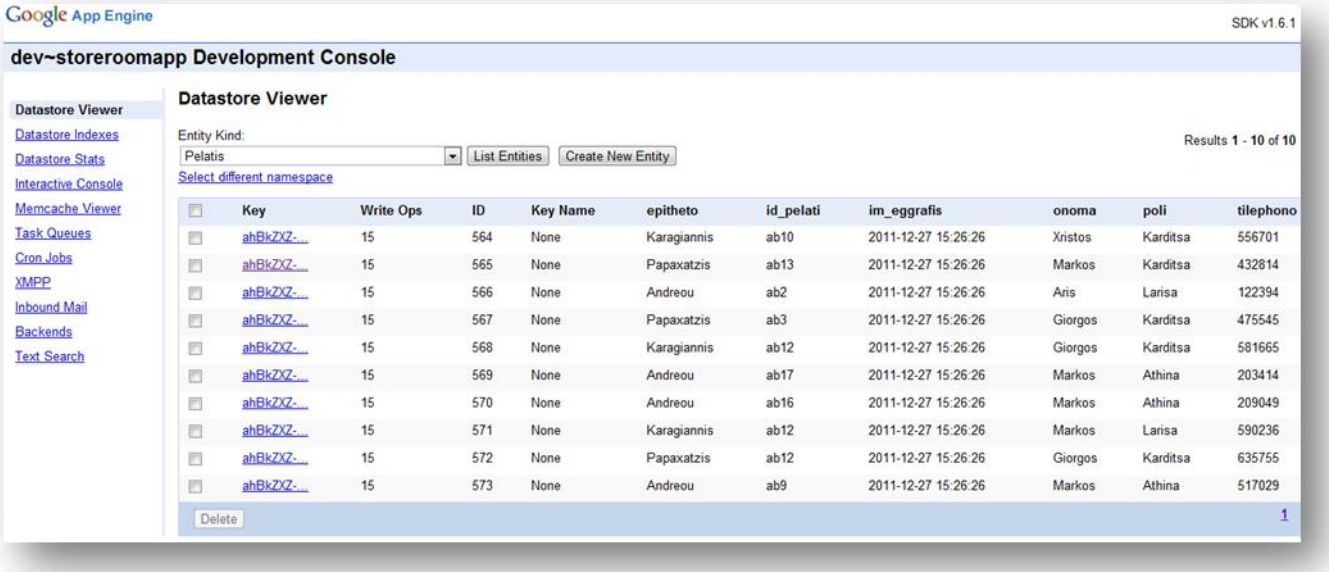
Για να διαγράψουμε μια οντότητα από το datastore μπορούμε είτε γνωρίζοντας το κλειδί που του έχει εκχωρηθεί είτε με το να τρέξουμε ένα ερώτημα με τα κατάλληλα κριτήρια για να βρούμε το κλειδί. Παρακάτω διαγράφουμε δύο αντικείμενα, που δημιουργήσαμε νωρίτερα, και με τους δύο τρόπους (εικόνα 2) :



εικόνα 2: Διαγραφή αντικειμένων από το datastore

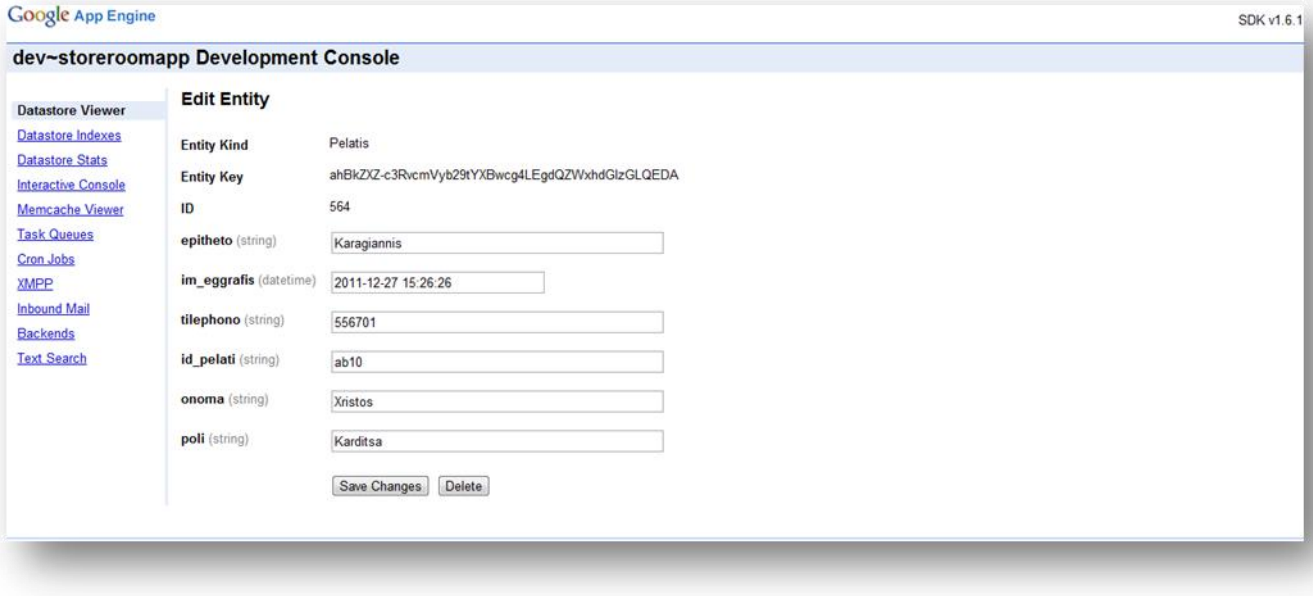
Αν και είχαμε δημιουργήσει 10 αντικείμενα στην αρχή, πλέον έχουν μείνει 6, μιας και είχαμε 3 εγγραφές με το όνομα Μάρκος Ανδρέου που χρησιμοποιήσαμε για κριτήριο διαγραφής.

Μέσα από την διεπαφή πατώντας στο datastore viewer, μπορούμε να δούμε τα αντικείμενα που είναι αποθηκευμένα στο datastore, να δημιουργήσουμε καινούργιους πίνακες, καινούργια αντικείμενα, να τα επεξεργαστούμε καθώς και να τα διαγράψουμε (εικόνα 3)



εικόνα 3: Εμφάνιση δεδομένων μέσα από το sdk

Επίσης το datastore viewer μας εμφανίζει και τα κλειδιά που φτιάχνει αυτόματα το app engine για το κάθε αντικείμενο που αποθηκεύεται στην βάση καθώς και το id του κάθε αντικειμένου που του εκχωρείτε όταν αποθηκεύεται στο datastore. Πατώντας πάνω σε κάποιο από τα αποθηκευμένα αντικείμενα μπορούμε να δούμε όλες τις πληροφορίες του καθώς και να τις επεξεργαστούμε, να τις αλλάξουμε όπως θέλουμε ή και να τις διαγράψουμε (εικόνα 4) .



εικόνα 4: Επεξεργασία εγγραφών μέσα από το sdk

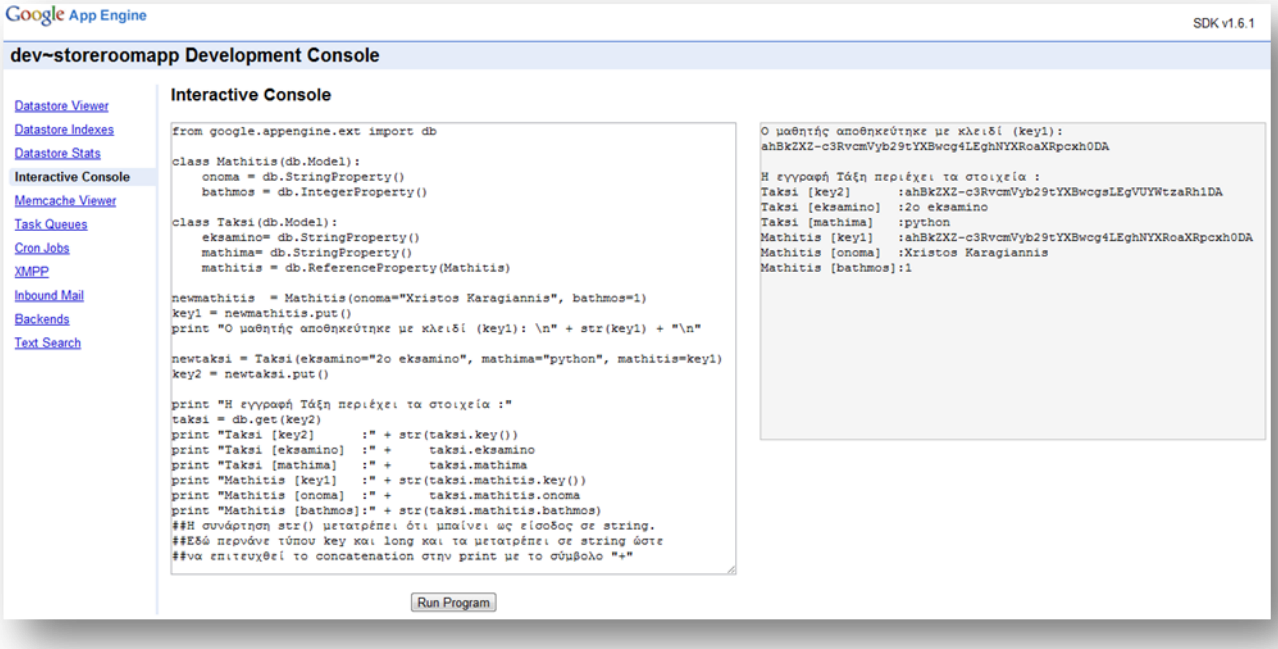
Σύνδεση μεταξύ πινάκων στο datastore

Σε κάθε οντότητα που αποθηκεύουμε μέσα στο datastore, το framework δημιουργεί ένα μοναδικό κλειδί, το πρωτεύων. Εάν θέλαμε να συνδυάσουμε τα στοιχεία ενός πίνακα με τα στοιχεία ενός άλλου, τότε μέσα στον κώδικα δημιουργίας της κλάσης του ενός, ορίζουμε μια ιδιότητα της ώστε να δέχεται το πρωτεύον κλειδί της οντότητας του άλλου πίνακα. Δημιουργεί έτσι ένα ξένο κλειδί, τύπου db.ReferenceProperty στον δεύτερο πίνακα ώστε να αντιστοιχίζεται με τον πρώτο κλειδί της οντότητας που μας ενδιαφέρει.

```
class Mathitis(db.Model):
    onoma = db.StringProperty()
    bathmos = db.FloatProperty()

class Taksi(db.Model):
    eksamino = db.StringProperty()
    mathima = db.StringProperty()
    mathitis = db.ReferenceProperty(Mathitis)
```

(Στην μεταβλητή mathitis του αντικειμένου Taksi, εκχωρείτε το κλειδί του αντικειμένου Mathitis)

Ένα παράδειγμα χρήσης του ReferenceProperty παρουσιάζεται στην παρακάτω εικόνα (εικόνα 5). Δημιουργούμε δύο κλάσεις, την Mathitis και την Taksi. Στην δεύτερη ορίζουμε μια ιδιότητα, την mathitis η οποία είναι τύπου ReferenceProperty. Δημιουργούμε έναν μαθήτή, τον newmathitis, τον αποθηκεύουμε στο datastore ενώ κρατάμε το κλειδί που μας επιστρέφετε στην μεταβλητή key1. Δημιουργούμε μιά τάξη, την newtaksi, και περνάμε στην μεταβλητή mathitis το κλειδί που μας επιστράφηκε νωρίτερα (key1), ενώ κρατάμε το κλειδί όταν αποθηκεύουμε την newtaksi, στην μεταβλητή key2. Τέλος ανακαλούμε την τάξη που μόλις αποθηκεύσαμε χρησιμοποιώντας το κλειδί key2 και περνάμε το αντικείμενο αυτό στην μεταβλητή taksi. Το ReferenceProperty το προσπελάζουμε σαν αντικείμενο μέσα σε αντικείμενο (taksi.mathitis) .

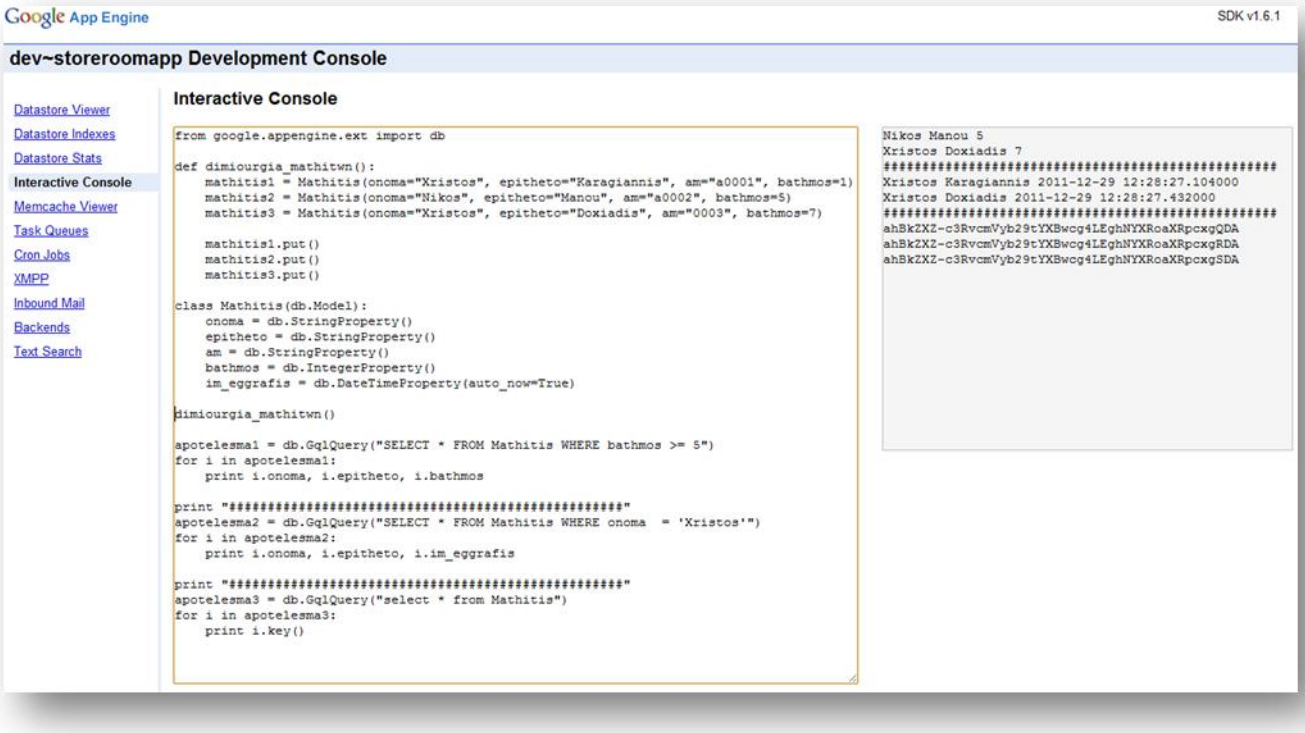


εικόνα 5: Παράδειγμα χρησιμοποιήσεις ξένου κλειδιού

Χρήση του datastore με την GQL

Για την εκτέλεση ερωτημάτων πάνω στους πίνακες της εφαρμογής η Google δημιούργησε επίσης την Gql, η σύνταξη της οποίας μοιάζει με την Sql. Υποστηρίζει μόνο τις δυνατότητες της μηχανής ερωτημάτων του datastore, και για αυτό πολλές από τις λειτουργίες, που μπορεί κάποιος να βρει στην Sql, δεν υπάρχουν. Το σύνολο της σύνταξης της καθώς και ένα παράδειγμα χρήσης (εικόνα 6) της παρουσιάζεται παρακάτω:

```
SELECT [* | __key__]
    [FROM <kind>]]
    [WHERE <condition> [AND <condition> ...]]
    [ORDER BY <property> [ASC | DESC] [, <property> [ASC | DESC] ...]]
    [LIMIT [<offset>,]<count>]
    [OFFSET <offset>]

  <condition> := <property> {< | <= | > | >= | = | != } <value>
  <condition> := <property> IN <list>
  <condition> := ANCESTOR IS <entity or key>
  <list> := (<value> [, <value> ...])
```



εικόνα 6: Παράδειγμα ανάκτησης δεδομένων με Gql

Η Gql παρέχεται και αυτή από την βιβλιοθήκη google.appengine.ext, μέσα από το πακέτο db. Στο παραπάνω παράδειγμα δημιουργούμε, με την βοήθεια της συνάρτησης dimiourgia_mathitwn() τρία αντικείμενα, τύπου Mathitis, τα οποία αποθηκεύουμε στο datastore. Έπειτα εκτελούμε τρία ερωτήματα σε Gql. Το πρώτο μας εμφανίζει ποιοί έχουν βαθμό μεγαλύτερο του 5, το δεύτερο ποιός έχει όνομα Χρήστος και το τελευταίο μας εμφανίζει τα κλειδιά των αντικειμένων μέσα στο datastore.

Memcache

Στα σύγχρονα πληροφοριακά συστήματα απαιτούνται μονάδες αποθήκευσης ώστε τα δεδομένα να παραμένουν σε συνεπή μορφή ακόμα και αν προκύψει κάποιο πρόβλημα υλικού. Το μέσο που επιλέγεται είναι οι σκληροί δίσκοι, όπου μαγνητικοί δίσκοι περιστρέφονται με μεγάλες ταχύτητες και κεφαλές γράφουν τα δεδομένα πάνω σε αυτούς. Η εγγραφή ή η ανάγνωση στοιχείων από τους δίσκους απαιτεί κάποιο χρόνο, τον χρόνο αναζήτησης. Πόσο θα χρειαστεί δηλαδή έως ότου το απαιτούμενο τμήμα του δίσκου να εμφανιστεί κάτω από την κεφαλή. Αν και στους σύγχρονους δίσκους αυτοί οι χρόνοι είναι πολύ μικροί, σε συστήματα όπως το app engine μια τέτοια λειτουργία προσθέτει κόστος σε χρόνο ως προς την συνολική απόκριση της εφαρμογής.

Οι υψηλές επιδόσεων web εφαρμογές χρησιμοποιούν μέσα αποθήκευσης όπως η ram των μηχανών. Επιτυγχάνονται καλύτερες ταχύτητες από αυτές των σκληρών δίσκων. Το μειονέκτημα με αυτές τις υλοποιήσεις είναι ότι δεν αποτελούν λειτουργική επιλογή για αποθήκευση μόνιμων δεδομένων μιας και σε πιθανή διακοπή ρεύματος τα δεδομένα χάνονται μέσα από την Ram. Πολλά από τα δεδομένα που δημιουργούνται από την εφαρμογή δεν χρειάζεται να αποθηκεύονται μόνιμα. Το memcache αναλαμβάνει να τα αποθηκεύει σε προσωρινές μνήμες για κάποια ορισμένα χρονικά διαστήματα. Είναι μια υπηρεσία που προσφέρεται στο app engine και δίνει την δυνατότητα να εκχωρούνται πληροφορίες σε αντικείμενα, που με την σειρά τους αποθηκεύονται, στην μνήμη ram των εξυπηρετητών, και η πρόσβαση σε αυτά γίνεται με το αντίστοιχο κλειδί του αντικειμένου.

Πληροφορίες όπως οι προτιμήσεις των χρηστών για την συνεδρία (session), άμεσα μηνύματα που στέλνουν μεταξύ τους οι χρήστες, δεδομένα που δημιούργησε η εφαρμογή και γενικά πληροφορίες που δεν χρειάζονται μόνιμη αποθήκευση μπορούν να αποθηκευτούν στο memcache. Σχεδιάστηκε ώστε τα ερωτήματα να εκτελούνται ταχύτατα, με μικρούς χρόνους απόκρισης στην αποθήκευση και ανάκτηση τους. Κάποιοι από τους βασικούς κανόνες του memcache παρουσιάζονται παρακάτω:

- Υπάρχει μία μνήμη (cache memory) που διαμοιράζεται ανάμεσα σε όλες τις οντότητες της εφαρμογής. Εάν αποθηκευτεί ή διαγραφεί κάτι από την μνήμη, σε μία οντότητα, τότε οι αλλαγές εφαρμόζονται άμεσα και στις υπόλοιπες.
- Το memory cache λειτουργεί σαν μια μεγάλη μεταβλητή, τύπου dictionary στην Python, όπου κάθε αντικείμενο μέσα στο dictionary έχει ένα και μοναδικό κλειδί

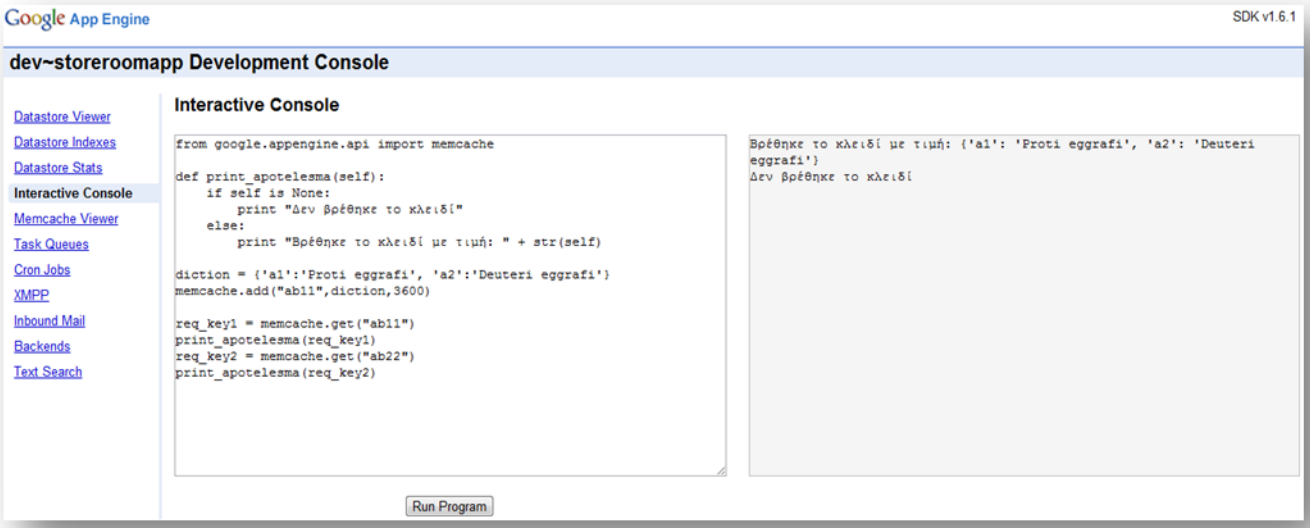
- Κάθε αντικείμενο έχει κάποιο ορισμένο χρόνο ζωής που μπορεί να οριστεί από μερικά δευτερόλεπτα μέχρι και ένα μήνα. Ο εξορισμού χρόνος, αν δεν δηλωθεί διαφορετικά, διαχειρίζεται από την εφαρμογή, η οποία θα το διατηρήσει για όσο περισσότερο μπορεί. Αυτό μπορεί να είναι ανάμεσα σε μερικά δευτερόλεπτα έως και ώρες.
- Τα αντικείμενα που αποθηκεύονται θα πρέπει να μην έχουν μεγάλο μέγεθος .
- Οι πληροφορίες που πρόκειτε να αποθηκευτούν θα πρέπει να έχουν ένα μεγάλο ποσοστό να γίνει ανάκληση τους.

Ένα παράδειγμα χρήσης του memcache παρουσιάζεται παρακάτω:

```
from google.appengine.api import memcache

diction = {'a1':'Proti eggrafi', 'a2':'Deuteri eggrafi'}
memcache.add("ab11",diction,3600)

req_key = memcache.get("ab11")

if req_key is None:
    print "Το κλειδί που ζητήθηκε δεν βρέθηκε"
else:
    print "Το κλειδί βρέθηκε με τιμή: "
    print req_key
```

Στον παραπάνω κώδικα δημιουργούμε ένα dictionary με όνομα diction. Εισάγουμε δύο αντικείμενα το a1,a2 και βάζουμε μία τιμή σε αυτά. Έπειτα καλούμε την memcache.add η οποία αναλαμβάνει να αποθηκεύσει μέσα στο memory cache το diction εκχωρώντας του το κλειδί ab11, για να το διαχωρίσουμε από τα άλλα αντικείμενα, και ορίζοντας ότι θα παραμείνει στην μνήμη για 3600 δευτερόλεπτα. Το αντικείμενο μας αποθηκεύετε και μπορούμε να το ανακτήσουμε με το κλειδί που του ορίσαμε. Αυτό επιτυγχάνεται με το memcache.get περνώντας ως όρισμα το κλειδί του αντικειμένου που μας ενδιαφέρει. Τέλος ελέγχουμε αν η μεταβλητή πάνω στην οποία καλέσαμε την memcache.get είναι κενή ή έγινε επιτυχώς η ανάκτηση.

Δημιουργώ ένα αντικείμενο το ab11 και προσπαθώ να ανακτήσω δύο αντικείμενα εκ των οποίων το ένα δεν υπάρχει. Έπειτα καλώ την συνάρτηση print_apotelesmata για την έξοδο των αποτελεσμάτων όπως φαίνεται στην παρακάτω εικόνα (εικόνα 7) .

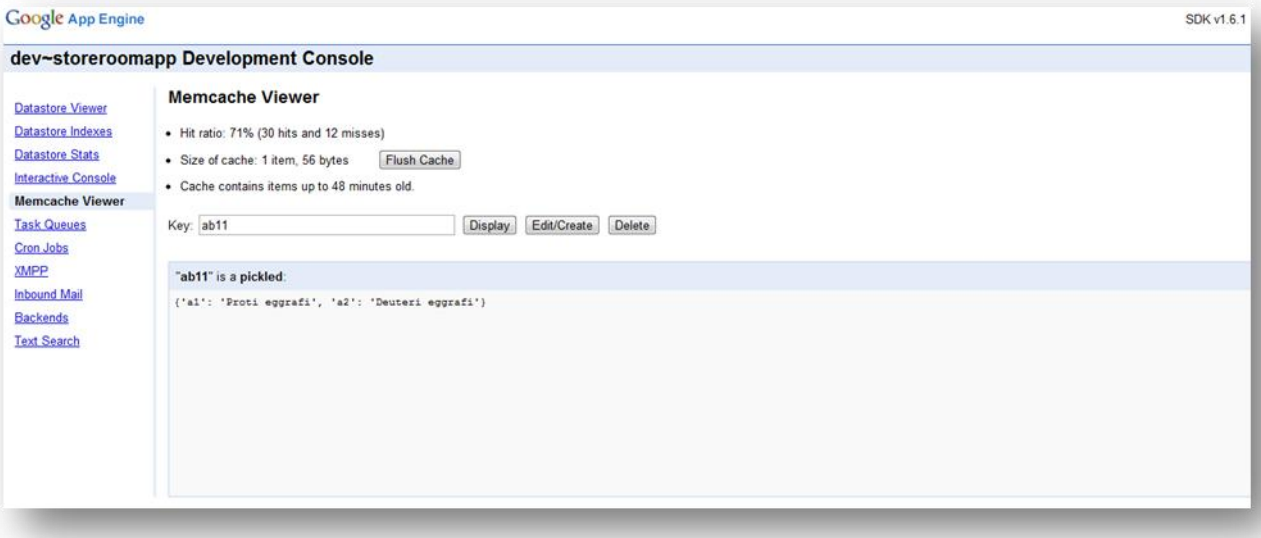


εικόνα 7: Παράδειγμα χρήστης memcache

Επίσης μέσα από το περιβάλλον ανάπτυξης μπορούμε και να δούμε τα αντικείμενα που είναι ήδη αποθηκευμένα μέσα στο memcache, όπως και να δημιουργήσουμε καινούργια ή και να διαγράψουμε όσα δεν θέλουμε, επιλέγοντας το memcache viewer στην διεπαφή.

Το memcache viewer μας εμφανίζει πληροφορίες σχετικά με το ποσοστό επιτυχίας ή αποτυχίας ανάκτησης δεδομένων, πόσα από αυτά υπάρχουν αποθηκευμένα, πόσο χώρο καταλαμβάνουν στην μνήμη καθώς και ο χρόνος που έχει περάσει από την στιγμή που δημιουργήθηκαν. Το κουμπί flush cache αναλαμβάνει να καθαρίσει όλα τα αποθηκευμένα στοιχεία που υπάρχουν στην εφαρμογή μας.

Για να προσπελάσουμε ένα αντικείμενο συμπληρώνουμε το όνομα του στην φόρμα, στο πεδίο key και έπειτα πατάμε display. Με το κουμπί edit/create μπορούμε να επεξεργαστούμε ή και να δημιουργήσουμε ένα καινούργιο αντικείμενο ενώ με το κουμπί delete διαγράφουμε όποια δεν χρειαζόμαστε πλέον. Παρακάτω εμφανίζεται το ab11 που δημιουργήσαμε προηγουμένως (εικόνα 8) :



εικόνα 8: Εμφάνιση αντικειμένων από το memcache μέσα από το sdk

Αποστολή και λήψη email

Το app engine δίνει την δυνατότητα στις εφαρμογές να επικοινωνούν με τους χρήστες μέσω email ώστε να τους ενημερώνουν για αιτήματα που δέχτηκαν από διάφορα κοινωνικά δίκτυα, για να παρακολουθούν κινήσεις στον λογαριασμό τους, για να επιβεβαιώσουν συναλλαγές με την εφαρμογή. Κρατάνε ενήμερο τον διαχειριστή της εφαρμογής για σφάλματα που δημιουργήθηκαν ή για την κατάσταση της εφαρμογής. Εκτός του να στέλνει email μια εφαρμογή μπορεί και να λαμβάνει.

Ένα παράδειγμα αποστολής ηλεκτρονικού μηνύματος εμφανίζεται παρακάτω. Η υπηρεσία είναι διαθέσιμη στην εφαρμογή μας μέσω της βιβλιοθήκης google.appengine.api και συγκεκριμένα του πακέτου mail. Δημιουργούμε ένα αντικείμενο, τύπου mail και χρησιμοποιώντας την EmailMessage της mail, περνάμε τα απαραίτητα στοιχεία για τον αποστολέα (sender), το θέμα του μηνύματος (subject), τον παραλήπτη (minima.to) και το μήνυμα (minima.body) . Τέλος πάνω στο αντικείμενο που δημιουργήσαμε εκτελούμε την send().

```
from google.appengine.api import mail
minima = mail.EmailMessage(sender="test.gr <support@test.gr>",
                           subject="Exete kainourgio minima")
minima.to = "Xristos karagiannis <axrikar@test.gr>"
minima.body = """
Agapite xristo:
Exeis neo minima
```

```
The mail api"""
minima.send()
```

Επίσης εκτός του να στέλνει μπορεί και να λαμβάνει μηνύματα η εφαρμογή. Το email είναι της μορφής string@appid.appspotmail.com . Για να ενεργοποιηθεί η υποστήριξη λήψης email πρέπει μέσα στο app.yaml να συμπληρώσουμε με :

```
inbound_services:
- mail
```

Url fetch api

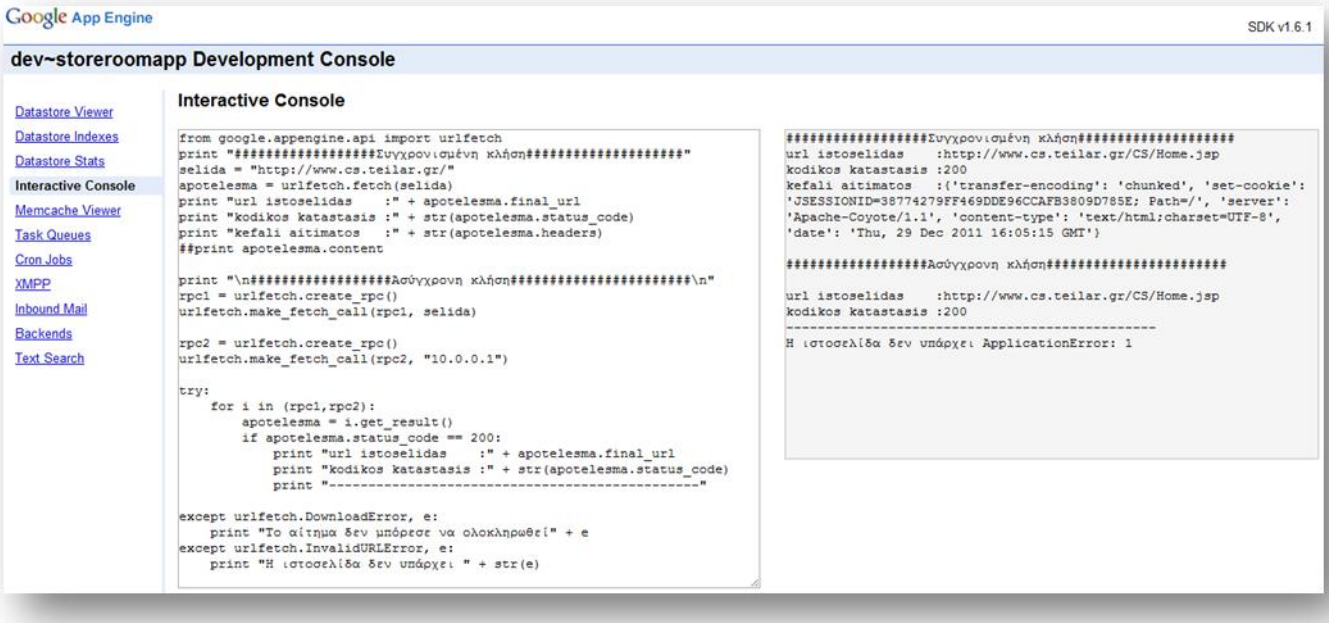
Πολλές από της σύγχρονες εφαρμογές βασίζονται σε πληροφορίες που μπορούν να λαμβάνουν από άλλες ιστοσελίδες μέσα στο internet καθώς και από μηχανήματα, εκτός app engine, που μέσα από διάφορες υπηρεσίες που τρέχουν δημιουργούν δεδομένα. Πληροφορίες κοινωνικής δικτύωσης, μετεωρολογικές προβλέψεις, ειδήσεις, παρακολούθηση λιστών ανακοινώσεων μπορούν να παίξουν τον ρόλο πηγών για την παραγωγή από την εφαρμογή, δυναμικού περιεχομένου, που θα είναι συνεχώς ενημερωμένο.

Η python διαθέτει πολλές βιβλιοθήκες για την εκτέλεση συνδέσεων με απομακρυσμένες μηχανές, λήψη του περιεχομένου που διαθέτουν, εξαγωγή και επεξεργασία αυτών των πληροφοριών ώστε το τελικό προϊόν της διεργασίας να είναι ευανάγνωστο και να μπορεί να ξαναχρησιμοποιηθεί. Η λήψη του περιεχομένου γίνεται συνήθως με τον ίδιο τρόπο που συνδέεται και ένας φυλλομετρητής. Η python δημιουργεί ένα αίτημα προς μία διεύθυνση url, συνήθως προς έναν web server. Ο server με την σειρά του αναλαμβάνει να επεξεργαστεί το αίτημα και να απαντήσει συνήθως σε κώδικα html. Για έναν φυλλομετρητή είναι εύκολο να εμφανίσει με δομημένο τρόπο την html σελίδα. Για την python όμως είναι ένα απλό αρχείο το οποίο το εμφανίζει ακριβώς όπως το έλαβε το οποίο όμως είναι δυσανάγνωστο.

Το app engine δεν δίνει την δυνατότητα στους προγραμματιστές να γράψουν κώδικα σε Python, ο οποίος θα συνδέετε απευθείας πάνω σε τρίτους server για την ανάκτηση πληροφοριών. Σχεδιάστηκε έτσι ώστε να αποτραπούν συνδέσεις οι οποίες θα έστελναν και θα λάμβαναν χωρίς κανένα περιορισμό αρχεία, κακόβουλες επιθέσεις μέσω του app engine σε ιστοσελίδες και server τρίτων, κενά ασφαλείας στις εφαρμογές καθώς και την αυξημένη κίνηση δικτύου λόγω συνεχών αιτημάτων.

Αντί της απευθείας σύνδεσης παρέχει μια υπηρεσία ώστε να εξυπηρετηθούν αυτά τα αιτήματα. Ο προγραμματιστής χρησιμοποιεί το Url fetch api ώστε να λάβει πληροφορίες από άλλες ιστοσελίδες στο διαδίκτυο. Η εφαρμογή ζητάει από το app engine να συνδεθεί σε μια τοποθεσία και αυτό με την σειρά του αναλαμβάνει να διεκπεραιώσει το αίτημα του, να συνδεθεί στην τοποθεσία και να επιστρέψει το περιεχόμενο που του ζητήθηκε. Επίσης μπορεί να δημιουργήσει ένα ασύγχρονο αίτημα (rpc) με την υπηρεσία, ώστε η εφαρμογή να συνεχίσει την εκτέλεση της μέχρι να παραλάβει το αίτημα που έκανε. Παρακάτω ακολουθεί ένα παράδειγμα αυτής της υπηρεσίας (εικόνα 9) :

```
from google.appengine.api import urlfetch

#Σύγχρονο αίτημα
selida = "http://www.cs.teilar.gr/"
apantisi = urlfetch.fetch(selida)
if apantisi.status_code == 200:
  print apantisi.content

#Ασύγχρονο αίτημα
rpc = urlfetch.create_rpc()
urlfetch.make_fetch_call(rpc, selida)
#κώδικας python για εκτέλεση ενώ περιμένει την απάντηση
try:
    apotelesma = rpc.get_result()
    if apotelesma.status_code == 200:
       print apotelesma.content
except urlfetch.DownloadError, e:
    print "Το αίτημα δεν μπόρεσε να ολοκληρωθεί " + str(e)
except urlfetch.InvalidURLError, e:
    print "Η ιστοσελίδα δεν υπάρχει " + str(e)
```



εικόνα 9: Παράδειγμα χρήσης του url fetching

Στο πρώτο σκέλος υλοποιούμε μια σύγχρονη κλήση της υπηρεσίας. Καλούμε την υπηρεσία και μας επιστρέφει το αίτημα που έχουμε κάνει. Ορίζουμε στην μεταβλητή selida το url της τοποθεσίας που θέλουμε, καλούμε την urlfetch.fetch(selida) με όρισμα αυτή την σελίδα, και μας επιστρέφει τον html κώδικα στην μεταβλητή apotelesma. Μέσα στο αντικείμενο apotelesmata έχουμε το url στο apotelesma.final_url (ακόμα και μετα απο redirect), τον κωδικό επιτυχίας στο apotelesma.status_code, την κεφαλή του αιτήματος στο apotelesma.header καθώς και τον κώδικα html στο apotelesma.content.

Στο δεύτερο σκέλος πραγματοποιούμε μια ασύγχρονη κλήση της υπηρεσίας. Φτιάχνουμε δύο αντικείμενα. Το πρώτο rpc1 με την selida που χρησιμοποιήσαμε νωρίτερα και το δεύτερο στην διεύθυνση 10.0.0.1. Έπειτα βάζουμε τον κώδικα μας μέσα σε ένα exception handler (try - except) ώστε να συγκρατήσουμε κάποιο λάθος που μπορεί να γίνει και να εμφανίσουμε ένα μήνυμα πιο φιλικό προς τον χρήστη. Μέσα στον exception handler χρησιμοποιούμε έναν for βρόγχο ώστε το i να πάρει κάθε φορά και τα δύο αντικείμενα που δημιουργήσαμε νωρίτερα.

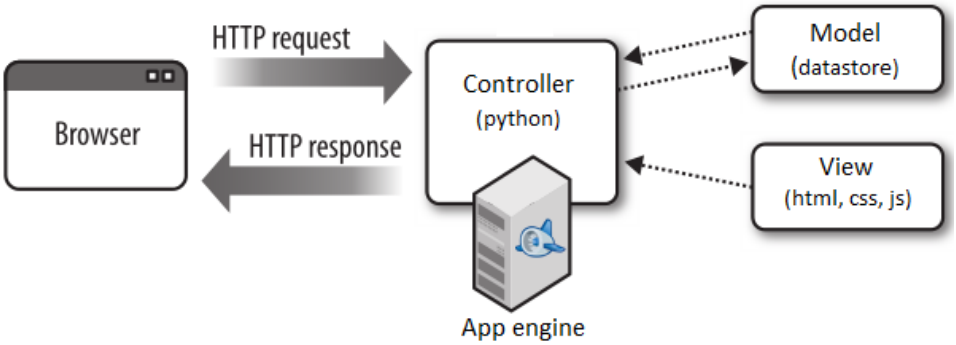
Τέλος ελέγχουμε εάν το status_code είναι ίσο με 200 (επιτυχία) ή όχι. Μιας και το δεύτερο αντικείμενο που δημιουργήσαμε έχει διεύθυνση προορισμού που δεν μπορεί να συνδεθεί εμφανίζει μήνυμα λάθους urlfetch.InvalidURlError οπότε ο exception handler αναλαμβάνει να εμφανίσει το αντίστοιχο μήνυμα που του ορίστηκε. Μέσα στο exception που θα ενεργοποιηθεί ορίσαμε το λάθος να περνάει στην μεταβλητή e ώστε να την χρησιμοποιήσουμε στην εμφάνιση του λάθους για μια ποιο ολοκληρωμένη εμφάνιση του λάθους.

MVC

Το app engine βασίζεται στο πρότυπο δημιουργίας κώδικα, το mvc, που ξεχωρίζει την ανάπτυξη κώδικα σε τρία ξεχωριστά μέρη:

- Το model διαχειρίζεται την συμπεριφορά και τα δεδομένα της εφαρμογής, δέχεται και αποκρίνεται σε αιτήματα για πληροφορίες σχετικά με την κατάσταση στην οποία βρίσκεται όπως επίσης μπορεί να αλλάξει την κατάσταση του ανάλογα με το τι του ζητήθηκε. Στο app engine τον ρόλο του model τον αναλαμβάνει το datastore που αναπαριστά τα δεδομένα της εφαρμογής, τα διαχειρίζεται και τα ανακτά, ενημερώνει ή διαγράφει ανάλογα με το τι του ζητήθηκε να εκτελέσει. Δεν χρειάζεται να γνωρίζει για το πως λειτουργούν τα view και τα controller που αλληλεπιδρούν με αυτό.

- Το view αναλαμβάνει την εμφάνιση της εφαρμογής καθώς και των δεδομένων που παίρνει από το model. Ο κώδικας σε html, css, javascript δημιουργεί την διεπαφή με τον τελικό χρήστη μέσα από περιεχόμενο, δυναμικό ή στατικό, που παράγει η εφαρμογή. Διαμορφώνει κατάλληλα τις πληροφορίες που του στέλνει το model για μια πιο δομημένη εμφάνιση τους
- Το controller δέχεται τα αιτήματα από τον χρήστη, τα διερμηνεύει ώστε να είναι κατάλληλα για εφαρμογή πάνω στο model και έπειτα χρησιμοποιεί το κατάλληλο view ώστε να επιστρέψει στον χρήστη το περιεχόμενο που ζήτησε. Μπορεί να αλληλεπιδράσει με το model και να του αλλάξει την κατάσταση στην οποία βρίσκεται. Στο app engine τον ρόλο αυτό αναλαμβάνει ο κώδικας σε python, που ανάλογα με τις εντολές που έχει μπορεί να διαμορφώσει τα δεδομένα που αποθηκεύονται στο datastore και να επιλέξει για κάθε αίτημα ποιο view (σελίδα html) να χρησιμοποιήσει.



Το mvc είναι ευρέως διαδεδομένο ανάμεσα σε πολλά περιβάλλοντα ανάπτυξης web εφαρμογών. Επιχειρήσεις που αναπτύσσουν κώδικα το χρησιμοποιούν ώστε να διαχωρίζουν σε τομείς την εφαρμογή και να αναλαμβάνει κάθε ομάδα προγραμματιστών έναν τομέα. Επιτυγχάνεται έτσι βελτιστοποίηση του κώδικα που παράγεται, ελαχιστοποίηση του χρόνου υλοποίησης της εφαρμογής και συνεργασία ανάμεσα στις ομάδες ώστε τα μέρη της εφαρμογής να λειτουργούν μεταξύ τους ανάλογα με τις απαιτήσεις του ενός από το άλλο.

Κάθε ομάδα αναλαμβάνει να αναπτύξει τον κώδικα πιο παραγωγικά μιας και οι προγραμματιστές που την απαρτίζουν έχουν πιο στοχευμένες και εξειδικευμένες γνώσεις πάνω στο αντικείμενο που τους ανατίθεται.

Web frameworks

Κατά της διάρκεια ανάπτυξης εφαρμογών οι προγραμματιστές είχαν πολλά προβλήματα που ζητούσαν λύση με κώδικα. Πολλές από τις εφαρμογές έπρεπε να επικοινωνούν με το δικτυακό επίπεδο του εξυπηρετητή, να χρησιμοποιούν το http πρωτόκολλο, να έχουν ένα σταθερό τρόπο για την εμφάνιση των δεδομένων στους τελικούς χρήστες. Για να ξεπεραστούν τα προβλήματα δημιουργήθηκαν τα web application frameworks. Αποτελούν πακέτα τα οποία χρησιμοποιούνται για την ανάπτυξη και επέκταση των εφαρμογών. Είναι έτοιμα κομμάτια κώδικα, σχεδιασμένα για μεγαλύτερη ασφάλεια, ώστε να διατηρούν συνέπεια στα δεδομένα και στη δομή της εφαρμογής, καθώς και να είναι επεκτάσιμα στο μέλλον από τους προγραμματιστές.

Δημιουργήθηκαν ώστε να εξοικονομείτε περισσότερος χρόνος κατά της ανάπτυξη της εφαρμογής αλλά και να είναι εύκολη στη χρήστη από τους τελικούς χρήστες. Χωρίς την χρήση κάποιου framework θα έπρεπε κάθε ιστοσελίδα να γράφετε ολόκληρη σε κώδικα html, css και javascript κάτι που απαιτεί πάρα πολύ χρόνο. Επίσης αν θα έπρεπε να υποστηριχτεί δυναμικό περιεχόμενο τότε ο προγραμματιστής θα ήταν επιφορτισμένος και με το αναπτύξει κώδικα ο οποίος θα συνδεόταν σε κάποια βάση δεδομένων, θα ανακτούσε τα δεδομένα που τον ενδιέφεραν και θα τα εμφάνιζε με δομημένο τρόπο στον τελικό χρήστη. Ένα σημαντικό πρόβλημα είναι και το ότι για οποιαδήποτε αλλαγή θα χρειάζονταν κάποιος με γνώσεις για το πως αναπτύχθηκε η εφαρμογή και με εμπειρία πάνω στον κώδικα που χρησιμοποιούσε η εφαρμογή, ώστε να ενημερώσει τις πληροφορίες της ιστοσελίδας.

Λύση σε όλα αυτά δόθηκε με τα web frameworks, που περιείχαν όλα τα απαραίτητα εργαλεία και μεθόδους αλλά και κώδικα για την απλούστευση των παραπάνω διεργασιών. Μία σύνδεση σε μια βάση δεδομένων επιτυγχάνονταν με την κλήση μιας συνάρτησης του framework που αναλάμβανε να διεκπεραιώσει την σύνδεση, να ανακτήσει τα δεδομένα και να χειριστεί κατάλληλα τα οποιαδήποτε λάθη προέκυπταν. Η χρησιμοποίηση προτύπων για τις ιστοσελίδες μέσω του framework απλοποιούσε την εργασία του προγραμματιστή και δημιουργούσε μια συνέπεια στον τρόπο εμφάνισης της ιστοσελίδας. Το δυναμικό περιεχόμενο εμφανίζονταν εκεί όπου δηλώνονταν, συνήθως με μεταβλητές, μέσα στον στατικό κώδικα html. Επαναχρησιμοποίηση κώδικα επιτυγχάνονταν χωρίς να χρειαστεί να ξαναγραφεί κώδικας από την αρχή.

Το app engine βασίζεται για την εμφάνιση του περιεχομένου της εφαρμογής σε αυτή την λογική. Δίνει την δυνατότητα να χρησιμοποιηθεί ένα πλήθος από web frameworks για την ανάπτυξη της εφαρμογής. Σαν προεπιλεγμένο για την python υποστηρίζει το Django.

Django

Είναι ένα ανοιχτού κώδικα web framework, γραμμένο σε γλώσσα python όπου ακολουθεί το mvc μοντέλο. Αναπτύχθηκε αρχικά από μια εταιρία, την The world company, η οποία δραστηριοποιούνταν στον τομέα της ενημέρωσης. Πήρε το όνομα του από τον κιθαρίστα Django Reinhardt και εμφανίστηκε υπό την BSD άδεια το 2005. Αργότερα, το 2008, το ίδρυμα Django software foundation ανέλαβε την μετέπειτα ανάπτυξη του. Το Django περιλαμβάνει όλα τα απαραίτητα εργαλεία για γρήγορη και ασφαλή ανάπτυξη εφαρμογών.

Μερικά από τα πλεονεκτήματα του Django είναι τα εξής:

- Έχει δικό του authentication σύστημα
- Μπορεί να παράγει rss περιεχόμενο
- Αυτόματη δημιουργία διεπαφής για τον διαχειριστή
- Πολύ δυνατό template σύστημα
- Έτοιμα κομμάτια κώδικα για άμεση χρησιμοποιήσει τους

```
<html>
<body>Καλώς ήρθατε
<h2>10 Πιο πρόσφατα σχόλια χρηστών</h2>

{% for anartisi in periexomena %}
    <br>
    <small>[<i>{{ anartisi.imerominia.ctime }}</i>]</small>
    <b>
    {% if anartisi.onoma %}
        <code>{{ anartisi.onoma }}</code>
    {% else %}
        <i>Ανώνυμος</i>
    {% endif %}
    </b>
    Σχολίασε:
    {{ anartisi.sxolio }}
{% endfor %}

<hr>
</body>
</html>
```

```
import os

from google.appengine.api import memcache, users
from google.appengine.ext import db, webapp
from google.appengine.ext.webapp.template import render
from google.appengine.ext.webapp.util import run_wsgi_app
```

```
class Anartisi(db.Model):
    onoma = db.UserProperty()
    sxolio = db.TextProperty()
    imerominia = db.DateTimeProperty(auto_now_add=True)

anartiseis = Anartisi.all().order('-date').fetch(10)
periexomena = {'anartiseis': anartiseis}
selida = os.path.join(os.path.dirname(__file__), 'index.html')
self.response.out.write(render(selida, periexomena))
```

Στο παραπάνω παράδειγμα δημιουργούμε μία σελίδα html και τον κώδικα που τρέχει στο app engine. Στα τμήματα που θέλουμε να εμφανίσουμε δυναμικό περιεχόμενο χρησιμοποιούμε τον κατάλληλο Django κώδικα. Μέσα στην Python δημιουργούμε ένα μοντέλο, το Anartisti, για τις αναρτήσεις που θέλουμε να αποθηκεύουμε. Ανακτούμε από το datastore τις 10 ποιο πρόσφατες και τις περνάμε σε μια μεταβλητή, την periexomena, τύπου dictionary. Χρησιμοποιούμε το os.path.join για να ενώσουμε το path που παίρνουμε σαν global μεταβλητή από τον server με το index.html, που είναι το view που θα χρησιμοποιήσουμε για την εμφάνιση των αποτελεσμάτων. Τέλος καλούμε την self.response.out.write με την μέθοδο render ώστε να εμφανίσει την σελίδα περνώντας μέσα τα αντικείμενα που ανακτήσαμε.

Στην html κώδικα δημιουργούμε το πρότυπο πάνω στο οποίο θα εμφανίζεται το περιεχόμενο που στέλνει το app engine καθώς και τον Django κώδικα. Για να εμφανίσουμε τις αναρτήσεις που ανακτήσαμε χρησιμοποιούμε ένα for βρόγχο που διαβάζει το κάθε ένα αντικείμενο μέσα στην μεταβλητή periexomena και τα περνάει στο anartisi. Δηλώνουμε την έναρξη του με το {% for anartisi in periexomena %}. Εμφανίζουμε την ημερομηνία και έπειτα ελέγχουμε αν υπάρχει όνομα μέσα στο κάθε αντικείμενο και το εμφανίζουμε με το {{ anartisi.onoma}}, αλλιώς εμφανίζουμε ανώνυμος. Για κάθε μεταβλητή που θέλουμε να εμφανίσουμε την βάζουμε σε διπλές αγκύλες. Τέλος για να σηματοδοτήσουμε τον τερματισμό του Django κώδικα βάζουμε {% endfor %} .

Μια από τις σημαντικές δυνατότητες του Django είναι η επαναχρησιμοποίηση κώδικα, καθώς η χρησιμοποίηση μιας σελίδας ως βάση, που χρησιμοποιούν όλες οι υπόλοιπες για την εμφάνιση του περιεχομένου τους. Μπορούμε να χρησιμοποιήσουμε μπλοκς που ορίζουν το μέρος της σελίδας που μπορεί να προστεθεί δυναμικό περιεχόμενο. Δηλώνουμε με {% block onoma %} {% endblock %} το τμήμα αυτό που μπορούν, χρησιμοποιήσουν οι υπόλοιπες σελίδες που βασίζονται στην αρχική, ώστε να προσθέσουν

κώδικα html, css και javascript. Επίσης μπορούμε μέσα σε αυτά τα block να εμφανίσουμε περιεχόμενο του datastore ή του app engine.

Σε κάθε μια σελίδα που θέλουμε να χρησιμοποιεί σαν βάση κάποια άλλη, δηλώνουμε στην αρχή του κώδικα της το {% extends basi.htm %} ώστε να αντιλαμβάνεται το Django ποια σελίδα θα χρησιμοποιήσουμε ώς πρότυπο. Σημαντικό πλεονέκτημα μιας και η διατήρηση της δομής της εφαρμογής, και της εμφάνισης της διεπαφής παίζει σημαντικό ρόλο στην χρησιμότητα και λειτουργικότητα της εφαρμογής.

Μέρος ΙΙ

Για την πτυχιακή μου επέλεξα να αναπτύξω μια εφαρμογή για την διαχείριση αποθήκης. Η εφαρμογή θα αναλαμβάνει την αποθήκευση και επεξεργασία πελατολογίου, λίστας προϊόντων καθώς και παραγγελιών για την αποθήκη του καταστήματος. Ο κώδικας της, είναι σε Python στην έκδοση 2.5. Τα δεδομένα θα αποθηκεύονται μέσα στο datastore ενώ την δυναμική εμφάνιση τους αναλαμβάνει το Django.

Ορισμός του αρχείου app.yaml

Το app engine δέχεται αιτήματα και διαβάζει πρώτα το αρχείο app.yaml για να αντιστοιχήσει το κάθε αίτημα, με το κατάλληλο script σε python για την διεκπεραίωση του. Το app.yaml της εφαρμογής μου παρουσιάζεται παρακάτω (κώδικας 1):

```
application: storeroomapp
version: 1
runtime: python
api_version: 1

handlers:
- url: /static
  static_dir: static

- url: /favicon\.ico
  static_files: static/images/favicon.ico
  upload: static/images/favicon\.ico

- url: .*
  script: main.py
```

κώδικας 1 : app.yaml εφαρμογής

Αρχικά ορίζουμε το όνομα της εφαρμογής μας μέσω του οποίου το app engine θα αντιστοίχηση και ένα url. Το url αυτό θα είναι της μορφής http://storeroom.appspot.com. Δηλώνουμε τον αριθμό της έκδοσης του κώδικα της εφαρμογής μας. Μπορούμε σε κάθε αναβάθμιση στον κώδικα που κάνουμε να ορίζουμε και διαφορετικό αριθμό ώστε να μπορούμε μέσα από την σελίδα διαχείρησης στο app engine να επιλέγουμε κάθε φορά ποιά έκδοση θα είναι διαθέσιμη στους χρήστες. Έπειτα ορίζουμε την γλώσσα την οποία επιλέξαμε για να αναπτύξουμε την εφαρμογή μας. Στην δική μου εφαρμογή έχω επιλέξει την Python. Τέλος ορίζουμε το api_version της γλώσσας που επιλέξαμε.

Στο επόμενο τμήμα δηλώνουμε τα handlers τα οποία θα χρησιμοποιεί το app engine για να διεκπεραιώνει τα αιτήματα των χρηστών. Αρχικά δηλώνουμε τον φάκελο στον οποίο

θα φιλοξενείτε το στατικό περιεχόμενο της εφαρμογής μας. Περιεχόμενο που θα αποστέλετε γρηγορότερα και χωρίς να χρειάζεται κάποια επεξεργασία από το app engine. Μέσα σε αυτό τον φάκελο κρατάμε τον κώδικα του css για την εμφάνιση της εφαρμογής μας, τον javascript κώδικα, φωτογραφίες και εικονίδια καθώς και τα templates που θα χρησιμοποιηθούν ως βάση για την εμφάνιση της εφαρμογής μας.

Κάθε ένα από αυτά τα στατικά αρχεία μπορούμε να τα χωρίσουμε σε ξεχωριστούς φακέλους, ώστε η διαχείριση τους και η προσθήκη επιπλέον περιεχομένου, να είναι ευκολότερη. Μέσα στον φάκελο templates μπορούμε να αποθηκεύσουμε εκτός από στατικό περιεχόμενο από html σελίδες, και τις σελίδες που θα περιέχουν κώδικα Django.

Μπορούμε εδώ να ορίσουμε και το εικονίδιο που θα εμφανίζεται στον browser μας όταν ζητάει να εμφανίσει μια σελίδα της εφαρμογής μας. Το favicon.ico το αποθηκεύουμε σε ένα φάκελο μέσα στο static περιεχόμενο, και δηλώνουμε την πλήρη διαδρομή για την θέση που βρίσκεται.

Τέλος ορίζουμε το script που θα αναλαμβάνει την επεξεργασία των αιτημάτων μας. Εδώ μπορούμε να χωρίσουμε σε τμήματα την εφαρμογή μας και ανάλογα με το τι αίτημα δέχεται από τον browser του χρήστη να γίνεται και δρομολόγηση στο κατάλληλο script. Για την δική μου εφαρμογή όλα τα αιτήματα τα αναλαμβάνει το main.py.

Βασικές τάξεις και διεργασίες για την εφαρμογή

Η ρίζα της εφαρμογής μας όπου σηματοδοτεί και την έναρξη της οντότητας πάνω στο app engine είναι ο παρακάτω κώδικας (κώδικας 2) :

```
if __name__ == '__main__':
    main()
```

κώδικας 2: Εκκίνηση εφαρμογής

Είναι μια συνθήκη η οποία είναι πάντα αληθής όταν η εφαρμογή μας, τρέχει πάνω στους server του app engine. Αφού η συνθήκη επιτυγχάνει πάντα, μπαίνει και καλεί την μέθοδο main. Η δήλωση της παρουσιάζεται παρακάτω (κώδικας 3):

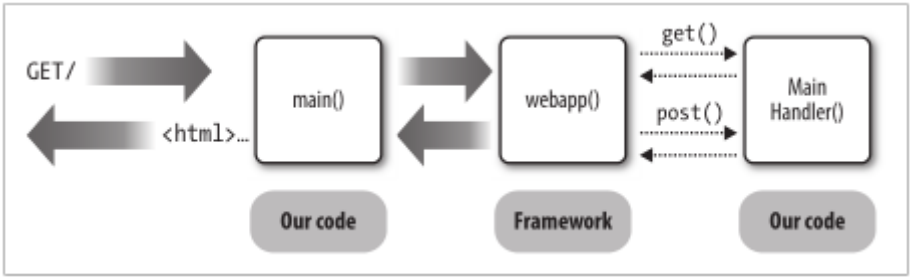
```
def main():
    application = webapp.WSGIApplication([
        ('/addClient', addClient),
        ('/showClient', showClient),
        ('/editClient', editClient),
        ('/searchClient', searchClient),
```

```
        ('/addItem', addItem),
        ('/showItem', showItem),
        ('/delItem', delItem),
        ('/editItem', editItem),
        ('/searchItem', searchItem),
        ('/addOrder', addOrder),
        ('/showOrder', showOrder),
        ('/searchOrder', searchOrder),
        ('/delOrder', delOrder),
        ('/delClient', delClient),
        ('/infoClient', infoClient),
        ('/infoItem', infoItem),
        ('/infoOrder', infoOrder),
        ('/GENERATEALL', GENERATEALL),
        ('/DELETECLIENT', DELETECLIENT),
        ('/DELETEORDER', DELETEORDER),
        ('/DELETEITEM', DELETEITEM),
        ('/DELETEBASI', DELETEBASI),
        ('/DELETEALL', DELETEALL),
        ('/.*', MainHandler)], debug=True)
    wsgiref.handlers.CGIHandler().run(application)
```

κώδικας 3: Handlers εφαρμογής

Η μέθοδος main()

Μία συνήθη τακτική στον αντικειμενοστραφή προγραμματισμό είναι η παραχώρηση του για τον χειρισμό μιας εισόδου σε ένα framework. Έτσι επιτρέπουμε στο framework να χρησιμοποιήσει τους δικούς μας χειριστές όταν αυτό είναι απαραίτητο. Στην αρχή δηλώνουμε σε ποιές περιπτώσεις θέλουμε να αλληλεπιδράσουμε με το framework, ώστε ο κώδικας μας να αναλάβει κάποια από την επεξεργασία των δεδομένων. Για παράδειγμα στην λήψη αιτήματος GET από την main μας, ορίζουμε σε ποιές περιπτώσεις, δηλαδή σε ποιά ζητούμενα URl θα θέλαμε να καλεί το framework, τον κώδικα μας. Το framework αυτό στο app engine είναι το webapp. Λαμβάνει τα αιτήματα, διαβάζει όλα τα δεδομένα μέσα στο αίτημα και αν χρειαστεί τα μετατρέπει σε είσοδο για τον κώδικα μας.



Στον προηγούμενο κώδικα δήλωσης της main ορίζουμε για κάθε ένα αίτημα που δέχεται η εφαρμογή σε ποιόν χειριστή - handler να το αναθέσει. Μέσα στις παρενθέσεις στο

πρώτο όρισμα που παίρνουν, μπαίνει η σελίδα που ζητήθηκε και έπειτα ποιός χειριστής στον κώδικα μας είναι κατάλληλα διαμορφωμένος ώστε να το αναλάβει, να το επεξεργαστεί και τέλος να στείλει τις πληροφορίες που ζητήθηκαν. Για οποιαδήποτε ερωτήματα, τα οποία δεν έχει οριστεί κάποιος χειριστής, τα αναλαμβάνει ο MainHandler. Επίσης η μεταβλητή, debug, μπορεί να βρεθεί σε δύο καταστάσεις. Είτε False που σημαίνει απενεργοποίηση της αποσφαλμάτωσης είτε true που σημαίνει ότι, όταν η εφαρμογή μας θα τρέχει στο development περιβάλλον και όχι στο app engine, τότε θα εμφανίζονται πληροφορίες για τα λάθη που δημιουργήθηκαν ώστε να γίνει αποσφαλμάτωση. Πάνω στο app engine ακόμα και αν έχει οριστεί ώς true, δεν εμφανίζονται λάθη αποσφαλμάτωσης στον τελικό χρήστη.

Ο MainHandler αναλαμβάνει τα αιτήματα στα οποία δεν αντιστοιχεί κάποιος χειριστής. Στην εφαρμογή μου, παίρνει αυτά τα ερωτήματα και απλά αναδρομολογεί στην αρχική σελίδα της εφαρμογής. Ο κώδικας του φαίνεται παρακάτω (κώδικας 4):

```
class MainHandler(webapp.RequestHandler):

    def get(self):
        if emfanise(self,self.request.path):
            return
        emfanise(self,'index.htm')
```

κώδικας 4: MainHandler

Σε ερωτήματα τύπου GET χρησιμοποιεί την emfanise για να βρει, αν υπάρχει το ζητούμενο πρότυπο σε html, το template, αλλιώς επιστρέφει την index.htm.

Τέλος από τα βασικότερα κομμάτια του κώδικα της εφαρμογής μου είναι η συνάρτηση emfanise. Αναλαμβάνει κάθε φορά που καλείτε, να εμφανίσει την σελίδα που της ζητείτε μέσα από τα πρότυπα των template, συνενώνοντας τα με τις dictionary μεταβλητές που θα χρησιμοποιεί το Django για την δημιουργία του δυναμικού κώδικα (κώδικας 5):

```
def emfanise(handler, selida = 'index.htm', times = {}) :
    server_path = os.path.join(
        os.path.dirname(__file__),
        'templates/' + selida)
    if not os.path.isfile(server_path):
        return False

    tel_times = dict(times)
    tel_times['path'] = handler.request.path
    emfanisi = template.render(server_path,tel_times)
    handler.response.out.write(emfanisi)
    return True
```

κώδικας 5 : Συνάρτηση emfanise

Ξεκινάει παίρνοντας ως όρισμα τρείς μεταβλητές. Τον Handler, την σελίδα προς εμφάνιση και ένα κενό dictionary. Μπορούμε κάθε μία από αυτές τις μεταβλητές να τις υπερφορτώνουμε κάθε φορά ανάλογα με το περιεχόμενο που θέλουμε να δημιουργήσουμε. Έτσι στο selida κάθε φορά περνάμε την σελίδα από τα templates, που θα χρησιμοποιήσουμε για την εμφάνιση στον browser του περιεχομένου που ζητήθηκε. Επίσης στην μεταβλητή times βάζουμε της μεταβλητές τύπου dictionary που δημιουργούμε με τον κώδικα μας για το δυναμικό περιεχόμενο.

Η μεταβλητή server_path συνενώνει την διαδρομή που παίρνει από τον server που έχει οριστεί στις global μεταβλητές του runtime environment (os.path.dirname(__file__)), με τον static φάκελο μας που περιέχει τα πρότυπα html, τον templates, και την ζητούμενη σελίδα από το selida. Ελέγχει αν αυτή η διαδρομή μπορεί να βρεθεί στον εξυπηρετητή. Εάν δεν βρεθεί επιστρέφει false στον κώδικα που την κάλεσε.

Εάν όντως υπάρχει αυτή η διαδρομή βάζει μέσα στην μεταβλητή tel_times το dictionary times, και προσθέτει μια ακόμη μεταβλητή σε αυτό, την path. Καλεί την μέθοδο render από το πακέτο template πάνω στα αντικείμενα, server_path και tel_times ώστε να δημιουργήσει ένα αντικείμενο το emfanisi κατάλληλο για το framework του app engine. Τέλος χρησιμοποιεί την handler.response.out.write(emfanisi) για τη αποστολή του αιτήματος στον χρήστη και επιστρέφει true για να σηματοδοτήσει την επιτυχή εκτέλεση της.

Το πρότυπο template

Για την εμφάνιση της εφαρμογής μου χρησιμοποιώ μια σελίδα html, όπου περιέχει κώδικα django πάνω στον οποίο θα δημιουργείτε το δυναμικό περιεχόμενο, ή θα χρησιμοποιείται ως βάση για την εμφάνιση της κάθε σελίδας, της εφαρμογής μου. Το μοντέλο αυτό επιτρέπει την γρήγορη ανάπτυξη της εφαρμογής, μιας και γίνεται επαναχρησιμοποιήσει του ήδη υπάρχοντος κώδικα, για την δημιουργία νέων σελίδων.

Η σελίδα base.htm είναι η βάση την οποία χρησιμοποιούν οι υπόλοιπες σελίδες, για να εμφανίσουν το περιεχόμενο τους. Με τον τρόπο αυτό επιτυγχάνουμε κυρίως την συνέπεια στην εμφάνιση της εφαρμογής μας, στον χρήστη, ανεξάρτητα ποιά σελίδα ζητάει κάθε φορά. Επίσης μπορούμε ανάλογα με τις ανάγκες τις κάθε σελίδας να φορτώνουμε και στον broswer δυναμικά, επιπλέον κώδικα html, css και javascript για να καλύπτουμε κάθε φορά τις ανάγκες

των σελίδων της εφαρμογής μας. Η base.htm περιέχει την βασική δομή της html καθώς και του css κώδικα που μοιράζονται όλες οι υπόλοιπες σελίδες για την εμφάνιση τους.

Για την επέκταση των λειτουργιών, χρησιμοποιούμε κώδικα Django που παρέχει την δυνατότητα, στα μέρη της βασικής σελίδας, που ορίζουμε με αυτόν να χρησιμοποιηθεί και επιπλέον κώδικας. Κομμάτια Django έχουν χρησιμοποιηθεί για την δυναμική φόρτωση κώδικα css και javascript ώστε να εξυπηρετηθεί η εφαρμογή και οι δυνατότητες της.

Επίσης κάθε σελίδα της εφαρμογής μας ανακαλούν την βασική, ώστε με βάση αυτή να εμφανίσουν το περιεχόμενο τους.

Στον παρακάτω κώδικα εμφανίζεται ένα κομμάτι από την base.htm (κώδικας 6), όπου υπάρχει κώδικας Django:

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>{% block title %}{% endblock %}</title>

<!-- CSS -->
  <link rel="stylesheet" href="../static/css/style.css" type="text/css"
        media="screen" />
{% block addcss %}{% endblock %}
<!-- JS -->
{% block addjs %}{% endblock %}
  <script type="text/javascript" src="../static/js/jquery-1.6.4.min.js"></script>
----------------------παρεμβάλεται κώδικας html---------------------------------
<!-- MAIN -->
<div id="main">
  <div class="wrapper">
   <div id="headline">
      <span class="main">
        {% block kefalida %}{% endblock %}
      </span>
      <span class="sub">
        {% block ipokefalida %}{% endblock %}
      </span>
    </div>
    {% block message %}{% endblock %}
    {% block error %}{% endblock %}
    <div id="content">
      <p>
        {% block bodycontent %}{% endblock %}
      </p>
    </div>
  </div>
</div <!-- ENDS MAIN -->
```

κώδικας 6: Τμήμα από την base.htm

Στα παραπάνω κομμάτια που ορίζονται μέσα σε αγκύλες, και ακολουθεί η λέξη block το Django αναλαμβάνει να εμφανίσει περιεχόμενο όταν του ζητηθεί. Έτσι μπορούμε να

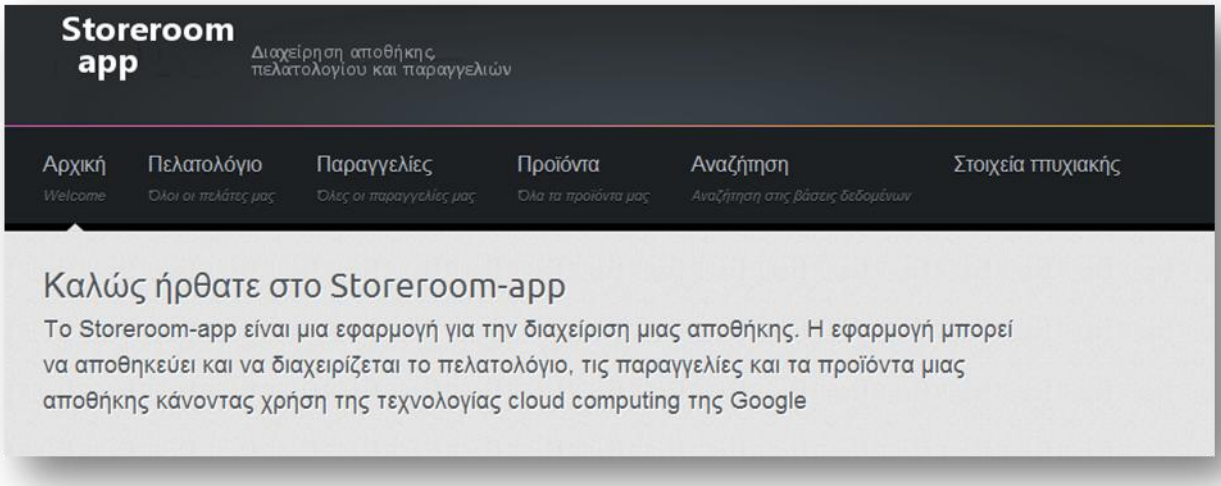
έχουμε ένα τίτλο που αλλάζει ανάλογα με την σελίδα της εφαρμογής μας, κώδικα css και javascript να φορτώνεται όταν ζητείται, καθώς και κεφαλίδες και υποκεφαλίδες στις διάφορες σελίδες μας. Επίσης τα block message και error χρησιμοποιούνται από τους χειριστές του κώδικα Python για να εμφανίζουν μηνύματα επιτυχίας ή αποτυχίας αντίστοιχα. Τέλος το κυρίως τμήμα των σελίδων μας εμφανίζεται μέσα στο block bodycontent. Ένα παράδειγμα χρησιμοποιήσεις της παραπάνω βάσης από μια άλλη σελίδα (index.htm) εμφανίζεται παρακάτω (κώδικας 7):

```
{% extends "base.htm" %}
{% block title %}Storeroom-app Αρχική{% endblock %}
{% block kefalida %}Καλώς ήρθατε στο Storeroom-app{% endblock %}
{% block ipokefalida %}Το Storeroom-app είναι μια εφαρμογής...{% endblock %}
{% block message %} <p style="color: green;"> {{message}} {% endblock %}
{% block error %} <p style="color: red;"> {{error}} </p> {% endblock %}
{% block bodycontent %}
---------------------------κώδικας html της σελίδας----------------------------
{% endblock %}
```

κώδικας 7: Τμήμα από την index.htm

Αρχικά με την χρησιμοποιήσει του extends δηλώνει ότι θέλει να χρησιμοποιήσει κάποια άλλη σελίδα ως βάση. Για κάθε ένα από επόμενα block το Django αντικαθιστά στην σελίδα βάσης τον κώδικα που υπάρχει σε αυτά. Στο συγκεκριμένο παράδειγμα έχουμε αντικαταστήσει τον τίτλο της σελίδας, προσθέσαμε κώδικα javascript, κεφαλίδα και χρησιμοποιήσαμε το block bodycontent για την εμφάνιση του κυρίως τμήματος της index.htm.

Η index.htm είναι και η αρχική σελίδα της εφαρμογής που εμφανίζεται στον χρήστη. Επίσης χρησιμοποιείτε όταν δηλωθεί κάποιο url από τον browser που δεν υπάρχει. Καλείτε πάντα από τον MainHandler(), που παρουσιάστηκε παραπάνω κώδικας 4. Ο χειριστής καλεί την μέθοδο emfanise περνώντας ως ορίσματα την ζητούμενη σελίδα. Η emfanise ελέγχει μέσα στο file system για το αν υπάρχει η συγκεκριμένη σελίδα. Αφού την βρίσκει εκτελεί με την βοήθεια της render και τον δυναμικό κώδικα του Django και τέλος εμφανίζει στον χρήστη την σελίδα όπως φαίνεται παρακάτω (εικόνα 10)



εικόνα 10: εμφάνιση της index.htm

Δημιουργία μοντέλων datastore

Τέσσερις πίνακες αναλαμβάνουν να αποθηκεύουν τα δεδομένα της αποθήκης (κώδικας 8). Ο πίνακας Client αναλαμβάνει να κρατάει τις πληροφορίες για το πελατολόγιο, ο Item για τα προϊόντα και ο Order για τις παραγγελίες που γίνονται. Στον πίνακα Basi θα αποθηκεύονται τα κλειδιά του κάθε αντικειμένου όταν δημιουργείται η παραγγελία. Επέλεξα να μην χρησιμοποιήσω ξένα κλειδιά στις παραγγελίες αλλά να συγκρατώ πλεοναστικά τα κλειδιά στον πίνακα Basi. Έτσι αν γίνει κάποια αλλαγή στα στοιχεία των προϊόντων ή των πελατών, ο πίνακας παραγγελιών θα κρατάει ιστορικό από τις παραγγελίες με τα αρχικά δηλωμένα στοιχεία της παραγγελίας.

```
##Πίνακας datastore για τον πελάτη
class Client(db.Model):
    onoma = db.StringProperty(required=True)
    epitheto = db.StringProperty(required=True)
    tilephono = db.StringProperty(required=True)
    dieuthinsi = db.StringProperty(required=True)
    poli = db.StringProperty(required=True)
    email = db.StringProperty(required=False)

##Πίνακας datastore για τα αντικείμενα
class Item(db.Model):
    marka = db.StringProperty(required=True)
    modelo = db.StringProperty(required=True)
    tipos = db.StringProperty(required=True)
    timi = db.FloatProperty(required=True)
    temaxia = db.IntegerProperty(required=True)
    perigrafi = db.StringProperty(required=False, multiline = True)
```

```
##Πίνακας datastore για τις παραγγελίες
class Order(db.Model):
    id_paraggelia = db.StringProperty()
    pelatis = db.StringProperty(required=True)
    proion = db.StringProperty(required=True)
    posotita = db.IntegerProperty(required=True)
    timi = db.FloatProperty()
    fpa = db.IntegerProperty(choices=set([6,9,12,16,20,23,25]))
    sinolo = db.FloatProperty()
    ekptosi = db.IntegerProperty()
    sinoloekpt = db.FloatProperty()
    created = db.DateTimeProperty(auto_now_add = True)

##Πίνακας datastore για την basi
class Basi(db.Model):
    id_paraggelia = db.StringProperty()
    id_pelatis = db.StringProperty()
    id_item = db.StringProperty()
    created = db.DateTimeProperty(auto_now_add = True)
```

κώδικας 8: Ορισμός των πινάκων στο datastore

Στην πρώτη τάξη δημιουργώ ένα μοντέλο για τους πελάτες. Περιέχει, ως ιδιότητες, το όνομα, επίθετο, τηλέφωνο, διεύθυνση, πόλη και το email των πελατών. Έπειτα δημιουργώ το μοντέλο για τα προϊόντα με ιδιότητες την μάρκα, μοντέλο, τύπο, τιμή , διαθέσιμα τεμάχια, και μια περιγραφή του προϊόντος. Ο πίνακας Order κρατάει το id_paraggelia που είναι ένας μοναδικός κωδικός για την κάθε μία παραγγελία, το ονοματεπώνυμο του πελάτη, την μάρκα και μοντέλο του προϊόντος, την ποσότητα του, την τιμή , το ΦΠΑ, το σύνολο χωρίς έκπτωση, την έκπτωση του πελάτη, το συνολικό ποσό μαζί με την έκπτωση και την ημερομηνία και ώρα δημιουργίας και αποθήκευσης του αντικειμένου. Τέλος μέσα στον πίνακα Basi αποθηκεύουμε το id_paraggelias που δημιουργήσαμε νωρίτερα, τα κλειδιά των αντικειμένων πελάτη και προϊόντος μέσα στο datastore καθώς και την ημερομηνία και ώρα καταχώρησης.

Αποθήκευση και ανάκτηση μοντέλων πελατών

Η λειτουργία της εφαρμογής βασίζεται στην κλασσική μέθοδο αλληλεπίδρασης τελικού χρήστη με εξυπηρετητή. Ο χρήστης κάνει ένα αίτημα προς το app engine και αυτό με την σειρά του, του στέλνει τα απαραίτητα αρχεία (html, css, js) και τον κώδικα που δημιουργήθηκε δυναμικά για να συνθέσει ο browser την διεπαφή. Για την αποθήκευση νέων εγγραφών από τους χρήστες, χρησιμοποιούνται φόρμες όπου συμπληρώνονται τα απαραίτητα στοιχεία των πελατών, προϊόντων και παραγγελιών. Τα στοιχεία αυτά αποστέλλονται από τον φυλλομετρητή πίσω στο app engine όπου ο κώδικας για την κάθε λειτουργία αναλαμβάνει να

τα επεξεργαστεί και να τα αποθηκεύσει στο datastore. Η μέθοδος που χρησιμοποιεί η φόρμα για να αποστείλει τα δεδομένα είναι POST.

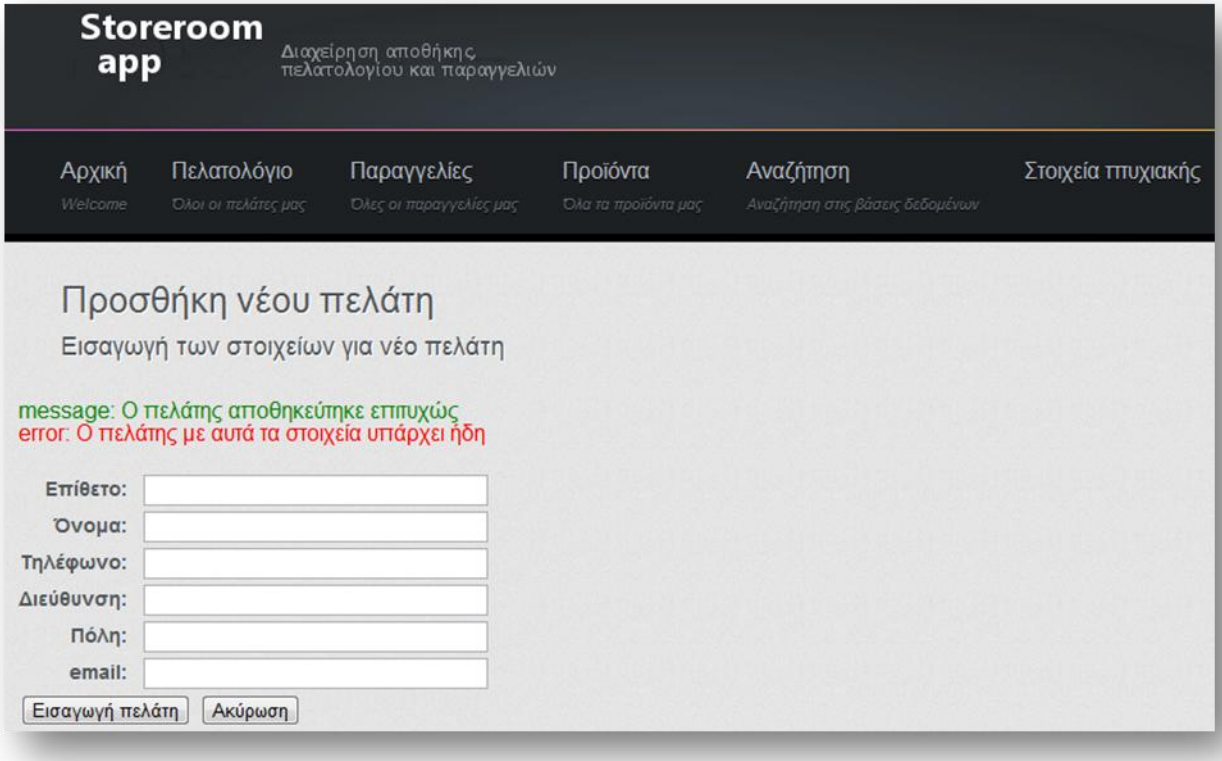
Το app engine με την παραλαβή των δεδομένων περνάει στις μεταβλητές της python το κάθε πεδίο της φόρμας και έπειτα επικυρώνει αυτές τις τιμές ώστε να μην υπάρχουν λάθη ή διπλότυπα. Παρακάτω παρουσιάζεται ο κώδικας (κώδικας 9) που αναλαμβάνει αυτή την διεργασία:

```
class addClient(webapp.RequestHandler):
    def get(self,error='',message=''):
        emfanise(self,'newclient.htm',{'message':message, 'error':error})

    def post(self):
        try:
            ##Eisagogi dedomenon apo tin forma html
            name = self.request.get('name')
            surname = self.request.get('surname')
            phone = self.request.get('phone')
            address = self.request.get('address')
            city = self.request.get('city')
            email = self.request.get('email')

            if (name == '' or surname == '' or phone == '' or address == ''):
                addClient.get(self,error='Υπήρξε λάθος με τα στοιχεία')
                return
            phone = phone.replace(' ','-')##Ean mesa sto tilefono iparxei keno to
                                           ##antikathistei me "-"
            ##Diereunisi stin basi dedomenon gia to an iparxei i idia kataxorisi
            que = db.Query(Client)
            que = que.filter('onoma =',name)
                             .filter('epitheto=',surname)
                             .filter('tilephono =',phone)
            results = que.fetch(limit=1)

        ##Ean o pelatis iparxei idi enimeronei ton xristi
            if len(results)>0:
                emfanise(self,'newclient.htm',{'error' : 'Ο πελάτης με τα στοιχεία
                                                          αυτά υπάρχει ήδη'})
                return
        except ValueError:##Ean ipirxe kapoio lathos me ta stoixeia pou eisaxthikan
                          ##enimeronei ton xristi
            addClient.get(self,error='Υπήρξε λάθος με τα στοιχεία που εισήγαγες')

        ##Eisagi ton stoixeion an den ipirxe lathos
        newuser = Client(onoma = name, epitheto = surname, tilephono = phone,
                         dieuthinsi = address, poli = city, email = email);
        newuser.put();
        ##Enimeroni ton xristi gia tin epitixia
        que = db.Query(Client).order('epitheto');
        client_list = que.fetch(limit=orio)
        emfanise(self,'showclient.htm',{'client_list' : client_list,'message' : 'Ο
                                                 πελάτης αποθηκεύτηκε επιτυχώς'})
        return
```

κώδικας 9: Αποθήκευση πελατών

Δημιουργούμε μία τάξη την addClient με δύο μεθόδους, την get και post. Ανάλογα με το αίτημα που δέχεται από τον φυλλομετρητή εκτελεί και την κατάλληλη συνάρτηση.

- Αν είναι get τότε αναλαμβάνει να αποστείλει την σελίδα newclient.htm όπου περιέχει μία φόρμα με τα πεδία για δημιουργία νέου πελάτη.
- Αν είναι post τότε παίρνει τα στοιχεία που εστάλησαν με την φόρμα και αρχικά τα περνάει σε μεταβλητές μέσα στο runtime environment. Ελέγχει εάν κάποιο από τα στοιχεία αυτά είναι κενό και αν ναι, τότε ενημερώνει τον χρήστη ότι υπήρξε λάθος με τα στοιχεία του. Το μήνυμα λάθους το περνάει μέσα σε ένα dictionary στην μεταβλητή error. Εάν έχουν εισαχθεί όλα τα στοιχεία τότε συνεχίζει και ελέγχει για διπλότυπα μέσα στο datastore. Τρέχει ένα ερώτημα πάνω στον πίνακα Client με κριτήρια το όνομα, επίθετο και τηλέφωνο. Αν βρει έστω και ένα, ενημερώνει ότι η εγγραφή υπάρχει ήδη πάλι μέσω της μεταβλητής error. Τέλος αν δεν έχει συμβεί κανένα από τα προηγούμενα συμβάντα, δημιουργεί το αντικείμενο και το αποθηκεύει στον πίνακα Client. Ενημερώνει τον χρήστη για την επιτυχή αποθήκευση μέσω της μεταβλητής message πλέον και όχι της error.



εικόνα 11: addclient.htm (με δοκιμαστικά message και error)

Στον κώδικα χρησιμοποιήσαμε και ένα block try-except ώστε να μην εισαχθούν δεδομένα που δεν θα μπορεί να χειριστεί η Python. Επίσης για την μεταβλητή phone για να υπάρχει συνέπεια στα δεδομένα, μέσω της replace αντικαταστήσαμε τα κενά που μπορεί να έβαλε ως είσοδο ο χρήστης, με παύλα. Η ανάκτηση των εγγραφών εμφανίζεται μέσω της showClient.htm και παρουσιάζεται παρακάτω (κώδικας 10):

```
class showClient(webapp.RequestHandler):
    def get(self):
        ##Γίνεται ανάκτηση των εγγραφών, ταξινομημένα σύμφωνα με το επίθετο, από
        ##τον πίνακα Client
        que = db.Query(Client).order('epitheto').order('onoma');
        client_list = que.fetch(limit=orio)
        ##Πέρασμα των αποτελεσμάτων στην μεταβλητή τύπου dictionary client_list και
        ##εμφάνιση τους
        emfanise(self,'showclient.htm',{'client_list' : client_list})
```

κώδικας 10: Εμφάνιση πελατών

Ο κώδικας σε Django μέσα στην showclient.htm που αναλαμβάνει να εμφανίσει τα αποτελέσματα (κώδικας 11) :

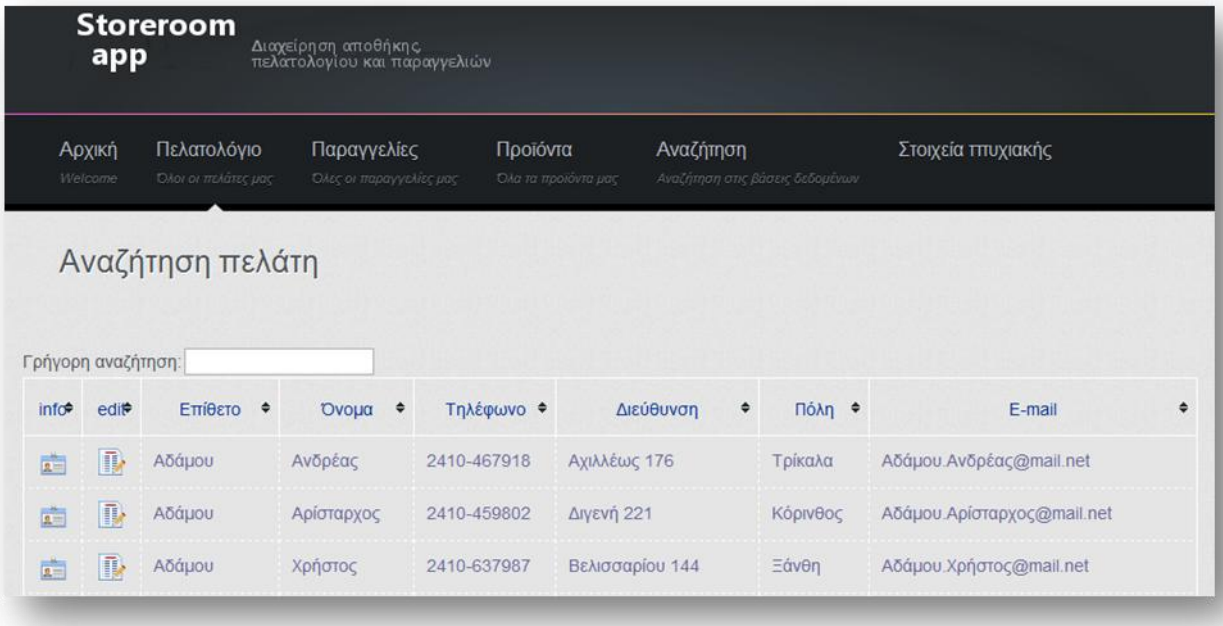
```
<table>
  {% for client in client_list %}
  <tr>
  <td><center>
      <form method="post" action="/infoClient">
        <input type="hidden" value="{{client.key}}" name="choice" />
        <input type="image" src="/static/img/info.png" alt="Submit button" />
      </form>
    </center></td>
  <td><center>
      <form method="get" action="/editClient">
        <input type="hidden" name="choice" value="{{client.key}}" />
        <input type="image" src="/static/img/edit.png" />
      </form>
    </center></td>
  <td> {{ client.epitheto   }} </td>
  <td> {{ client.onoma      }} </td>
  <td> {{ client.tilephono  }} </td>
  <td> {{ client.dieuthinsi }} </td>
  <td> {{ client.poli       }} </td>
  <td> {{ client.email      }} </td>
    </tr>
  {% endfor %}
</table>
```

κώδικας 11: Εμφάνιση πελατών μέσα από την showclient.htm

Δημιουργούμε έναν πίνακα που θα μας βοηθήσει να εμφανίσουμε με δομημένο τρόπο τα αποτελέσματα. Σε κάθε του γραμμή εμφανίζει τις πληροφορίες για τον κάθε πελάτη

ξεχωριστά. Στην πρώτη στήλη δημιουργεί μια εικόνα, όπου πατώντας της ενεργοποιεί το script infoclient που μας εμφανίζει πληροφορίες για τον πελάτη και παρουσιάζεται παρακάτω. Η δεύτερη στήλη περιλαμβάνει και αυτή μια εικόνα, που καλεί την editclient για την επεξεργασία των δεδομένων του πελάτη. Οι επόμενες στήλες γεμίζουν με τις πληροφορίες του κάθε πελάτη. Παίρνει από την python την dictionary μεταβλητή, client_list, και με την βοήθεια ενός for βρόγχου εμφανίζει για το κάθε αντικείμενο, μέσα στην μεταβλητή, τα στοιχεία που περιέχει. Έτσι δημιουργείτε ένας πίνακας για το πελατολόγιο που περιέχει σε κάθε γραμμή του τις εγγραφές των πελατών από το datastore.

Η σελίδα showclient.htm με μερικούς πελάτες παρουσιάζεται παρακάτω στην εικόνα 12. Αριστερά των εγγραφών οι στήλες info και edit εμφανίζουν πληροφορίες για τις παραγγελίες του κάθε πελάτη και την δυνατότητα για επεξεργασία των δεδομένων του πελάτη αντίστοιχα:



εικόνα 12: showclient.htm

Διαγραφή πελατών

Για την διαγραφή πελατών χρησιμοποιούμε την σελίδα delClient.htm . Μέσα σε αυτή την σελίδα, η φόρμα που δημιουργείτε δυναμικά περιέχει όλους τους πελάτες, μαζί με ένα checkbox κουτάκι όπου επιλέγεται ποιός θα διαγραφεί από τον πίνακα Client. Ο κώδικας της delClient.htm εμφανίζεται παρακάτω (κώδικας 12):

```
{% for client in client_list %}
  <tr>
    <td>
      <input type="checkbox" name="choosen{{ forloop.counter }}"
             value="{{ client.key }}" />
    </td>
    <td> {{ client.epitheto   }} </td>
    <td> {{ client.onoma      }} </td>
    <td> {{ client.tilephono  }} </td>
    <td> {{ client.dieuthinsi }} </td>
    <td> {{ client.poli       }} </td>
    <td> {{ client.email      }} </td>
  </tr>
{% endfor %}
```

κώδικας 12: Φόρμα επιλογής πελάτη προς διαγραφή

Χρησιμοποιούμε ένα for βρόγχο για την δημιουργία του πίνακα που περιέχει τις εγγραφές των πελατών. Στην πρώτη στήλη των εγγραφών ορίζουμε να δημιουργείτε ένα κουτάκι επιλογής (checkbox). Το όνομα για το κάθε κουτάκι το δημιουργούμε με συνένωση της συμβολοσειράς choosen και ενός μετρητή (counter) που αναλαμβάνει το Django να αυξάνει κατά μία μονάδα σε κάθε επανάληψη. Επίσης ως τιμή για το κάθε κουτάκι βάζουμε το κλειδί του πελάτη που έχουμε για την κάθε εγγραφή.

Έτσι όταν επιλέγουμε πολλά κουτάκια για διαγραφή έχουν τα απαραίτητα στοιχεία για τον χειριστή μας που εμφανίζεται παρακάτω (κώδικας 13) ώστε να κάνει την διαγραφή αυτών των αντικειμένων.

```
class delClient(webapp.RequestHandler):
    def get(self,message = '',error=''):
        que = db.Query(Client).order('epitheto');
        client_list = que.fetch(limit=orio)
        emfanise(self,'delclient.htm',{'client_list' : client_list,
                                       'message': message, 'error':error})
    def post(self):
        ##Stin epilogi apo tin forma ton pelaton pros diagrafi briskei poia apo ola
        ##exei epileksei o xristis
        j=0
        for i in range(1,orio):
            choosen = self.request.get('choosen'+str(i))
            if choosen:
                j = j +1
                ##Diagrafi apo tin basi ton pelaton
                db.delete(choosen)
        if j > 1:
            ##Enimerosi tou xristi gia epitixia diagrafis pollon pelaton
            delClient.get(self,message = 'Οι πελάτες διαγράφηκαν επιτυχώς')
        elif j == 0:
            delClient.get(self,error = 'Δεν επιλέχτηκε κανένας πελάτης')
```

```
        elif j == 1:
            ##Enimerosi tou xristi gia epitixia diagrafis enos pelati
            delClient.get(self,message = 'Ο πελάτης διαγράφηκε επιτυχώς')
        else:
            delClient.get(self,error = 'Error(deleting users): Υπήρξε λάθος στην
                                        βάση δεδομένων')
```

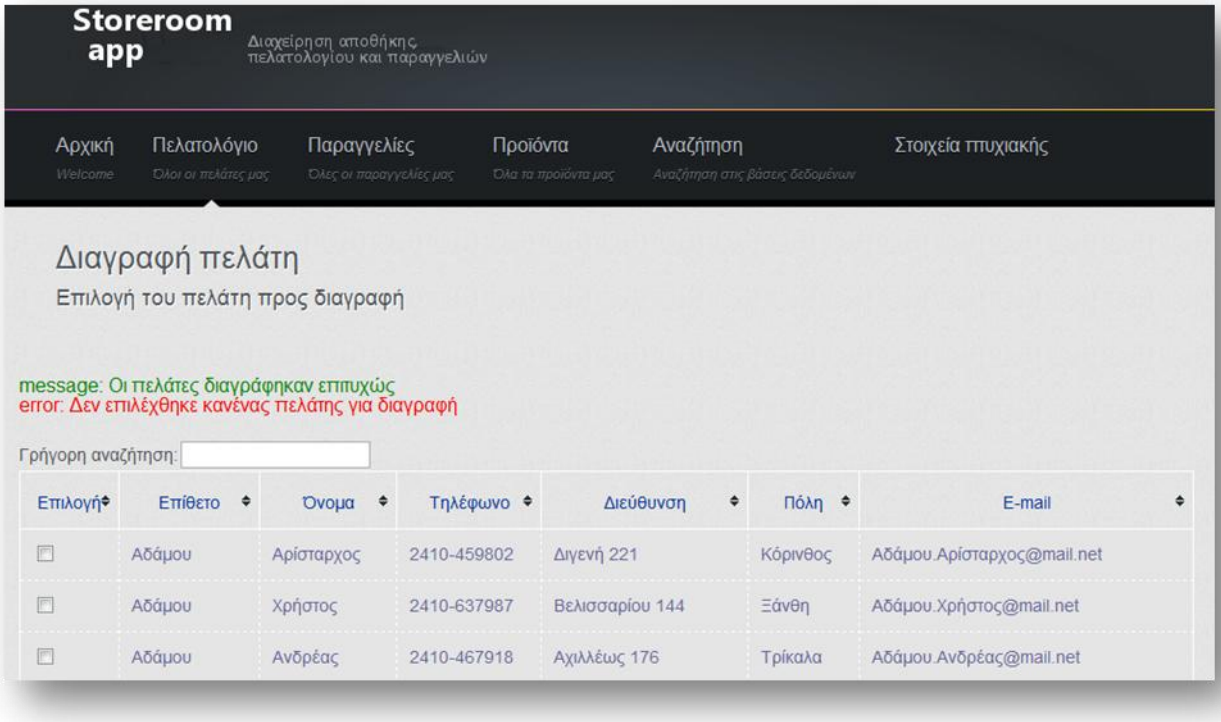
κώδικας 13: Διαγραφή πελάτη

Όπως σε όλες τις τάξεις που χειρίζονται αιτήματα έτσι και εδώ έχουμε δηλώσει δύο μεθόδους, την get και post.

- Στην get αναλαμβάνει απλά να εμφανίσει την λίστα με τους πελάτες που υπάρχουν στον πίνακα Client. Τους περνάει, μετά από το ερώτημα, στην μεταβλητή client_list την οποία και χρησιμοποιεί το Django για την εμφάνιση του καθένα πελάτη. Επίσης έχουμε ορίσει δύο μεταβλητές, message και error, που μπορούμε να τις υπερφορτώσουμε αργότερα για την παραγωγή μηνυμάτων προς τους χρήστες. Αρχικά είναι και οι δύο κενές μιας και θέλουμε απλή εμφάνιση του πελατολογίου.
- Στην post χρησιμοποιούμε ένα βρόγχο για να βρούμε τους επιλεχθέντες πελάτες προς διαγραφή. Μέσα στην φόρμα της html, το Django έχει δημιουργήσει στο checkbox μια μεταβλητή που ορίζει το choosen συν ένα αριθμό, για κάθε έναν πελάτη. Με την αποστολή της φόρμας, ο χειριστής μας, αναλαμβάνει να βρει ποιούς πελάτες έχει επιλέξει προς διαγραφή ο χρήστης, και να εκτελέσει την διαγραφή από τον πίνακα Client. Τέλος ενημερώνει τον χρήστη για την επιτυχία η αποτυχία του αιτήματος. Μιας και θέλουμε έπειτα από κάθε διαγραφή η εφαρμογή μας να επιστρέφει την ανανεωμένη λίστα των πελατών, καλούμε από την ίδια τάξη την μέθοδο get που περιγράψαμε νωρίτερα, περνώντας ως ορίσματα τα απαραίτητα μηνύματα.

Και εδώ χρησιμοποιούμε τις μεταβλητές message για την εμφάνιση μηνυμάτων επιτυχίας και του error για εμφάνιση αποτυχιών, όπου τις περνάμε μέσα στην emfanise όπου μπορεί να της χρησιμοποιήσει το Django για εμφάνιση περιεχομένου δυναμικά.

Στην εικόνα 13 εμφανίζεται η delclient.htm με δύο δοκιμαστικά μηνύματα επιτυχίας ή αποτυχίας. Επίσης αριστερά των εγγραφών τα checkbox εξυπηρετούν στην πολλαπλή επιλογή των πελατών προς διαγραφή.



εικόνα 13: delclient.htm

Επεξεργασία - ενημέρωση πληροφοριών πελάτη

Πολλά από τα στοιχεία των πελατών αλλάζουν, οπότε θα πρέπει να χρησιμοποιήσουμε έναν χειριστή που θα αναλαμβάνει την αλλαγή αυτών των δεδομένων στον πίνακα Client. Από την εμφάνιση των πελατών μέσα από το template showClient.htm επιλέγουμε από το αντίστοιχο κουμπάκι τον πελάτη που θέλουμε να αλλάξουμε. Αυτό με την συνέχεια καλεί τον χειριστή editClient μέσα στον κώδικα μας για να επεξεργαστεί το αίτημα που του δημιουργούμε. Ο κώδικας του editClient φαίνεται παρακάτω (κώδικας 14):

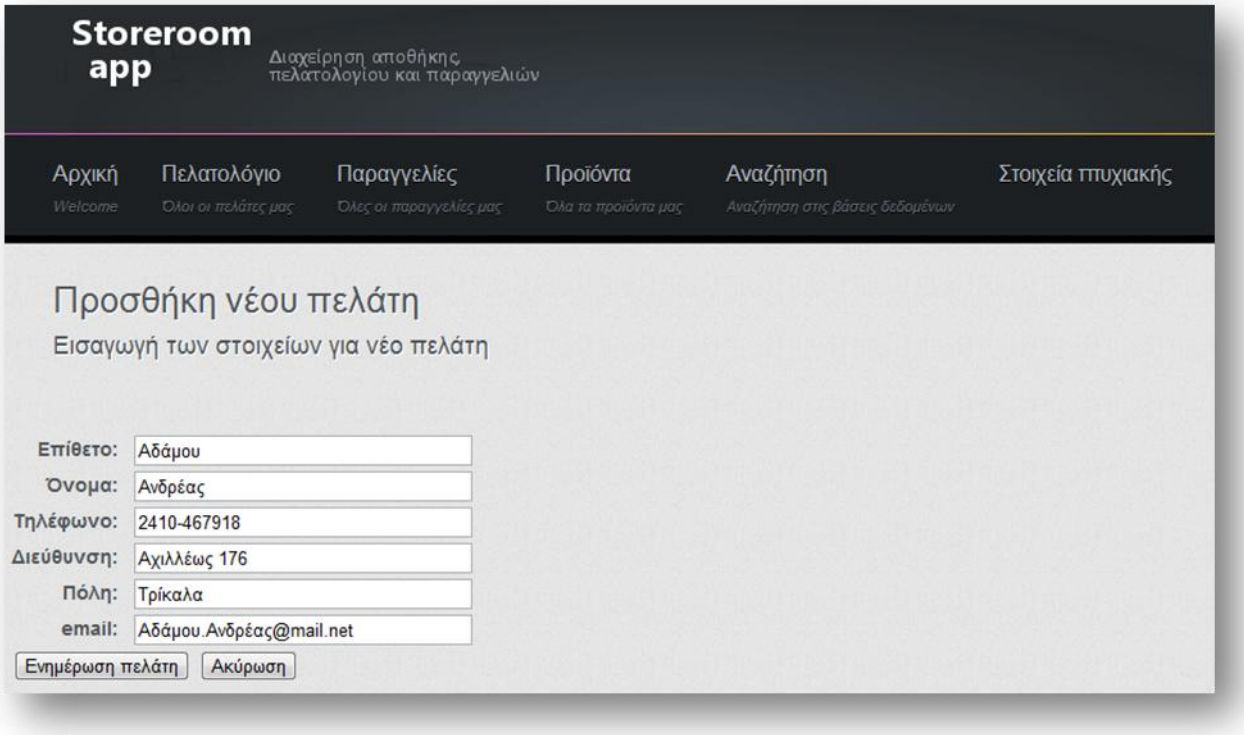
```
class editClient(webapp.RequestHandler):
    def get(self,message='', error=''):
        choosen=self.request.get('choice')
        obj = db.get(choosen)
        emfanise(self,'editclient.htm',{'client_list':obj,'message':message
                                                    ,'error':error})

    def post(self,message='',error=''):
        choosen = self.request.get('choice')
        name = self.request.get('name')
        surname = self.request.get('surname')
        phone = self.request.get('phone')
        address = self.request.get('address')
        city = self.request.get('city')
        email = self.request.get('email')
```

```
        db.delete(choosen)

        newuser = Client(key=db.Key(choosen), onoma = name, epitheto = surname,
                         tilephono = phone, dieuthinsi = address, poli = city,
                          email = email);
        newuser.put();

        que = db.Query(Client).order('epitheto');
        client_list = que.fetch(limit=orio)
        emfanise(self,'showclient.htm',{'client_list' : client_list,
                        'message':'Η ενημέρωση του πελάτη ολοκληρώθηκε'})
```

κώδικας 14: Επεξεργασία πληροφοριών πελάτη

- Αρχικά καλείται η μέθοδος GET από την φόρμα της showClient.htm σελίδας. Η μεταβλητή choosen παίρνει την τιμή της εγγραφής του πελατολογίου, που θέλουμε να επεξεργαστούμε. Ανακτούμε αυτό το αντικείμενο από το datastore στην μεταβλητή obj, και χρησιμοποιούμε την emfanise για την εμφάνιση της editClient.htm ώστε να αλλάξει ο χρήστης τα απαραίτητα δεδομένα του πελάτη. Επίσης τα ήδη υπάρχοντα δεδομένα του πελάτη περνάνε στην μεταβλητή client_list για εμφάνιση τους από το Django.
- Στην αποστολή των αλλαγμένων στοιχείων από τον χρήστη, ο browser του χρησιμοποιεί ένα POST αίτημα. Έτσι εκτελείτε η post συνάρτηση της editClient. Περνάει σε μεταβλητές τις αλλαγές που έχουν γίνει, διαγράφει το αντικείμενο από τον πίνακα Client, και δημιουργεί ένα νέο αντικείμενο με τα καινούργια στοιχεία για αποθήκευση. Χρησιμοποιεί το ήδη υπάρχον κλειδί της βάσης για τον πελάτη ώστε να μην δημιουργηθούν προβλήματα, κατά την αλλαγή των στοιχείων με τους άλλους πίνακες. Για να ορίσουμε ότι αυτό που θα αποθηκευτεί είναι σίγουρα τύπου key χρησιμοποιούμε το db.Key(). Αποθηκεύει το αντικείμενο, και ανακτά όλους του πελάτες του πελατολογίου ώστε να τους εμφανίσει ενημερωμένους πλέον στον χρήστη. Μέσα στο client_list, περνάει τους πελάτες ενώ μέσα στην μεταβλητή message, το μήνυμα επιτυχίας για την ενημέρωση του πελάτη.

Στην εικόνα 14 που ακολουθεί, εμφανίζεται η editclient.htm με μια εγγραφή πελάτη για επεξεργασία



εικόνα 14: editclient.htm

Αναζήτηση πελατών

Για την αναζήτηση των πελατών χρησιμοποιούμε μία φόρμα όπου υπάρχουν πεδία για το κάθε στοιχείο τους. Μπορούμε έτσι να αναζητήσουμε ονόματα, επίθετα, τηλέφωνα, email για να μας εμφανίσει όλες τις εγγραφές, που εκπληρώνουν αυτά τα κριτήρια. Η σελίδα searchClient.htm περιέχει μία φόρμα με αυτά τα πεδία και έναν πίνακα, αρχικά κενό, που θα μας εμφανίσει τις εγγραφές που ψάχνουμε. Ο χρήστης συμπληρώνει τα στοιχεία που θέλει στα πεδία, και η φόρμα με την μέθοδο POST, τα στέλνει στο app engine, όπου επιλέγεται ο χειριστής searchClient για την επεξεργασία και εμφάνιση αυτών των εγγραφών. Ο κώδικας της searchClient.htm (κώδικας 15):

```
<form method="post" action="/searchClient">
Αναζήτηση πελατών με βάση:
  <table>
    <tr>
      <th>Επίθετο:</th>
      <td><input type="text" name="surname" value=""><br></td>
    </tr>
    <tr>
      <th>Όνομα:</th>
      <td><input type="text" name="name" value=""><br></td>
    </tr>
    <tr>
      <th>Τηλέφωνο:</th>
```

```
      <td><input type="text" name="phone" value=""><br></td>
    </tr>
    <tr>
      <th>Διεύθυνση:</th>
      <td><input type="text" name="address" value=""><br></td>
    </tr>
    <tr>
      <th>Πόλη:</th>
      <td><input type="text" name="city" value=""><br></td>
    </tr>
    <tr>
      <th>E-mail:</th>
      <td><input type="text" name="email" value=""><br></td>
    </tr>
    <tr>
      <td><input type="submit" value="Αναζήτηση πελάτη"></td>
      <td><input type="reset" value="Ακύρωση" /></td>
    </tr>
  </table>
<form>
```

κώδικας 15: Φόρμα αναζήτησης πελατών

Ορίζουμε την φόρμα μας με μέθοδο post, και το action να καλεί τον χειριστή μας στο app engine. Για κάθε ένα πεδίο έχουμε δημιουργήσει και αντίστοιχει γραμμή στον πίνακα της html. Ξεχωρίζουμε το καθένα με το name και το value θα πάρει την τιμή που θα συμπληρώσει ο χρήστης στην φόρμα. Με το που αποσταλεί η φόρμα, θα καλέσει την μέθοδο POST από τον χειριστή μας, ο κώδικας του οποίου παρουσιάζεται παρακάτω (κώδικας 16):

```
class searchClient(webapp.RequestHandler):
    def get(self):
        emfanise(self,'searchclient.htm')

    def post(self):
        surname = self.request.get('surname')
        name = self.request.get('name')
        phone = self.request.get('phone')
        address = self.request.get('address')
        city = self.request.get('city')
        email = self.request.get('email')

        que = db.Query(Client)
        if surname == '':
            pass
        else:
            que = que.filter('epitheto =',surname)
        if name == '':
            pass
        else:
            que = que.filter('onoma =',name)
        if phone == '':
            pass
        else:
            que = que.filter('tilephono =',phone)
        if address == '':
            pass
```

```
        else:
            que = que.filter('dieuthinsi =',address)
        if city == '':
            pass
        else:
            que = que.filter('poli =',city)
        if email == '':
            pass
        else:
            que = que.filter('email =',email)

        client_list = que.fetch(limit=orio)

        if len(client_list) > 0:
            message = 'Βρέθηκαν ' + str(len(client_list)) + ' αποτελέσματα'
            emfanise(self,'searchclient.htm',{'client_list':client_list,
                                              'message':message})
        else:
            error='Δέν βρέθηκαν αποτελέσματα για τα κριτήρια που έδωσες'
            emfanise(self,'searchclient.htm',{'client_list':client_list,
                                              'error':error})
```

κώδικας 16: Handler αναζήτησης πελατών

Ο χειριστής searchClient αναλαμβάνει με τις δύο μεθόδους του, να αναζητήσει πελάτες στο πελατολόγιο, με τα κριτήρια του χρήστη, και να τα εμφανίσει στον χρήστη.

- Η μέθοδος GET αναλαμβάνει να εμφανίσει απλά την σελίδα searchClient.htm στον χρήστη. Είναι η σελίδα που εμφανίζεται χωρίς κανένα στοιχείο στον πίνακα του πελατολογίου.
- Η μέθοδος POST παίρνει το κάθε στοιχείο από την φόρμα και βάζει τα στοιχεία που πληκτρολόγησε ο χρήστης μέσα σε μεταβλητές. Στη συνέχεια δημιουργεί ένα αντικείμενο το que, που θα το χρησιμοποιήσει για να εκτελέσει το ερώτημα πάνω στο datastore. Μιας και τα ερωτήματα υπό κριτήρια χρειάζονται και κάποιο filter που να ενεργεί στο αντικείμενο του ερωτήματος, ο χειριστής αναλαμβάνει να βρεί ποια στοιχεία έχει συμπληρώσει ο χρήστης στην φόρμα. Ερευνά την κάθε μεταβλητή που δημιούργησε νωρίτερα για το αν είναι κενή ή όχι. Αν είναι κενή τότε συνεχίζει στην επόμενη. Αν δεν είναι κενή, χρεισιμοποιεί το filter, για να προσθέσει ένα κριτήριο αναζήτησης πάνω στο αντικείμενο que. Τέλος εκτελεί το ερώτημα και τα αποτελέσματα τα περνάει στην μεταβλητή client_list. Έπειτα ελέγχει αν βρέθηκαν αποτελέσματα με αυτά τα κριτήρια, ελέγχοντας το μέγεθος της μεταβλητής client_list, ώστε να εμφανίσει και τα κατάλληλα μηνύματα (message, error) στον χρήστη.

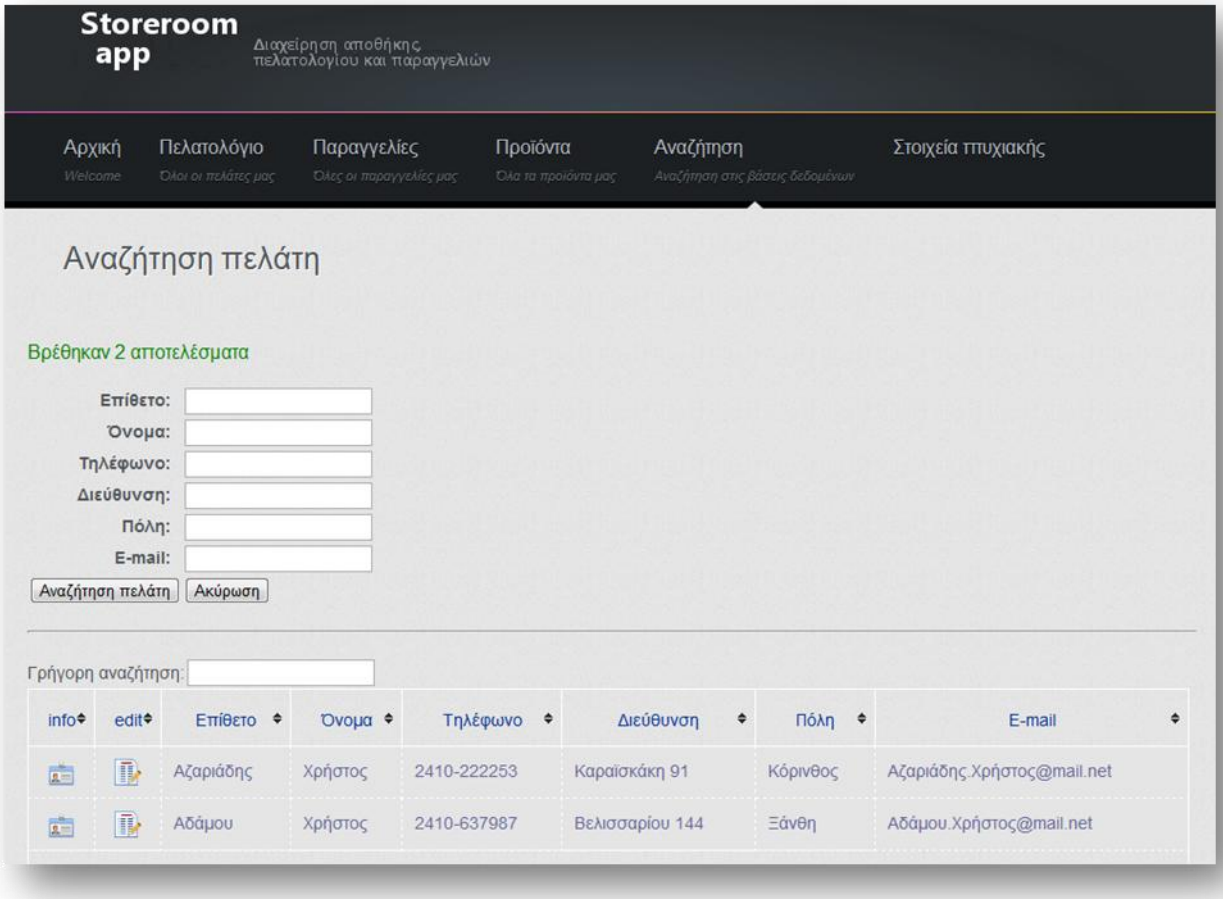
Η έξοδος του χειριστή την αναλαμβάνει το Django όπου χρησιμοποιεί την searchClient.htm ώστε να εμφανίσει τα αποτελέσματα. Ο πίνακας που χρησιμοποιείτε για την εμφάνιση τους παρουσιάζεται παρακάτω (κώδικας 17):

```
{% for client in client_list %}
<tr>
  <td>
    <center>
      <form method="post" action="/infoClient">
        <input type="hidden" value="{{client.key}}" name="choice"  />
        <input type="image" src="/static/img/info.png" alt="Submit button" />
      </form>
    </center>
  </td>
  <td>
    <center>
      <form method="get" action="editClient">
        <input type="hidden" name="choice" value="{{client.key}}" />
        <input type="image" src="/static/img/edit.png"/>
      </form>
    </center>
  </td>
  <td> {{ client.epitheto   }} </td>
  <td> {{ client.onoma      }} </td>
  <td> {{ client.tilephono  }} </td>
  <td> {{ client.dieuthinsi }} </td>
  <td> {{ client.poli       }} </td>
  <td> {{ client.email      }} </td>
</tr>
{% endfor %}
```

κώδικας 17: Εμφάνιση πελατών που βρέθηκαν από αναζήτηση

Ο παραπάνω κώδικας συνθέτει τον πίνακα αποτελεσμάτων για την αναζήτηση των πελατών. Χρησιμοποιεί μια γραμμή για κάθε εγγραφή. Πριν από τα στοιχεία του κάθε πελάτη, τοποθετείτε μια εικόνα για την διαχείριση αυτών των εγγραφών. Η πρώτα εικόνα καλεί τον χειριστή infoClient για την εμφάνιση των πληροφοριών του χρήστη. Η δεύτερη καλεί τον editClient, που είδαμε νωρίτερα για την επεξεργασία αυτής της εγγραφής. Και εδώ χρησιμοποιούμε ένα for βρόγχο για την εμφάνιση των αποτελεσμάτων από την python.

Στην εικόνα 15 παρουσιάζεται η searchclient.htm. Στο πάνω μέρος της κύριου τμήματος της html υπάρχει η φόρμα αναζήτησης με τα αντίστοιχα πεδία, ενώ κάτω από αυτή εμφανίζονται μερικά αποτελέσματα για αναζήτηση στους πελάτες του ονόματος Χρήστος.



εικόνα 15: searchclient.htm

Πληροφορίες πελατών

Σε μια εφαρμογή για την διαχείριση αποθήκης θα πρέπει να έχουμε εννοποιημένα και τις παραγγελίες, με τα προϊόντα και το ποσό που πλήρωσε ο πελάτης για αυτές. Αυτό το επιτυγχάνουμε με την χρησιμοποιήσει του χειριστή infoClient. Ο χειριστής αυτός ενεργοποιείται κάθε φορά που ο χρήστης πατάει πάνω στο κουμπάκι για εμφάνιση των πληροφοριών. Η φόρμα από το κάθε κουμπάκι περιέχει και το κλειδί της κάθε εγγραφής, ώστε ο χειριστής να εμφανίζει το αντικείμενο, στο datastore, που αναζητούμε. Ο κώδικας από αυτό το κουμπάκι, που εμφανίζεται με εικονίδιο, φαίνεται παρακάτω (κώδικας 18):

```
<form method="post" action="/infoClient">
        <input type="hidden" value="{{client.key}}" name="choice"  />
        <input type="image" src="/static/img/info.png" alt="Submit button" />
      </form>
```

κώδικας 18: Φόρμα επιλογής πληροφοριών πελάτη

Ενεργοποιείτε ο χειριστής infoClient ο κώδικας του οποίου αναλαμβάνει να βρει την εγγραφή στο πελατολόγιο που αναζητούμε, και να αναζητήσει έπειτα όλες τις παραγγελίες που έχει κάνει αυτός ο πελάτης. Το ιστορικό των παραγγελιών του εμφανίζεται σε έναν πίνακα κάτω από τις πληροφορίες του πελάτη, με πολλά από τα στοιχεία των παραγγελιών του. Ο κώδικας του χειριστή εμφανίζεται παρακάτω (κώδικας 19):

```
class infoClient(webapp.RequestHandler):
    def post(self):
        choosenkey = self.request.get('choice')
        if choosenkey == 'Nonexist':
            emfanise(self,'infoclient.htm',{'error':'Ο πελάτης έχει διαγραφεί ή
                                                έχουν αλλάξει τα στοιχεία του'})
        else:
            client_list = Client.get(choosenkey)
            que = db.Query(Basi)
            que=que.filter('id_pelatis =',choosenkey)
            results = que.fetch(limit=orio_med)
            idparaggelia = []
            for i in results:
                idparaggelia.append(str(i.id_paraggelia))
            mylista =[]
            for i in range(len(idparaggelia)):
                que = db.Query(Order)
                que = que.filter('id_paraggelia=',idparaggelia[i]).order('created')
                results = que.fetch(limit=1)
                mylista.append(results)
emfanise(self,'infoclient.htm',{'client_list':client_list,'mylista':mylista})
```

κώδικας 19: Handler πληροφοριών πελάτη

Μιας και ο χειριστής μας ενεργοποιείτε μέσα από το πάτημα ενός κουμπιού, που περιέχει ήδη της απαραίτητες πληροφορίες, δεν χρειαζόμαστε κάποια μέθοδο GET. Η POST ξεκινάει, διαβάζοντας από την φόρμα που στάλθηκε το κλειδί του πελάτη που μας ενδιαφέρει. Ελέγχει αν το κλειδί αυτό ισχύει. Σε περίπτωση που ο πελάτης έχει διαγραφή, ή το κλειδί του έχει αλλάξει τότε το app engine βάζει την τιμή Nonexist. Αν το κλειδί υπάρχει, ανακτούμε τα στοιχεία από το datastore, ώστε να τα εμφανίσουμε πάνω από τον πίνακα ιστορικού της σελίδας.

Δημιουργούμε ένα αντικείμενο ερωτήματος, για την βάση Basi, ώστε να ανακτήσουμε όλες τις παραγγελίες αυτού του χρήστη. Μέσα στην βάση για αποθηκεύουμε κλειδιά χρηστών, και κωδικούς παραγγελίας. Τρέχουμε το ερώτημα πάνω στην Basi, με κριτήριο να υπάρχει κάποια id_pelatis με το επιλεγμένο κλειδί. Τα αποτελέσματα τα περνάμε στην μεταβλητή results, και δημιουργούμε μια κενή λίστα, την idparaggelia, μέσα στην οποία θα εισάγουμε τους κώδικούς παραγγελίων του συγκεκριμένου χρήστη. Δημιουργούμε μια ακόμα κενή λίστα, την mylista, και την γεμίζουμε με έναν for βρόγχο, όπου ανακτούμε τις

παραγγελίες, με κριτήρια τους κωδικούς που βρίσκονται στην λίστα idparaggelia. Τέλος επιστρέφουμε τα στοιχεία του πελάτη, client_list, και το ιστορικό των παραγγελιών του μέσω της mylista.

Το Django αναλαμβάνει να μας τα εμφανίσει μέσα στην σελίδα infoClient.htm. Στο πρώτο μέρος της σελίδας εμφανίζει τα στοιχεία του πελάτη, και στο δεύτερο δημιουργείτε ένας πίνακας που γεμίζει με τις παραγγελίες που έχει κάνει ο πελάτης συνολικά. Το προϊόν, η τιμή που πλήρωσε, η έκπτωση που είχε καθώς και η ημερομηνία παραγγελίας εμφανίζονται με δομημένο τρόπο μέσα στον πίνακα.

Μιας και έχουμε μία εγγραφή για τον πελάτη που ψάχνουμε δεν χρειαζόμαστε κάποιο βρόγχο for, για να εμφανίσουμε τα στοιχεία της λίστας client_list. Μπορούμε να προσπελάσουμε τα δεδομένα της καλόντας τις ιδιότητες του αντικειμένου. Για την λίστα με τα προϊόντα αντίθετα, χρειαζόμαστε έναν τέτοιο βρόγχο μιας και ψάχνουμε μέσα σε μια λίστα που περιέχει περισσότερα του ενός εγγραφές.

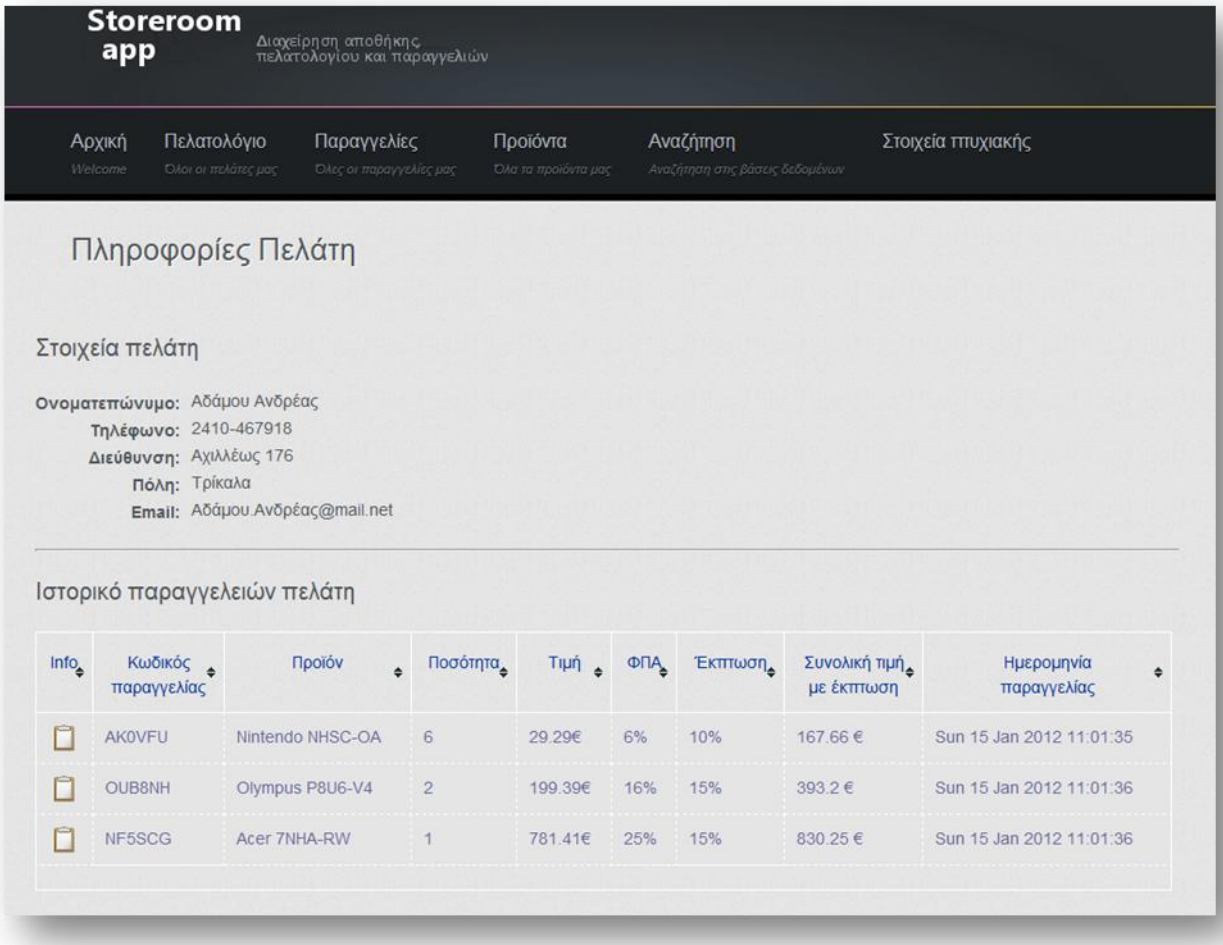
Ο κώδικας της σελίδας infoclient.htm εμφανίζεται παρακάτω (κώδικας 20):

```
<p><h6>Στοιχεία πελάτη</h6><br />
    <table>
      <tr><th>Ονοματεπώνυμο:</th>
           <td>{{ client_list.epitheto }} {{ client_list.onoma }}<br /></td></tr>
      <tr><th>Τηλέφωνο:</th>
          <td>{{ client_list.tilephono }}</td></tr>
      <tr><th>Διεύθυνση:</th>
          <td>{{ client_list.dieuthinsi }}</td></tr>
      <tr><th>Πόλη:</th>
          <td>{{ client_list.poli }}</td></tr>
      <tr><th>Email:</th>
          <td>{{ client_list.email }}</td></tr>
    </table></p>
<table>
{% for i in mylista %}
  <tr>
    <td><center>
        <form method="post" action="/infoOrder" >
          <input type="hidden" name="choice" value="{{i.0.key}}" />
          <input type="image" src="/static/img/order.png" />
        </form>
      </center></td>
    <td> {{ i.0.id_paraggelia                }}</td>
    <td> {{ i.0.proion                       }}</td>
    <td> {{ i.0.posotita                     }}</td>
    <td> {{ i.0.timi                         }}&#8364;</td>
    <td> {{ i.0.fpa                          }}%</td>
    <td> {{ i.0.ekptosi                      }}%</td>
    <td> {{ i.0.sinoloekpt                   }}&#8364;</td>
    <td> {{ i.0.created|date:"D d M Y H:m:s"  }}</td>
  </tr>
{% endfor %}
</table>
```

κώδικας 20: Εμφάνιση πελατών - infoclient.htm

Στον πρώτο for βρόγχο εμφανίζουμε τα στοιχεία του πελάτη όπως τα παίρνουμε μέσα από το datastore. Στον δεύτερο διαβάζουμε τα στοιχεία που αποθηκεύσαμε στην λίστα mylista που φτιάξαμε και εμφανίζουμε όλα τα προϊόντα που έχει παραγγείλει ο πελάτης ως τώρα. Το μηδέν στις μεταβλητές δηλώνει την θέση των στοιχείων μέσα στην λίστα μιας και είναι μια λίστα που φτιάξαμε και θέλει περισσότερα στοιχεία για το που βρίσκονται. Χρησιμοποιούμε μια διαμόρφωση στο Django για το πως θα μας εμφανίσει την ημερομηνία και ώρα των παραγγελιών.Το D είναι για την ημέρα ολογράφως, d για αριθμό της ημέρας, το M για τον μήνα ολογράφως, το Y για τον χρόνο και τα H:m:s για ώρα, λεπτά, δευτερόλεπτα.

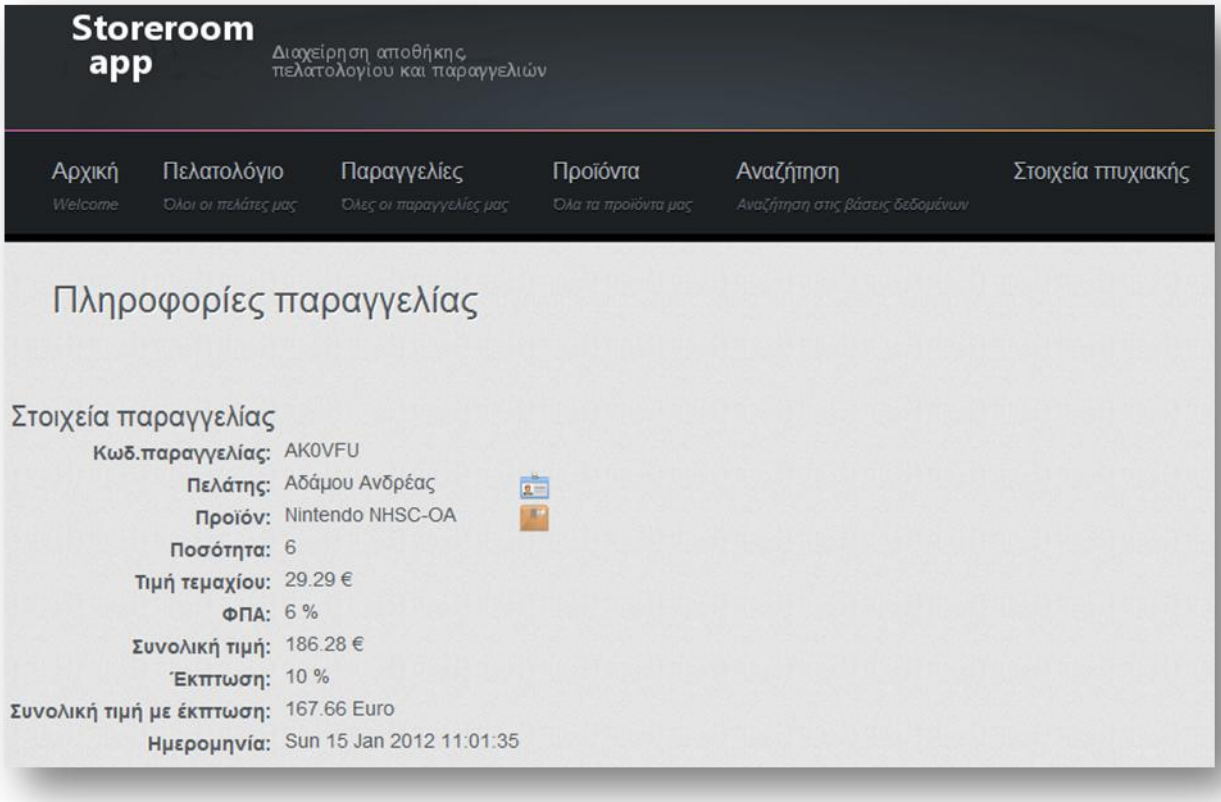
Στην εικόνα 16 που ακολουθεί εμφανίζονται οι πληροφορίες ενός πελάτη στο πάνω μέρος καθώς και όλες οι παραγγελίες που έχει κάνει ο πελάτης από κάτω:



εικόνα 16: infoclient.htm

Από το ιστορικό παραγγελιών του πελάτη μπορούμε να επιλέξουμε να δούμε στοιχεία συγκεκριμένα για μια παραγγελία. Πατώντας αριστερά στα εικονίδια της στήλης info, για κάποια παραγγελία, μεταφερόμαστε στην infoOrder.htm ,όπως στην εικόνα 17, όπου μας εμφανίζονται

περισσότερα στοιχεία για την παραγγελία. Για την επιλεγμένη παραγγελία, διαβάζουμε τα στοιχεία από τον πίνακα Order και τα εμφανίζουμε στον χρήστη. Σε αυτό μας βοηθάει ο χειριστής infoOrder που εμφανίζεται παρακάτω.



εικόνα 17: infoorder.htm

```
class infoOrder(webapp.RequestHandler):
    def post(self):
        choosenkey = self.request.get('choice')##Pairnei to epilegmeno kleidi apo
                                                ## tin forma
        lista = Order.get(choosenkey)##Pairnei ta stoixeia apo tin basi Order gia
                                      ##auto to kleidi

        (myepitheto,myonoma) = lista.pelatis.split(' ')##xorizei ta enomena
                                                       ##epitheto-onoma
        que = db.Query(Client)
        que = que.filter('epitheto =',myepitheto)
                 .filter('onoma =',myonoma)##psaxnei mesa stin basi Client gia
                                           ##auto to onoma kai epitheto
        results = que.fetch(limit=1)
        if len(results) == 1:
            for i in results:
                key_pelatis = str(i.key())##pairnei to kleidi tou sigkekrimenou
                                          ##client me to parapano onoma kai epitheto

        else:
            key_pelatis = 'Nonexist'

        (mymarka,mymodelo) = lista.proion.split(' ')##xorizei ta enomena marka-
                                                     ##modelo
```

```
        que=db.Query(Item)
        que = que.filter('marka =',mymarka).filter('modelo =',mymodelo)
##psaxnei mesa tin basi Item gia auti ti marka kai modelo
        results = que.fetch(limit=1)
        if len(results) == 1:
            for i in results:
                key_item = str(i.key())##pairnei to kleidi tou sigkrkrimenou item
                                       ##me tin parapano marka kai modelo
            emfanise(self,'infoorder.htm',{'lista':lista,'pelatis':key_pelatis,
                                           'antikeimeno':key_item})
        else:
            key_item='Nonexist'
            emfanise(self,'infoorder.htm',{'lista':lista,'pelatis':key_pelatis,
                                           'antikeimeno':key_item})
```

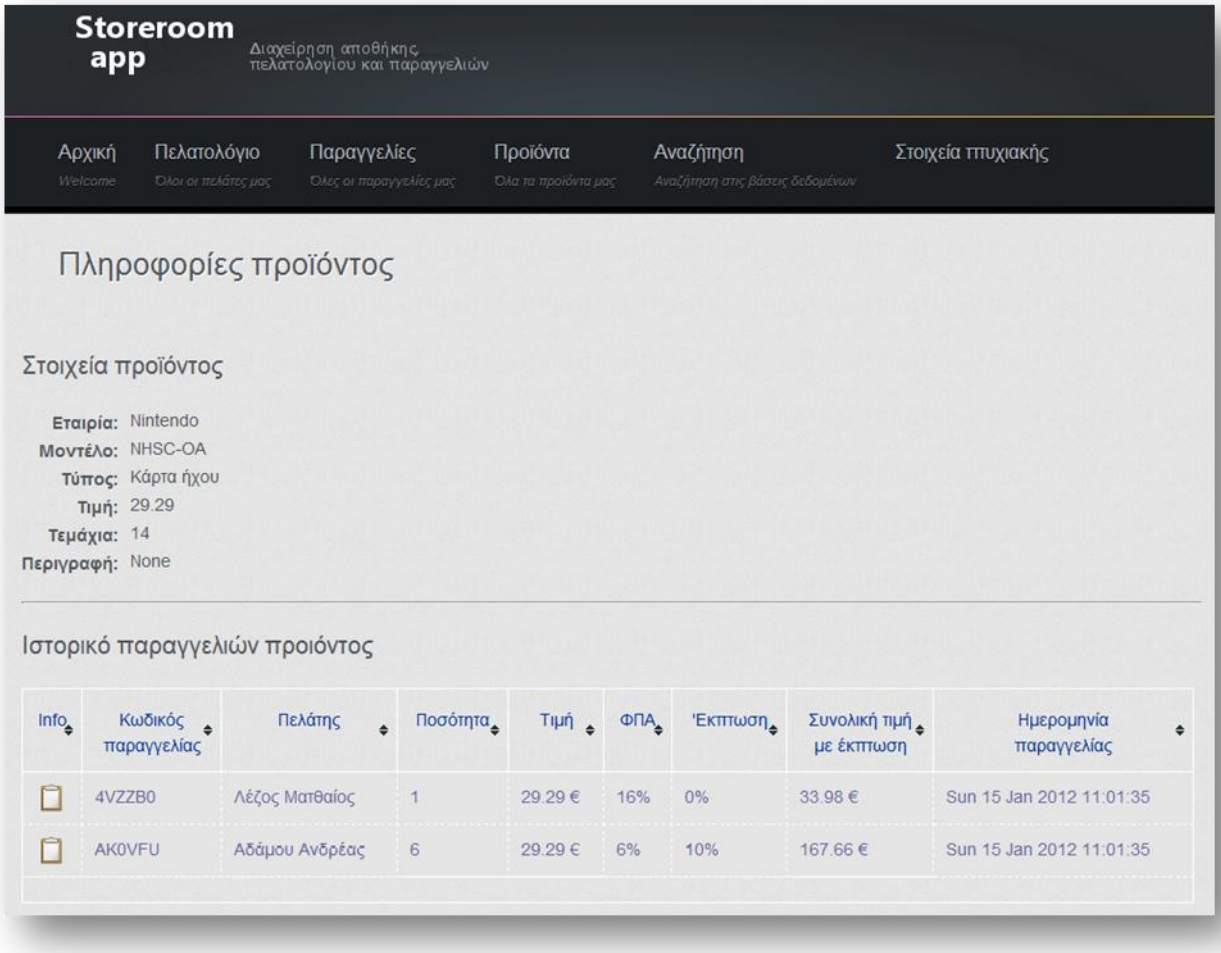
κώδικας 21: Χειριστής infoOrder

Επίσης μέσα από την infoOrder μπορούμε να επιλέξουμε είτε στο εικονίδιο δίπλα από τον χρήστη που θα μας επιστρέψει στης πληροφορίες πελάτη (infoclient.htm) εικόνα 16, είτε το εικονίδιο δίπλα από το προϊόν που θα μας εμφανίσει τις πληροφορίες του προϊόντος (infoitem.htm) καθώς και το ιστορικό παραγγελίας από άλλους πελάτες για αυτό το προϊόν εικόνα 18.

Η εμφάνιση των πληροφοριών του προϊόντος επιτυγχάνεται με την βοήθεια του χειριστή infoItem (κώδικας 22). Επιλέγοντας το εικονίδιο του προϊόντος που μας ενδιαφέρει, το αίτημα στέλνετε για επεξεργασία στον infoItem. Αυτός ανακατάει από την βάση Item το επιλεγμένο προϊόν και έπειτα ψάχνει στον πίνακα Basi για αντίστοιχες εγγραφές με αυτό το κλειδί. Πέρνει τα κλειδιά των πελατών από αυτόν και ψάχνει στον πίνακα παραγγελιών για αυτούς τους πελάτες. Τέλος επιστρέφει τα αντικείμενα από την παραγγελία για εμφάνιση από την infoitem.htm - εικόνα 18.

```
class infoItem(webapp.RequestHandler):
    def post(self):
        choosenkey = self.request.get('choice')
        if choosenkey == 'Nonexist':
            emfanise(self,'infoitem.htm',{'error':'Το προϊόν έχει διαγραφεί ή έχουν
                                                   αλλάξει τα στοιχεία του'})
        else:
            item_list = Item.get(choosenkey)##Pairno ta stoixeia tou proiontos gia
                                            ##auto tho choosenkey
            que = db.Query(Basi)
            que=que.filter('id_item =',choosenkey)##Psaxno stin basi Basi gia to
                                                  ##proion me auto ton kodiko
            results = que.fetch(limit=orio_med)
            idparaggelia = []
            for i in results:
                idparaggelia.append(str(i.id_paraggelia))
            mylista =[]
            for i in range(len(idparaggelia)):
                    que = db.Query(Order)
                    que = que.filter('id_paraggelia =',idparaggelia[i])
                                     .order('created')
                    results = que.fetch(limit=1)
                    mylista.append(results)
            emfanise(self,'infoitem.htm',{'mylista':mylista,'item_list':item_list})
```

κώδικας 22: χειριστής infoItem



εικόνα 18: infoitem.htm

Εάν κάποιος επιλέξει πάλι από το ιστορικό κάποια παραγγελία από το αντίστοιχο εικονίδιο τότε θα του εμφανιστεί η σελίδα infoorder.htm (εικόνα 17).

Κώδικας για τα προϊόντα

Μιας και ο κώδικας για τα προϊόντα, ανάκτηση τους (showItem), διαγραφή του (deleteItem) , αναζήτηση (searchItem) και αποθήκευση τους δεν διαφοροποιείτε ιδιαίτερα ως προς τις πράξεις μέσα στον κώδικα, από ότι με τους πελάτες δεν θα γίνει παρουσίαση του κώδικα αυτού.

Αποθήκευση και ανάκτηση μοντέλων παραγγελίας

Για την αποθήκευση και ανάκτηση μιας παραγγελίας γίνεται μια πιο πολύπλοκη εργασία μιας και για κάθε παραγγελία θα πρέπει να δημιουργούνται εγγραφές από τους άλλους δύο πίνακες, πελατολόγιο και προϊόντα. Ο πίνακας παραγγελίες λειτουργεί και σαν ιστορικό των παραγγελιών, μιας και αν αλλάξουν ή διαγραφούν τα στοιχεία των πελατών ή

των προϊόντων δεν θα γίνουν αλλαγές στις εγγραφές των παραγγελιών. Χρησιμοποιείτε επίσης ο πίνακας Basi για να κρατάει τα κλειδιά των αντικειμένων που θα χρησιμοποιηθούν ώστε να γίνεται η διασύνδεση των πινάκων.

Αρχικά δημιουργούμε μια φόρμα εισαγωγής της παραγγελίας, μέσα στην σελίδα. newOrder.htm. Υπάρχουν 4 πεδία μέσα στην φόρμα αποστολής. Ο χρήστης της εφαρμογής επιλέγει τον πελάτη, το προϊόν, το ΦΠΑ καθώς και την ποσότητα των προϊόντων για την παραγγελία. Η φόρμα αποστέλλετε στον χειριστή addOrder με την μέθοδο Post ώστε να εκτελέσει την επεξεργασία και αποθήκευση των δεδομένων.

Ο κώδικας της newOrder.htm παρουσιάζεται παρακάτω (κώδικας 21):

```
<form method="post" action="/addOrder" >
  <table>
    <tr>
      <th>Πελάτης:</th>
        <td>
          <select name="client">
          {% for who in client_list %}
            <option>{{who.epitheto}} {{who.onoma}} {{who.tilephono}}</option>
          {% endfor %}
          </select>
        </td>
      </tr>
    <tr>
    <th>Προϊόν:</th>
      <td>
        <select name="item">
        {% for what in item_list %}
          <option>{{what.marka }} {{what.modelo}}</option>
        {% endfor %}
        </select>
      </td>
    </tr>
  <tr>
  <th>Φπα:</th>
    <td>
      <select name="foros">
        <option value="6">6%</option>
              ...
        <option value="25">25%</option>
      </select>
    </td>
  <tr>
  <th>Ποσότητα:</th>
    <td>
      <input type="text" name="quantity" value="" />
    </td>
  </tr>
</table>
    <input type="submit" value="Εισαγωγής παραγγελίας">
```

```
    <input type="reset" value="Ακύρωση"></form>
```

κώδικας 23: Φόρμα παραγγελίας - neworder.htm

Αρχικά χρησιμοποιούμε έναν βρόγχο for για την εμφάνιση των ονομάτων, επιθέτων και τηλεφώνων των πελατών που μας αποστέλλει ο χειριστής addOrder μέσω της μεθόδου του Get. Είναι μία λίστα dropdown όπου ο χρήστης επιλέγει τον πελάτη. Ένας δεύτερος βρόγχος χρησιμοποιείται για να δημιουργήσει μία λίστα dropdown για την επιλογή του προϊόντος. Επίσης επιλέγεται ο φόρος και στο τελευταίο πεδίο εισάγεται η ποσότητα των τεμαχίων της παραγγελίας. Η αποστολή γίνεται στην Post μέθοδο του χειριστή που εμφανίζεται παρακάτω (κώδικας 22):

```
class addOrder(webapp.RequestHandler):
    def get(self,error='',message=''):
        que1 = db.Query(Client).order('epitheto');
        client_list = que1.fetch(limit=orio)

        que2 = db.Query(Item).order('marka');
        item_list = que2.fetch(limit=orio)
        emfanise(self,'neworder.htm',{'client_list':client_list,
                        'item_list':item_list,'error':error,'message':message})

    def post(self):
        try:
            client = self.request.get('client')
            item = self.request.get('item')
            foros = int(self.request.get('foros'))
            quantity = int(self.request.get('quantity'))

            if client == '' or item == '' or foros == '' or quantity == '':
                addOrder.get(self,error='Υπήρξε λάθος με τα στοιχεία που έδωσες')


            (myepitheto,myonoma,mytilephono) = client.split(' ')
            (mymarka,mymodel) = item.split(' ')
            key_pelatis = stoixeia_pelati(myepitheto,myonoma,mytilephono
             discount = ekptosi(myepitheto,myonoma,quantity)
            (key_item,temaxio) = stoixeia_antikeimenou(mymarka,mymodel)
            temaxio = temaxio + temaxio*foros/100
            sinoliko = quantity * temaxio
            mekptosi = sinoliko - sinoliko*discount/100.0

            al_tem = allagi_temaxion(self,key_item,quantity)
            paraggelia = kodikos_par

            if al_tem == True:
                neworder = Order(id_paraggelia = paraggelia,
                pelatis = myepitheto + ' ' + myonoma,
                proion = item, posotita = quantity, timi = float(temaxio),
                fpa = foros, sinolo = float(sinoliko),
                ekptosi = discount, sinoloekpt = mekptosi)

                newbasi = Basi(id_paraggelia = paraggelia,
                               id_pelatis = key_pelatis, id_item = key_item)
```

```
                neworder.put();
                newbasi.put();
                self.redirect("/showOrder")
            else:
                addOrder.get(self,error='Τα διαθέσιμα τεμάχια δεν επαρκούν για την
παραγγελία')
        except ValueError:
            addOrder.get(self,error='Υπήρξε λάθος με τα στοιχεία που έδωσες')
```

κώδικας 24: Handler δημιουργίας παραγγελίας

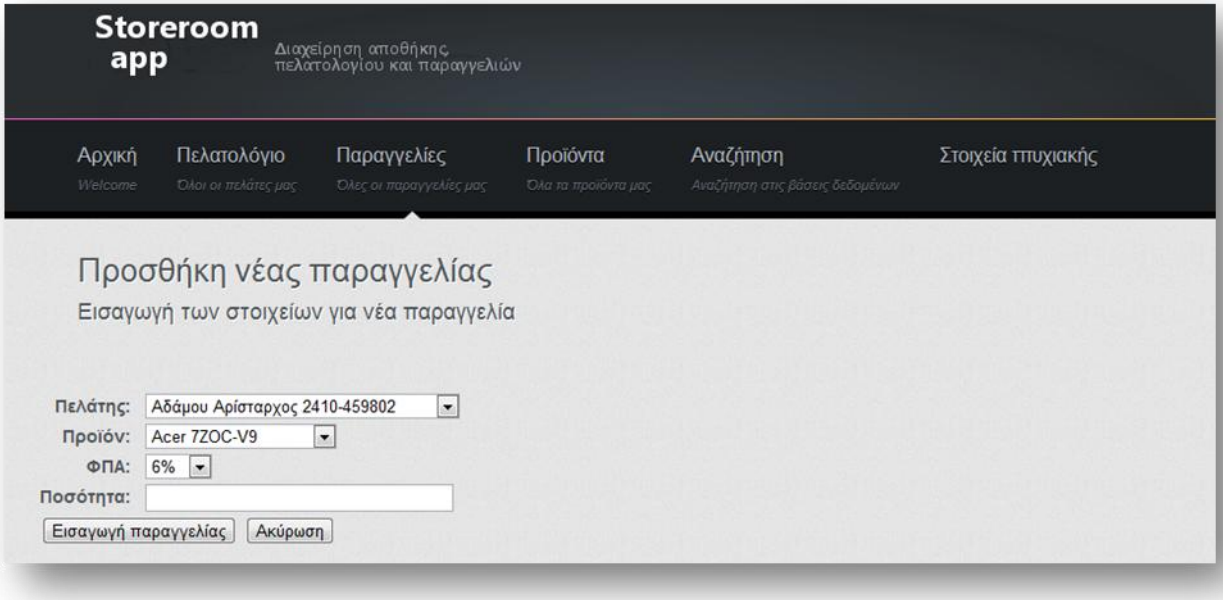
- Η μέθοδος get χρησιμοποιείται για να δημιουργήσει τις δύο λίστες που θα επεξεργαστεί ο κώδικας Django για να εμφανίσει τις επιλογές μέσα στη σελίδα.
- Η μέθοδος Post αναλαμβάνει τα δεδομένα που στέλνει η φόρμα, και τα περνάει σε μεταβλητές για τον χειρισμό τους. Ελέγχει αν κάποιο από αυτά τα στοιχεία είναι κενό και αν ναί τότε καλεί την μέθοδο get για να εμφανίσει ξανά την φόρμα για την παραγγελία καθώς και το μήνυμα αποτυχίας που δημιουργήθηκε. Έαν όλα τα στοιχεία είναι σωστά τότε χειρίζεται αρχικά τον πελάτη. Χρησιμοποιεί την συνάρτηση split της python για να διαχωρίσει το όνομα, επίθετο και τηλέφωνο που παίρνει από την φόρμα, και τα εισάγει σε τρείς μεταβλητές. Το ίδιο κάνει και για το προϊόν. Χρησιμοποιεί την συνάρτηση stoixeia_pelati, που παρουσιάζεται παρακάτω, για να βρεί το κλειδί του πελάτη. Έπειτα για να υπολογίσει την έκπτωση καλεί την συνάρτηση ekptosi και εισάγει την έξοδο της στην μεταβλητή discount.

Στην συνέχεια μέσα από την συνάρτηση stoixeia_antikeimenou ανακαλεί το κλειδί του προϊόντος και την τιμή του και τις περνάει σε δύο μεταβλητές. Υπολογίζει την τιμή του προϊόντος σύμφωνα με το ΦΠΑ που επέλεξε ο χρήστης και το συνολικό ποσό μαζί με ΦΠΑ της παραγγελίας. Στην μεταβλητή mekptosi υπολογίζει την τιμή της παραγγελίας μετά την έκπτωση για τον κάθε πελάτη. Στη συνέχεια μειώνει τα διαθέσιμα τεμάχια που υπάρχουν μέσα στην αποθήκη για το προϊόν την παραγγελίας, με την βοήθεια της συνάρτηση al_temaxion. Τέλος χρησιμοποιεί την συνάρτηση kodikos_par για να παράγει έναν κωδικό μοναδικό για την κάθε παραγγελία.

Για να εκτελέσει την εισαγωγή της παραγγελίας ελέγχει αν έγινε επιτυχώς η αλλαγή στα διαθέσιμα τεμάχια. Αν δεν έγινε επιτυχώς σημαίνει ότι τα τεμάχια μέσα στην αποθήκη για το προϊόν δεν επαρκούν για την παραγγελία. Αλλιώς φτιάχνει δύο αντικείμενα neworder και newbasi και τα εισάγει στους αντίστοιχους πίνακες του datastore. Τέλος χρησιμοποιούμε την self.redirect ώστε να κλειθεί η get μέθοδος της

showOrder για να εμφανίσει όλες τις παραγγελίες καθώς και την παραγγελία που μόλις αποθηκεύτηκε. Ο κώδικας try, except ενεργοποιείτε όταν κάποιο από τα στοιχεία που στάλθηκαν από την φόρμα δεν μπορεί να τα διαχειριστεί το datastore.

Στην εικόνα 19 εμφανίζεται η φόρμα εισαγωγής της παραγγελίας:



εικόνα 19: neworder.htm

Παρακάτω ακολουθούν οι τέσσερις συναρτήσεις που χρησιμοποιήθηκαν ώστε να γίνει επιτυχώς η αποθήκευση της παραγγελίας:

- Η συνάρτηση stoixeia_pelati (κώδικας 23) παίρνει ώς ορίσματα τρείς μεταβλητές, myepitheto, myonoma και mytilephono. Δημιουργεί ένα αντικείμενο, το que. Προσθέτουμε τα κριτήρια αναζήτησης ώστε να βρούμε τον πελάτη που ψάχνουμε σύμφωνα με το επίθετο, όνομα και τηλέφωνο. Τέλος εκτελούμε το ερώτημα και τα αποτελέσματα τα περνάμε στην μεταβλητή results, η οποία περιέχει πλέον τον ζητούμενο πελάτη. Βρίσκουμε το κλειδί του και το επιστρέφουμε στην μεταβλητή, στον κώδικα addOrder, από την οποία κλήθηκε η συνάρτηση.

```
def stoixeia_pelati(myepitheto,myonoma,mytilephono):
    que = db.Query(Client)
    que = que.filter('epitheto =',myepitheto).filter('onoma =',myonoma)
                                             .filter('tilephono =',mytilephono)
    results = que.fetch(limit=1) ##Εκτελεί το ερώτημα
    for i in results:
        key_pelatis = str(i.key())
    return key_pelatis##Και επιστρέφει το κλειδί του πελάτη
```

κώδικας 25: Συνάρτηση εύρεσης κλειδιού πελάτη

- Η συνάρτηση allagi_temaxion (κώδικας 24) χρησιμοποιείτε για να ελέγξει τα διαθέσιμα τεμάχια του προϊόντος και να τα μειώσει ώστε να εκτελεστεί επιτυχώς η παραγγελία. Πέρνει ώς όρισμα το κλειδί του προϊόντος και την ποσότητα της παραγγελίας. Ανακτά από τον πίνακα Item το ζητούμενο προϊόν, και ελέγχει αν η διαθέσιμη ποσότητα είναι επαρκής για να ικανοποίηση την ζήτηση της παραγγελίας.

```
def allagi_temaxion(self,key, posotita): ##Ελεγχει αν υπάρχουν διαθέσιμα τα
    obj = db.get(key)                    ##ζητούμενα τεμάχια
    if obj.temaxia <=0:
        return False
    elif obj.temaxia > 0 and obj.temaxia < posotita:
        return False
    else:
        obj.temaxia = obj.temaxia - posotita
        obj.put()
        return True
```

κώδικας 26: Συνάρτηση αλλαγής τεμαχίων προϊόντων

- Η συνάρτηση έκπτωση (κώδικας 25) ορίζεται με τρεις μεταβλητές που λαμβάνει από το κώδικα addOrder που την κάλεσε. Η συνάρτηση αυτή ψάχνει μέσα στον πίνακα Order και βρίσκει όλες τις παραγγελίες που αφορούν ένα πελάτη. Για να υπολογίσει την έκπτωση αθροίζει τις έως τώρα ποσότητες των παραγγελιών του και ανάλογα δημιουργεί την έκπτωση του. Συνενώνει τις μεταβλητές myepitheto και myonoma ώστε να βρει τον πελάτη μέσω ενός ερωτήματος που κάνει με αυτά ως κριτήρια. Αθροίζει τις ποσότητες στο quantitysum και επιλέγει, ανάλογα με την τιμή της, ποιά είναι η έκπτωση του πελάτη. Τέλος επιστρέφει την υπολογισθείσα έκπτωση.

```
def ekptosi(myepitheto,myonoma,quantity):
    client = myepitheto + ' ' + myonoma ##Συνενώνει το επίθετο και όνομα
    que = db.Query(Order)
    que = que.filter('pelatis =',client) ##Ψάχνει να βρει τις εγγραφές του πελάτη
    results = que.fetch(limit=7)
    quantitysum = 0

    for i in results:
        quantitysum = quantitysum + i.posotita
        quantitysum = quantitysum + quantity
    if quantitysum>=0 and quantitysum<=2:
        discount = 0
    elif quantitysum>=3 and quantitysum<=6:
        discount = 10
    else:
        discount = 15

    return discount
```

κώδικας 27: Συνάρτηση εύρεσης έκπτωσης

- Η συνάρτηση kodikas_par (κώδικας 26) χρησιμοποιεί την βιβλιοθήκη random της python, και συγκεκριμένα την μέθοδο choice για να επιλέξει τυχαία ανάμεσα σε αλφαριθμητικά για την δημιουργία ενός κωδικού παραγγελίας. Τέλος επιστρέφει τον κωδικό αυτό.

```
def kodikos_par():
    import random
    from random import choice
    import string
    return  ''.join(random.choice(string.ascii_uppercase + string.digits) for x in
range(6))
```

κώδικας 28: Συνάρτησης αυτόματης δημιουργίας κωδικού παραγγελίας

Η showOrder εμφανίζει με την σελίδα showorder.htm, ταξινομημένα κατά ημερομηνία δημιουργίας, τις εγγραφές. Δημιουργεί ένα ερώτημα προς τον πίνακα Order και δηλώνει την ταξινόμηση, ως προς την ιδιότητα του μοντέλου, created. Το πλήν μπροστά από το created δηλώνει την ταξινόμηση φθίνουσα. Εκτελεί το ερώτημα και περνάει τα δεδομένα μέσα στην μεταβλητή order_list. Τέλος καλείτε η συνάρτηση emfanise (κώδικας 27) ώστε να εμφανίσει με την σελίδα που ορίζετε, την showorder.htm, τις εγγραφές του πίνακα παραγγελιών.

```
class showOrder(webapp.RequestHandler):
    def get(self):
        que = db.Query(Order).order('-created');
        order_list = que.fetch(limit=orio)
        emfanise(self,'showorder.htm',{'order_list':order_list})
```

κώδικας 29: Handler εμφάνισης παραγγελιών

Η εμφάνιση στον browser γίνεται μέσω της σελίδας showorder.htm (κώδικας 28):

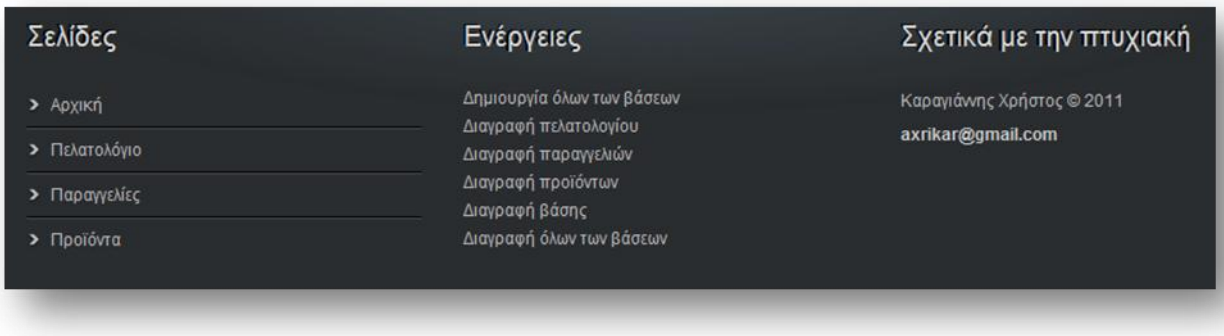
```
{% for order in order_list %}
<tr>
  <td><center>
    <form method="post" action="/infoOrder" >
      <input type="hidden" name="choice" value="{{order.key}}" />
      <input type="image" src="/static/img/order.png" alt="Submit button" />
    </form>
  </center></td>
  <td> {{ order.id_paraggelia }}</td>
  <td> {{ order.pelatis       }}</td>
  <td> {{ order.proion        }}</td>
  <td> {{ order.posotita      }}</td>
  <td> {{ order.timi          }}&#8364;</td>
  <td> {{ order.fpa           }}%</td>
  <td> {{ order.sinolo        }}&#8364;</td>
  <td> {{ order.ekptosi       }}%</td>
  <td> {{ order.sinoloekpt    }}&#8364; </td>
</tr>{% endfor %}
```

κώδικας 30: Εμφάνιση παραγγελιών - showorder.htm

Μιας και οι υπόλοιπες μέθοδοι που χρησιμοποιούνται δεν παρουσιάζουν κάποια ιδιαιτερότητα και διαφοροποίηση στα μέχρι τώρα κομμάτια κώδικα που έχουμε δει, δεν θα αναπτυχθούν περαιτέρω. Παρακάτω θα παρουσιαστούν συναρτήσεις τις οποίες δημιούργησα για την αυτόματη παραγωγή δεδομένων και αποθήκευση τους μέσα στο datastore. Αυτές οι συναρτήσεις γεμίζουν τους πίνακες της εφαρμογής μου ώστε να με βοηθήσουν κατά την παρουσίαση της.

Ορισμός των βοηθητικών συναρτήσεων παραγωγής περιεχομένου και διαγραφής των δεδομένων από το datastore

Η μέθοδος main() (κώδικας 29), που παρουσιάστηκε στην αρχή αυτού του μέρους της πτυχιακής μου, περιέχει τρείς εγγραφές οι οποίες καλούνται μέσα από την διεπαφή του χρήστη. Οι εγγραφές αυτές ενεργοποιούν τρείς συναρτήσεις, πρώτα την generateClient(), δεύτερη την generateItem() και τέλος την generateOrder(). Στο κάτω μέρος της εφαρμογής μου υπάρχουν διαθέσιμες επιλογές σχετικά με εργασίες που εκτελούνται πάνω στα δεδομένα της εφαρμογής μου.



εικόνα 20: Πρόσθετες ενέργειες της εφαρμογής

Η επιλογή "Δημιουργία όλων των βάσεων" ενεργοποιεί την GENERATEALL (κώδικας 30) η οποία εκτελεί με την σειρά τις συναρτήσεις που προαναφέρθηκαν, και στο τέλος επιστρέφει στην αρχική σελίδα της εφαρμογής.

```
def main():
    application = webapp.WSGIApplication([
        ('/addClient', addClient),
                ...
        ('/GENERATEALL', GENERATEALL),
                ...
```

```
    wsgiref.handlers.CGIHandler().run(application)
```

κώδικας 31: Δήλωση του Handler GENERATEALL

```
class GENERATEALL(webapp.RequestHandler):
    def get(self):
        generateClient()
        generateItem()
        generateOrder()
        self.redirect("/")
```

κώδικας 32: Handler δημιουργίας όλων των εγγραφών

Η πρώτη συνάρτηση που καλείται είναι η generateClient() (κώδικας 31). Αναλαμβάνει μέσα από μια λίστα από ονόματα, επίθετα, πόλεις να επιλέξει τυχαία κάποια δεδομένα. Η επιλογή αυτή γίνεται με την βιβλιοθήκη random. Διαγράφει αρχικά όλες τις εγγραφές που υπάρχουν μέσα στον πίνακα client και έπειτα δημιουργεί τέσσερις λίστες με τα προς επιλογή στοιχεία. Ένας βρόγχος for αναλαμβάνει να δημιουργήσει για εμάς 100 εγγραφές τις οποίες τις αποθηκεύει στο datastore. Μιας και τα ονόματα που θα αποθηκεύσουμε είναι γραμμένα στα ελληνικά, χρησιμοποιούμε την συνάρτηση unicode για να ορίσουμε την κωδικοποίηση των συμβολοσειρών πριν την αποθήκευση τους.

```
def generateClient():
    import random
    from random import choice
    db.delete(Client.all())
    surname = ['Αδάμου','Αλαμάνης','Αζαριάδης','Ακριτίδης','Ασλάνης',...]
    name = ['Γεώργιος','Μάριος','Μάρκος','Νίκος','Φίλιππος','Βασίλειος',...]
    address = ['Φαρσάλων','Άρτης','Αχιλλέως','Αυλίδος','Διγενή','Χάλκης',...]
    city = ['Αθήνα','Λάρισα','Τρίκαλα','Καρδίτσα','Αρτα','Κόρινθος',...]
    for i in range(100):
        surname1 = unicode(choice(surname),"utf-8")
        name1 = unicode(choice(name),"utf-8")
        address1 = unicode(choice(address),"utf-8")+" "+str(random.randint(1,250))
        city1 = unicode(choice(city),"utf-8")
        phone1 = str(random.randint(102321,789526))
        newclient = Client(onoma = name1, epitheto = surname1, dieuthinsi =
address1, poli = city1, email = surname1+"."+name1+"@mail.net", tilephono = "2410-
"+phone1)
        newclient.put();
```

κώδικας 33: Handler δημιουργίας εγγραφών πελάτη

Μέσα στην unicode η choice αναλαμβάνει να επιλέξει για εμάς, μία οποιαδήποτε τιμή από τις λίστες που της ορίζονται. Η unicode παίρνει ώς ορίσματα δύο τιμές. Η πρώτη είναι η συμβολοσειρά που θέλουμε να μετατρέψουμε και η δεύτερη τον τύπο της κωδικοποιήσης που θέλουμε να χρησιμοποιήσει. Για το τηλέφωνο ορίζουμε ένα σύνολο ανάμεσα στο οποίο θα επιλέξει ένα εξα-ψήφιο αριθμό και τον οποίο θα καταχωρήσουμε ώς τηλέφωνο πελάτη. Τέλος όλες αυτές τις μεταβλητές τις περνάμε σε ένα αντικείμενο, το newclient και το αποθηκεύουμε στο datastore.

Η δεύτερη συνάρτηση που καλείτε είναι η generateItem (κώδικας 32). Όπως και στην προηγούμενη συνάρτηση, έτσι και εδώ χρησιμοποιημούμε λίστες γεμάτες με τιμές από τις οποίες θα γίνει η επιλογή, για την μάρκα, το μοντέλο και τον τύπο του προϊόντος. Με την βοήθεια ενός βρόγχου for, και εδώ, δημιουργούμε 100 εγγραφές τις οποίες και αποθηκεύουμε στον πίνακα Item.

```
def generateItem():
    import random
    from random import choice
    import string
    db.delete(Item.all())
    brands = ["Microsoft","Apple","Ibm","Logitech",....]
    chars = string.ascii_uppercase + string.digits
    type1 = ["Οθόνη","Ποντίκι","Πληκτρολόγιο","Εκτυπωτής",.....]

    for i in range(100):
        brands1 = choice(brands)
        model1 = ''.join([random.choice(chars) for x in range(4)])+ '-' +
                 ''.join([random.choice(chars) for x in range(2)])
        type11 = unicode(choice(type1),"utf-8")
        price1 = random.uniform(1,1150)
        sin_tem = random.randint(1,30)
        newitem = Item(marka = brands1, modelo = model1, tipos = type11,
                       timi = round(price1,2), temaxia = sin_tem)
        newitem.put();
```

κώδικας 34: Handler δημιουργίας εγγραφών αντικειμένου

Σε αυτή την συνάρτηση εκτός της random, φορτώνουμε και την string η οποία θα μας βοηθήσει να φτιάξουμε μεταβλητή η οποία θα περιλαμβάνει κεφαλαία λατινικά γράμματα και αριθμούς. Τέλος οι επιλεγείσες τιμές αποθηκεύονται μέσα στο αντικείμενο newitem και αυτό με την σειρά του μέσα στο datastore.

Τελευταία συνάρτηση είναι η generateOrder (κώδικας 33). Σε αυτή την συνάρτηση εκτός από την αυτόματη επιλογή και δημιουργία τιμών, ανακαλεί αντικείμενα μέσα από το datastore, για να πάρει στοιχεία από τους πίνακες client και item.

```
def generateOrder():
    import random
    from random import choice
    que = db.Query(Client)
    results1 = que.fetch(limit=100)

    que = db.Query(Item)
    results2 = que.fetch(limit=100)

    ep_forou = [6,9,12,16,20,23,25]

    for i in range(100):
        flag1 = random.randint(0,99)##flag gia kathe egrafi
```

```
        flag2 = random.randint(0,99)
        ant1 = kodikos_par()
        ant2 = results1[flag1].epitheto + ' ' + results1[flag1].onoma##Pairnei ena
onoma apo tin Basi Client
        ant3 = results2[flag2].marka + ' ' + results2[flag2].modelo##Pairnei ena
proion apo tin basi Item
        ant4 = random.randint(1,6)##posotita
        ant5 = results2[flag2].timi##timi
        ep = choice(ep_forou)
        antf5 = ant5 + ant5*ep/100##Ipologismos tou forou gia kathe proion
        ant6 = ant4 * antf5##Ipologismos tou sinolou ton proionton me foro gia to
kathena
        discount = ekptosi(results1[flag1].epitheto,results1[flag1].onoma,ant4)
        ant7 = ant6 - ant6*discount/100.0##Ipologismos tis ekptosis gia tin
sinoliki timi ton proionton
        neworder = Order(id_paraggelia = ant1, pelatis = ant2, proion = ant3,
posotita = ant4, timi = ant5, fpa = ep, sinolo = round(ant6,2), ekptosi = discount,
sinoloekpt = round(ant7,2))
        neworder.put();

        ent1 = results1[flag1].key()##to kleidi tou pelati
        ent2 = results2[flag2].key()##to kleidi tou proiontos

        newbasi = Basi(id_paraggelia = ant1, id_pelatis = str(ent1),
                       id_item = str(ent2))
        newbasi.put();
```

κώδικας 35: Handler δημιουργίας εγγραφών παραγγελιών

Αρχικά διαβάζει όλες τις εγγραφές από τους πίνακες client και item και τις περνάει στις μεταβλητές results1, results2 αντίστοιχα. Δημιουργεί μια λίστα την ep_forou με τις δυνατές τιμές για το πεδίο του φπα. Μέσα στο for loop τα flag1 και flag2 είναι μεταβλητές επιλογής, οι οποίες σε κάθε επανάληψη του βρόγχου παίρνουν και διαφορετικές τιμές ανάμεσα στο 0 και 99. Τα δύο αυτά flags θα είναι υπεύθυνα για την τυχαία επιλογή εγγραφών από τους πίνακες client και item. Στη συνέχεια χρησιμοποιείτε η συνάρτηση kodikos_par(), που είδαμε νωρίτερα, ώστε να γίνει αυτόματη παραγωγή κωδικών παραγγελίας για τις καινούργιες εγγραφές μας.

Στην μεταβλητή ant2 επιλέγεται, από μια τυχαία επιλογή των εγγραφών των πελατών, το όνομα και το επίθετο του πελάτη. Στην ant3 περνάνε οι πληροφορίες μάρκας και μοντέλου από την τυχαία επιλογή του προϊόντος. Οι ποσότητες των παραγγελιών δημιουργούνται τυχαία έπειτα στην μεταβλητή ant4, ενώ η ant5 περιέχει την τιμή του προϊόντος που επιλέχθηκε νωρίτερα. Έπειτα γίνεται ο υπολογισμός των τιμών που χρειάζεται να αποθηκευτούν μέσα σε μια εγγραφή Order και τέλος αποθηκεύεται το αντικείμενο που δημιουργήσαμε με τις προηγούμενες τιμές. Μιας και θα πρέπει να υπάρχει και αντιστοιχεί εγγραφή στον πίνακα Basi, παίρνουμε τα κλειδιά των επιλεγμένων εγγραφών για τον πελάτη και προϊόν και τα αποθηκεύουμε και αυτά στον πίνακα Basi.

Με την ολοκλήρωση και αυτού του βρόγχου η εκτέλεση επιστρέφει στον χειριστή GENERATEALL ο οποίος καλεί την redirect για να μας εμφανίσει την αρχική σελίδα και να τελειώσει την εκτέλεση του.

Τέλος μέσα στην main() (κώδικας 34) συνάρτηση έχουμε ορίσει ακόμα πέντε χειριστές οι οποίοι αναλαμβάνουν να διαγράψουν τα δεδομένα μας από το datastore. Μπορούμε με αυτούς τους χειριστές να διαγράψουμε τον κάθε πίνακα ξεχωριστά ή και όλους μαζί. Στο κάτω τμήμα της σελίδας της εφαρμογής μου δίνεται η πρόσβαση σε αυτούς τους χειριστές. Το τμήμα της main() που περιέχει αυτούς τους χειριστές φαίνεται παρακάτω:

```
def main():
    application = webapp.WSGIApplication([
        ('/addClient', addClient),
                ...
        ('/DELETECLIENT', DELETECLIENT),
        ('/DELETEORDER', DELETEORDER),
        ('/DELETEITEM', DELETEITEM),
        ('/DELETEBASI', DELETEBASI),
        ('/DELETEALL', DELETEALL),
        ('/.*', MainHandler)],
                                            debug=True)
    wsgiref.handlers.CGIHandler().run(application)
```

κώδικας 36: Δήλωση των Handler διαγραφής

Ο κώδικας των χειριστών παρουσιάζεται παρακάτω (κώδικας 35):

```
class DELETECLIENT(webapp.RequestHandler):
    def get(self):
        db.delete(Client.all())
        emfanise(self,'index.htm',{})
class DELETEORDER(webapp.RequestHandler):
    def get(self):
        db.delete(Order.all())
        db.delete(Basi.all())
        emfanise(self,'index.htm',{})
class DELETEITEM(webapp.RequestHandler):
    def get(self):
        db.delete(Item.all())
        emfanise(self,'index.htm',{})
class DELETEBASI(webapp.RequestHandler):
    def get(self):
        db.delete(Basi.all())
        emfanise(self,'index.htm',{})
class DELETEALL(webapp.RequestHandler):
    def get(self):
        db.delete(Client.all())
        db.delete(Order.all())
        db.delete(Item.all())
        db.delete(Basi.all())
        self.redirect("/")
```

κώδικας 37: Handler για την διαγραφή πινάκων

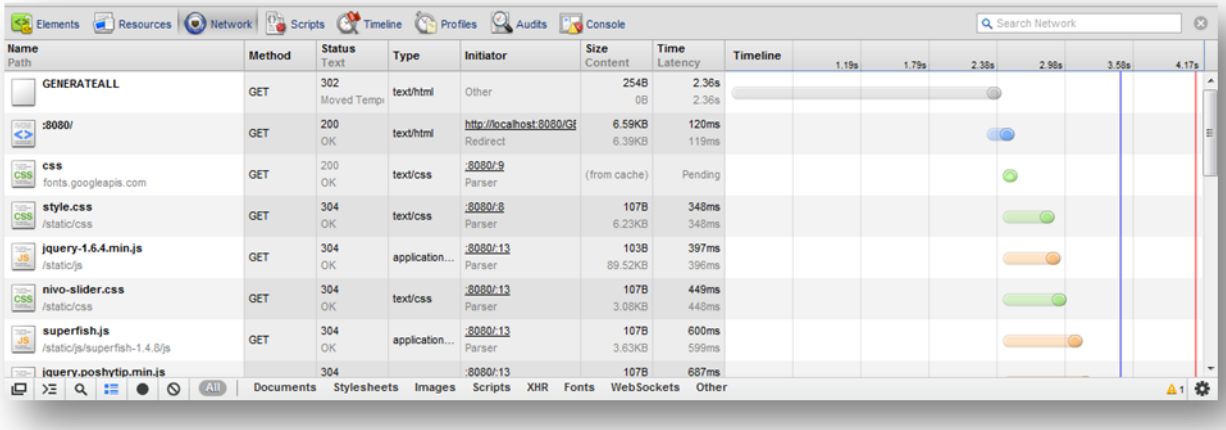
Σε κάθε έναν από αυτούς τους χειριστές, και ανάλογα με το ποιόν ή ποιους πίνακες θέλουμε να διαγράψουμε, χρησιμοποιούμε την db.delete περνώντας ως όρισμα το όνομα του πίνακα όπου πάνω σε αυτόν ενεργεί η συνάρτηση .all(). Διαγράφονται με αυτό τον τρόπο όλα τα δεδομένα που έχουν αποθηκευμένα μέσα.

Χρόνοι δημιουργίας εγγραφών σε διάφορα περιβάλλοντα

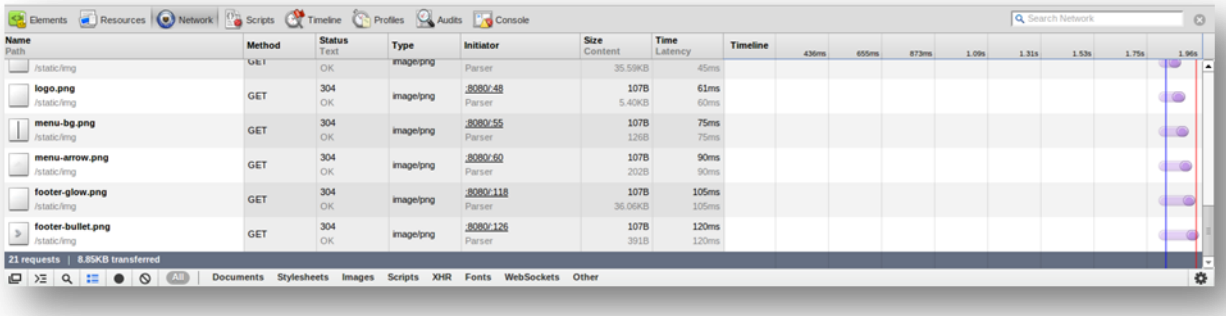
Στις παρακάτω εικόνες παρουσιάζονται οι χρόνοι που χρειάζεται ο development server και η εφαρμογή στο app engine για να δημιουργήσουν το περιεχόμενο με την GENERATEALL. Οι χρόνοι καταγράφηκαν με την βοήθεια του Google Chrome με την επιλογή που έχει για development tools. Εμφανίζει πολλές πληροφορίες για το κάθε αρχείο που στάλθηκε πίσω στον browser από το app engine, τον τύπο του το μέγεθος καθώς και τον χρόνο που χρειάστηκε. Επίσης εμφανίζεται ένα χρονοδιάγραμμα με τους χρόνους που απαιτήθηκαν για την αποστολή του html κώδικα, των css, javascript.

Για τον development server ο χρόνος που απαιτήθηκε (με απενεργοποιημένα antivirus, firewall και σε περιβάλλον windows 7) ήταν 4.17 δευτερόλεπτα (7s με ενεργοποιημένα). Σε περιβάλλον linux απαιτήθηκαν σχεδόν 2 δευτερόλεπτα (debian-derived διανομή με Gnome 3 και default services). Ίσως να πετυχαίναμε και ακόμα καλύτερους χρόνους με κάποια stripped-down διανομή όπως Slackware, gentoo ή και arch χωρίς γραφικό περιβάλλον και πρόσθετες διεργασίες.

Μέσα σε αυτό τον χρόνο στο datastore έγιναν 400 εγγραφές στους 4 πίνακες της εφαρμογής καθώς και πολλαπλάσιες αναγνώσεις μιας και για κάθε εγγραφή, επικυρώνονταν ότι τα στοιχεία αυτά δεν υπήρχαν ήδη μέσα στους πίνακες. Επίσης πολλαπλάσιες πράξεις έγιναν από τον κώδικα Python για την επιλογή στοιχείων από τις λίστες, υπολογισμού της τιμής της κάθε παραγγελίας, έκπτωσης για τον κάθε πελάτη και για κάθε παραγγελία ξεχωριστά.

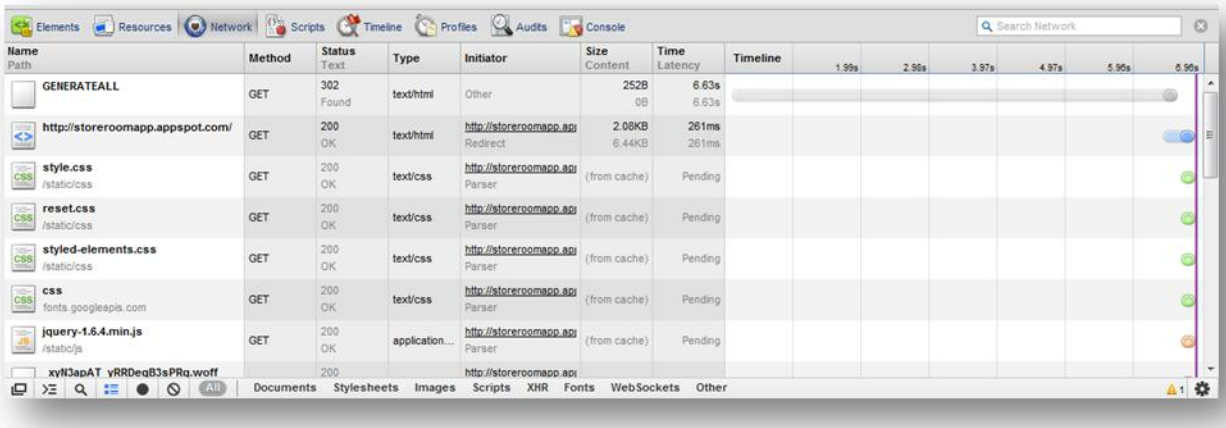


εικόνα 21: Χρόνος δημιουργίας όλων των εγγραφών στον development server σε windows



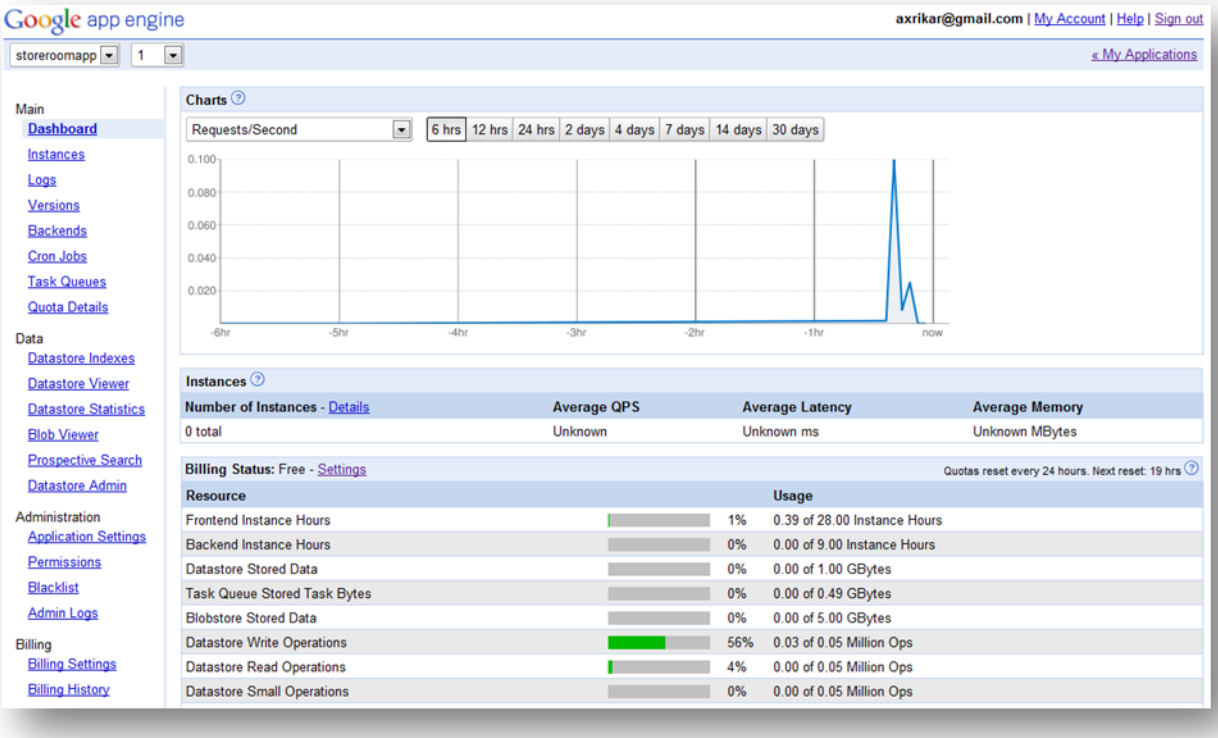
εικόνα 22: Χρόνος δημιουργίας όλων των εγγραφών στον development server σε linux

Ο χρόνος που χρειάστηκε το app engine για την διεκπεραίωση όλων των παραπάνω είναι σχεδόν 7 δευτερόλεπτα.



εικόνα 23: Χρόνος δημιουργίας όλων των εγγραφών στο App engine

Στην εικόνα 24 εμφανίζεται και το γράφημα της εφαρμογής μου για δύο εκτελέσεις διαγραφής και δημιουργίας όλων των εγγραφών στο Datastore. Μιας και οι εγγραφές είναι πιο ακριβές (ως προς την επεξεργασία, αποθήκευση, διαγραφή) έχουν χρησιμοποιήσει περισσότερους πόρους για την διεκπεραίωση τους. 0.03 από τις 0.05 εκατομμύρια λειτουργίες (operations) έχουν χρησιμοποιηθεί από την αποθήκευση περίπου 800 εγγραφών σε αντικείμενα στο datastore της εφαρμογής.



εικόνα 24: Γράφημα εφαρμογής για την GENERATEALL

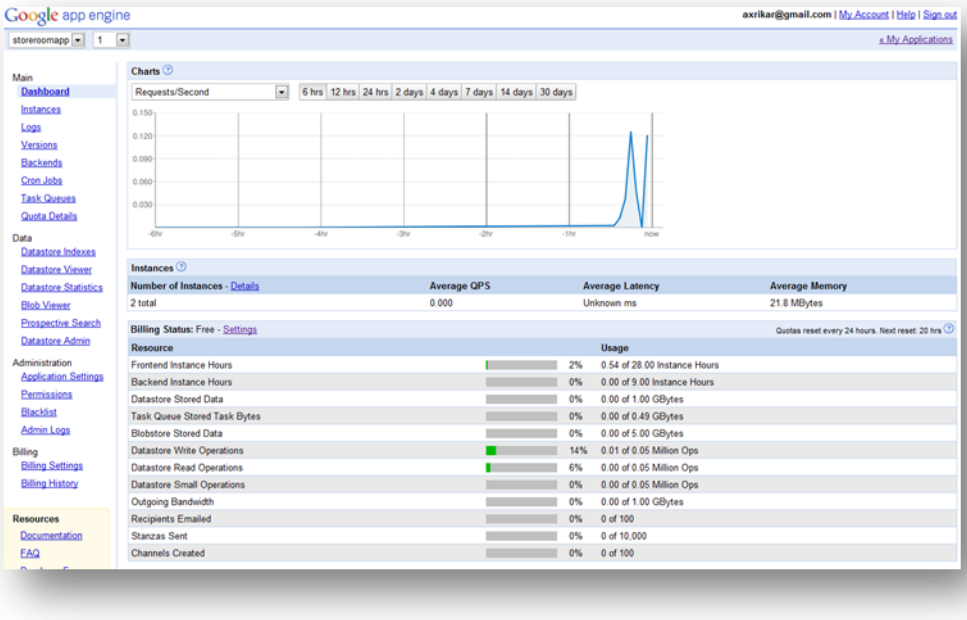
Μέρος ΙΙΙ

Διαχείρηση εφαρμογής μέσα από την σελίδα διαχείρησης

Για την διαχείριση της εφαρμογής η Google παρέχει και ένα web περιβάλλον. Η πρόσβαση σε αυτό γίνεται, με τα στοιχεία σύνδεσης του διαχειριστή στο google accounts και επιλέγοντας έπειτα την εφαρμογή στο app engine που δημιουργήσαμε. Μεταφερόμαστε με αυτό τον τρόπο στην διεπαφή όπου μας επιτρέπει να επιλέξουμε διάφορες λειτουργίες που έχουν να κάνουν με τον τρόπο που προσφέρεται η εφαρμογή μας. Μπορούμε να συλλέξουμε στοιχεία σχετικά με το πόσο χρόνο έτρεξε η εφαρμογή μας πάνω στο app engine, πόσους

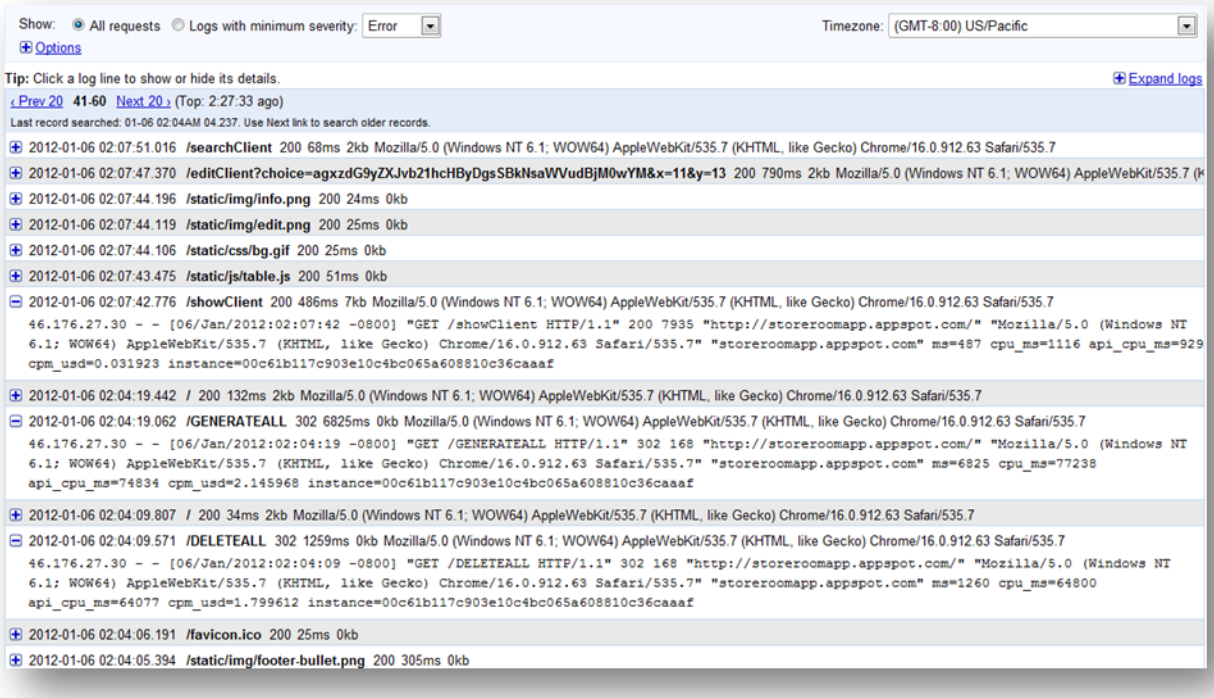
πόρους δικτυακούς χρησιμοποίησε, πόση επεξεργαστική ισχύ χρειάστηκε για την διεκπεραίωση των εργασιών της.

Μέσα από αυτό το περιβάλλον ο διαχειριστής μπορεί να παρακολουθήσει ξεχωριστά πόσα αιτήματα έγιναν για την κάθε σελίδα της εφαρμογής καθώς και πόσος χρόνος απαιτήθηκε από τους χειριστές των αιτημάτων για την διεκπεραίωση τους. Ενημερώνεται για το αν κάποιος από τους χειριστές χρησιμοποιεί πολλούς πόρους ώστε να τους διαμορφώσει καλύτερα για γρηγορότερη επεξεργασία. Επίσης παρακολουθεί τα σφάλματα που μπορεί να συμβούν. Το περιβάλλον εμφανίζει ένα γράφημα παρακολούθησης της εφαρμογής, το πόσες οντότητες της εφαρμογής εκτελούνται σε κάθε χρονικό διάστημα, καθώς και το ποσοστό της χρησιμοποίησης των πόρων που έχει στην διάθεση της η κάθε εφαρμογή.



εικόνα 25: Εμφάνιση πληροφοριών μέσα από το web interface της εφαρμογής

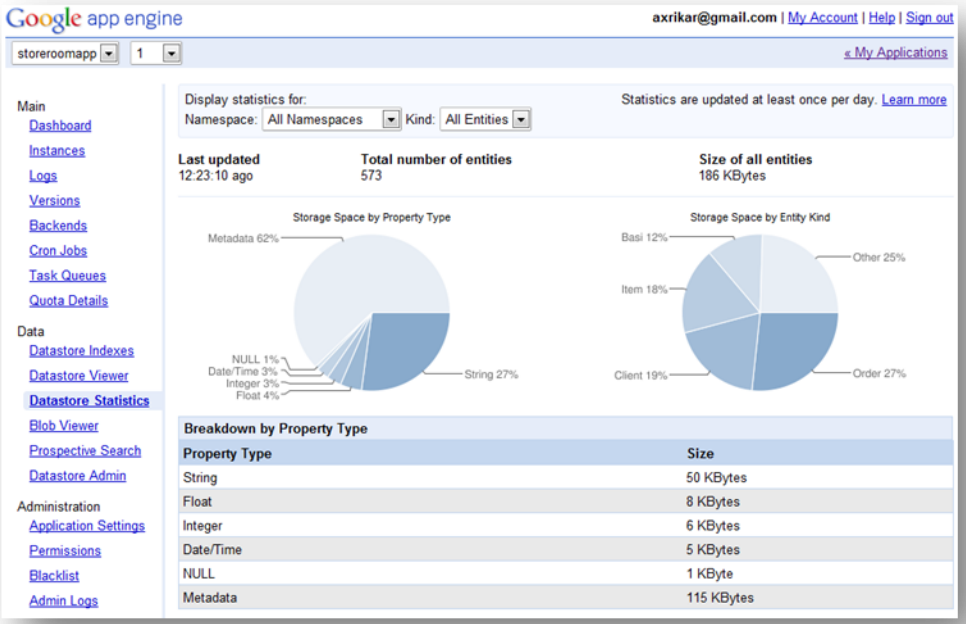
Σημαντική δυνατότητα είναι η πρόσβαση στα αρχεία καταγραφής της εφαρμογής. Το app engine δίνει την δυνατότητα στον προγραμματιστή να εισάγει μηνύματα καταγραφής όπου κρίνει αυτός απαραίτητο μέσα στον κώδικα του. Αυτά τα μηνύματα συλλέγονται και παρουσιάζονται στον διαχειριστή μέσω της web διεπαφής. Πληροφορίες όπως η είσοδος χρηστών, αιτήματα που έκαναν, δεδομένα που ανέκτησαν, αποθήκευσαν ή διέγραψαν, όλα αυτά είναι δυνατό να διαβαστούν από τον διαχειριστή της εφαρμογής.



εικόνα 26: Αρχεία καταγραφής πληροφοριών

Το app engine παρέχει ορισμένους πόρους του προς την εφαρμογή δωρεάν. Μόλις συμπληρωθούν αυτοί οι πόροι θα πρέπει ο διαχειριστής να καταβάλει κάποιο αντίτιμο για την αγορά περισσοτέρων πόρων ώστε να συνεχίσει την εύρυθμη λειτουργία της η εφαρμογή. Μέσα από το web περιβάλλον δίνεται η δυνατότητα να οριστεί το μέγιστο χρηματικό ποσό που μπορεί να διαθέσει ο διαχειριστής προς την εφαρμογή. Μπορούν να εξασφαλιστούν περισσότεροι πόροι ως προς την διαδικτυακή κίνηση, την επεξεργαστική ισχύ, email που μπορούν να σταλούν.

Επίσης παρουσιάζονται αναλύσεις σχετικά με το datastore

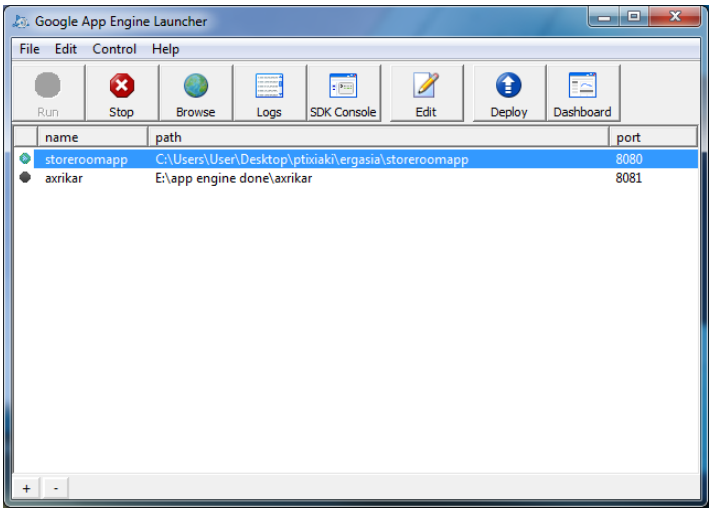


εικόνα 27: Αναλυτικά δεδομένα για την εφαρμογή

Εγκατάσταση περιβάλλοντος ανάπτυξης

- Το app engine απαιτεί την ύπαρξη της γλώσσας python στον υπολογιστή για την λειτουργία του. Σε περιβάλλοντα όπως το linux συνήθως είναι προεγκατεστημένη. Μέχρι στιγμής υποστηρίζεται η python 2.5 και python 2.7. Η εγκατάσταση γίνεται με κατέβασμα της γλώσσας από το http://python.org/download/.

  Για να ελέγξουμε αν η python είναι ήδη εγκατεστημένη μέσα στον υπολογιστή μας σε command prompt εκτελούμε την εντολή python. Αυτό έχει ως συνέπεια την εισαγωγή μας στο shell της python. Μας εμφανίζετε η έκδοση της python και διάφορες άλλες πληροφορίες ανάλογα με την έκδοση.
- Το ίδιο το περιβάλλον του app engine μπορούμε να το κατεβάσουμε από το http://code.google.com/intl/el-GR/appengine/downloads.html. Μέσα σε αυτό πακέτο εκτός των framework και των υπηρεσιών που έχει ετοιμάσει η google παρέχεται και ένας developement server. Στο περιβάλλον των windows και mac os δίνεται η δυνατότητα μέσα από ένα γραφικό περιβάλλον, να ενεργοποιηθεί ο server αυτός. Επίσης ο προγραμματιστής μπορεί να εκτελέσει διάφορες διεργασίες όπως, το ανέβασμα της εφαρμογής στο app engine, την παρακολούθηση του αρχείο πεπραγμένων, την μεταφορά στο local administration page.



εικόνα 28: Περιβάλλον sdk

Με την ενεργοποίηση του server ο προγραμματιστής μπορεί να αποκτήσει πρόσβαση στην εφαρμογή του από έναν browser στην σελίδα http://localhost:8080 . Η 8080 είναι η πόρτα που δίνει το sdk στην εφαρμογή μας. Για κάθε μία εφαρμογή παρέχεται και διαφορετικό port. Για περιβάλλοντα linux η ενεργοποίηση και διαχείριση του server γίνεται μέσα από το command prompt.

Βιβλιογραφία

1. **Lutz, Mark.** *Learning Python, 4th edition.* s.l. : O'Reilly Media, 2009.

2. **Barry, Paul.** *Head First Python.* s.l. : O'Reilly Media, 2010.

3. **Severance, Charles.** *Using Google App Engine.* s.l. : O'Reilly Media, 2009.

4. **Sanderson, Dan.** *Programming Google App Engine.* s.l. : O'Reilly Media, 2009.

5. **Maren, Alianna J.** *Cloud Computing with Python.* 2010.

6. **Alchin, Marty.** *Pro Django.* s.l. : Apress, 2008.

7. **http://en.wikipedia.org/wiki/Cloud_computing.**

8. **http://code.google.com/intl/el-GR/appengine/docs/.**

9. Developing and deploying an application on Google App Engine. *http://www.youtube.com/watch?v=bfgO-LXGpTM.* [Online]

10. Campfire One: Introducing Google App Engine (pt. 1). *http://www.youtube.com/watch?v=3Ztr-HhWX1c.* [Online]

11. Bigtable: A Distributed Storage System for Structured Data. *http://research.google.com/archive/bigtable.html.* [Online]

12. App scale. *http://appscale.cs.ucsb.edu/.* [Online]

13. http://code.google.com/intl/el-GR/appengine/docs/. [Online]