Πανεπιστήμιο Πειραιώς

Μεταπτυχιακό στην "Διδακτική της Τεχνολογίας και Ψηφιακών Συστημάτων"

Κατεύθυνση: Δικτυοκεντρικά Συστήματα

ΝΤΟΛΙΑΣ ΝΙΚΟΛΑΟΣ

ΜΕ-0687

Επίδειξη και Χρήση Νέων Τεχνολογιών Βασισμένων στην J2EE Αρχιτεκτονική για την Υλοποίηση Επεκτάσιμων Web Εφαρμογών και Web Services

Επιβλέπων: Λέκτορας Βέρα-Αλεξάνδρα Σταυρουλάκη

01/09/2009

University Of Piraeus

Postgraduate Degree in "Technology Education and Digital Systems"

Area of Study: Net Centric Systems

NTOLIAS NIKOLAOS

ME-0687

Use and Presentation of New Technologies Based on J2EE Architecture for Implementing Scalable Web Applications and Web Services

Superviser: Lecturer Vera-Alexandra Stavroulaki

01/09/2009

Table of Contents

Table of Figures

Περίληψη

Σήμερα η ευρεία εξάπλωση της χρήσης του παγκοσμίου ιστού για προσωπικούς και επαγγελματικούς λόγους έχει σαν αποτέλεσμα να αυξηθούν οι απαιτήσεις από τις διαδικτυακές εφαρμογές σε μεγάλο βαθμό, αυτή η κατάσταση έχει δημιουργήσει έναν άτυπο αγώνα στο ποιος μπορεί να κατασκευάσει την πληρέστερη εφαρμογή για κάθε επιχειρηματικό και όχι μόνο αντικείμενο. Το αποτέλεσμα της παραπάνω κατάστασης βοήθησε στην εξέλιξη της αρχιτεκτονικής των εφαρμογών αλλά επίσης δημιούργησε και το πρόβλημα της μεγάλης πολυπλοκότητας και της δυσκολίας συντήρησης αυτών. Χαρακτηριστικά όπως επεκτασιμότητα, διαθεσιμότητα, εύκολη συντήρηση και χαμηλοί χρόνοι απόκρισης στα διάφορα αιτήματα των πελατών είναι απαραίτητα για την δημιουργία μιας επιτυχημένης διαδικτυακής εφαρμογής.

Στην παρούσα εργασία θα γίνει μία προσπάθεια για την κατασκευή μιας εφαρμογής που θα διαθέτει τα παραπάνω χαρακτηριστικά και παράλληλα θα στοχεύει στο να διατηρήσει την πολυπλοκότητα στο ελάχιστο δυνατό επίπεδο. Για να επιτευχθεί κάτι τέτοιο πρέπει να γίνουν οι κατάλληλες επιλογές σε επίπεδο αρχιτεκτονικής και επιλογής τεχνολογιών που θα χρησιμοποιηθούν. Σε επόμενο στάδιο πρέπει να παρθούν οι κατάλληλες σχεδιαστικές αποφάσεις που θα εκμεταλλεύονται την χρήση αλλά και την συνεργασία μεταξύ των νέων αυτών τεχνολογιών.

Σε επίπεδο αρχιτεκτονικής η επιλογή στην συγκεκριμένη εργασία είναι το MVC (Model View Controller) μοντέλο το οποίο είναι έτσι δομημένο ώστε να προωθεί μια πολυ-στρωματική σχεδίαση της εφαρμογής κατά την οποία κάθε στρώμα είναι επιφορτισμένο με το να προσφέρει μια συγκεκριμένη ομάδα παρόμοιων υπηρεσιών. Αυτή η αρχιτεκτονική διαχωρίζει τις ευθύνες για την περάτωση ενός αιτήματος του χρήστη σε επίπεδα, έχοντας σαν αποτέλεσμα την καλύτερη οργάνωση του κώδικα και την ελάχιστη εξάρτηση μεταξύ των επιπέδων πράγμα που διευκολύνει τις αλλαγές που μπορεί να χρειαστούν στον κώδικα ή την προσθήκη νέων υπηρεσιών.

Σε επίπεδο τεχνολογιών η βάση η οποία χρησιμοποιήθηκε, πάνω στην οποία ενσωματώθηκαν και άλλες τεχνολογίες, είναι το Spring Framework. Η επιλογή της συγκεκριμένης τεχνολογίας, που χρησιμοποιείτε στην κατασκευή διαδικτυακών εφαρμογών σε Java βασισμένη στην J2EE πλατφόρμα, έγινε για πολλούς λόγους όπως ότι βοηθάει στην μείωση της πολυπλοκότητας τέτοιων εφαρμογών, αποτελεί ένα συνολικό και ολοκληρωμένο πλαίσιο με νέες ιδέες και τεχνολογίες το οποίο έχει την δυνατότητα να συνεργάζεται με άλλες υπάρχουσες τεχνολογίες σε κάθε επίπεδο της MVC αρχιτεκτονικής. Αυτό δίνει τη δυνατότητα επιλογής της βέλτιστης για την κάθε

περίπτωση τεχνολογίας που απαιτείται για το κάθε επίπεδο του MVC μοντέλου, παρέχοντας έναν μεγάλο αριθμό συνδυασμών και δυνατοτήτων και ιδιαίτερη ευελιξία.

Η εφαρμογή η οποία αναπτύχθηκε ώστε να επιδείξει τα πλεονεκτήματα και το τι προσφέρουν οι τεχνικές επιλογές που έγιναν παραπάνω ονομάζεται Show Face. Το Show Face είναι μία απλή διαδικτυακή εφαρμογή που προσφέρει κάποιες υπηρεσίες κοινωνικής δικτύωσης μέσα από ένα εύχρηστο και ευχάριστο περιβάλλον για τον χρήστη. Ο εξυπηρετητής που χρησιμοποιήθηκε για να φιλοξενήσει την εφαρμογή είναι ο Apache Tomcat 6 ο οποίος δεν έχει μεγάλες απαιτήσεις σε σχέση με άλλους εξυπηρετητές εφαρμογών και με τη βοήθεια του Spring Framework μπορεί να επιτελέσει λειτουργίες που παλιότερα απαιτούσαν την χρήση εξυπηρετητών με μεγάλες απαιτήσεις, κόστος και πολυπλοκότητα. Η βάση δεδομένων που χρησιμοποιήθηκε για την εφαρμογή είναι η HSQL DB η οποία είναι μια «ελαφριά» βάση που εκτελείτε στην μνήμη του υπολογιστή και για αυτόν το λόγο είναι ιδιαίτερα γρήγορη, βέβαια υπάρχει περιορισμός στον όγκο δεδομένων που μπορεί να διαχειριστεί.

Όπως αναφέρθηκε παραπάνω η εφαρμογή παρέχει κάποιες απλές υπηρεσίες κοινωνικής δικτύωσης. Ποιο συγκεκριμένα ένας χρήστης αφού δημιουργήσει έναν λογαριασμό μπορεί να ανεβάζει τις φωτογραφίες του στην εφαρμογή και να τις βλέπει σε thumbnails ή στο αρχικό τους μέγεθος, επίσης μπορεί να διαγράψει από το σύστημα όποιες θέλει. Επίσης μπορεί να δει τους υπάρχοντες χρήστες της εφαρμογής και να τους στείλει ένα αίτημα φιλίας όπως και άλλοι μπορούν να στείλουν στον ίδιο. Κάποιος χρήστης μπορεί να δεχτεί ή να αρνηθεί ένα αίτημα φιλίας. Σε περίπτωση που το αποδεχτεί και 2 χρήστες γίνονται φίλοι μεταξύ τους τότε μπορεί ο ένας να βλέπει τις φωτογραφίες του άλλου καθώς και κάποια παραπάνω στοιχεία για αυτόν. Τέλος κάποιος χρήστης μπορεί να διαγράψει έναν άλλο από φίλο του, αυτό θα έχει ως αποτέλεσμα να μην μπορούν να βλέπουν τις φωτογραφίες ο ένας του άλλου.

Μετά την παραπάνω σύντομη περιγραφή της εφαρμογής, θα γίνει μια αναφορά στις τεχνολογίες που ενσωματώθηκαν στο Spring Framework για κάθε επίπεδο του MVC μοντέλου. Στο Model επίπεδο χρησιμοποιήθηκαν POJOs για τον ορισμό του μοντέλου τα οποία διαχειρίζεται το Spring Container όσον αφορά το transaction management το οποίο γίνετε με τέτοιο τρόπο ώστε να είναι συγκεντρωμένο σε ένα σημείο με όλα τα πλεονεκτήματα που αυτό συνεπάγεται (ευκολία αλλαγών, καθαρός κώδικας). Η τεχνολογία που επιλέχθηκε για την διαχείριση των δεδομένων με την βάση είναι το Hibernate το οποίο συνεργάζεται με το Spring ιδιαίτερα εύκολα και αποτελεσματικά. Στο επίπεδο View χρησιμοποιήθηκαν τα γνωστά html, css και jsps τα οποία σε συνδυασμό με εξωτερικές βιβλιοθήκες του Spring και την βιβλιοθήκη Displaytag κάνουν πολύ ευκολότερη την διασύνδεση με την πλευρά του εξυπηρετητή ώστε η πληροφορίες να εμφανίζονται στον χρήστη κατάλληλα μορφοποιημένες και χωρίς να χρειάζονται πολλές γραμμές κώδικα που αυξάνουν την πολυπλοκότητα. Τέλος στο Controller επίπεδο

χρησιμοποιήθηκε το Spring MVC το οποίο είναι ενσωματωμένο στο Spring Framework και συνεργάζονται ιδιαίτερα καλά όπως είναι αναμενόμενο. Επίσης σε αυτό το επίπεδο χρησιμοποιήθηκαν σε συνδυασμό με το Spring MVC και τα Apache Tiles τα οποία μορφοποιούν με εύκολο τρόπο την οργάνωση της τελικής σελίδας που βλέπει ο χρήστης όσον αφορά το περιεχόμενό της, δίνοντας και πάλι την δυνατότητα εύκολων αλλαγών λόγω της συγκέντρωσης αυτής της πληροφορίας σε ένα σημείο (separation of concerns).

Όλες οι παραπάνω επιλογές γίνανε ώστε να μειωθεί όσο είναι δυνατό η πολυπλοκότητα της εφαρμογής καθώς και να είναι εύκολη η συντήρηση της και η προσθήκη νέων λειτουργιών. Όλα αυτά καθώς και ο σχεδιασμός των πακέτων του κώδικα δίνουν συγκεκριμένες ευθύνες σε κάθε σημείο του κώδικα και των αρχείων παραμετροποίησης έτσι ώστε η συντήρηση, η επεκτασιμότητα και οι πιθανές μελλοντικές αλλαγές/προσθήκες να απαιτούν λιγότερο κόπο. Το επόμενο σημαντικό σημείο είναι να μετρηθεί ο χρόνος απόκρισης της εφαρμογής στα πιθανά αιτήματα ενός χρήστη αλλά και το φορτίο που μπορεί να αντέξει όταν εξυπηρετεί πολλούς χρήστες ταυτόχρονα. Αυτό είναι ιδιαίτερα σημαντικό γιατί μία εφαρμογή όσο καλά οργανωμένη και αν είναι αν αργεί να απαντήσει τα αιτήματα των χρηστών και δεν μπορεί να εξυπηρετήσει μεγάλο αριθμό αυτών, είναι μη αποτελεσματική ως προς το σκοπό της αλλά και δύσχρηστη.

Για την μέτρησης της απόδοσης της εφαρμογής χρησιμοποιήθηκε το Jmeter το οποίο είναι ένα εργαλείο που μπορεί να προσομοιώσει αιτήματα χρηστών προς την εφαρμογή. Για να γίνει αυτό κατασκευάστηκαν διάφορα σενάρια χρήσης της εφαρμογής στο Jmeter στα οποία γίνονται αιτήσεις στις βασικές λειτουργίες της εφαρμογής από έναν παραμετροποιήσιμο αριθμό ταυτόχρονων χρηστών μέσα σε κάποιο ορισμένο διάστημα. Στα διάφορα σενάρια που εκτελέστηκαν με φυσιολογικό έως πολύ μεγάλο και αφύσικο φορτίο τα στατιστικά αποτέλεσμα και τα γραφήματα που παρήχθησαν δείχνουν το πόσο καλά συμπεριφέρεται η εφαρμογή και προσαρμόζεται στην εξυπηρέτηση μεγαλύτερων φορτίων (scalability). Τέλος με την χρήση και ενός Profiling εργαλείου καταγράφηκε η χρήση των πόρων του συστήματος σε περιόδους έντονου φόρτου (που παρήγαγε το Jmeter) όπου και εκεί τα αποτελέσματα δείξανε τη σωστή και ομαλή διαχείριση μνήμης και επεξεργαστικής ισχύς που έκανε η εφαρμογή.

Εν κατακλείδι ο όλος σχεδιασμός και οι επιλογές που γίνανε για την υλοποίηση του Show Face δίνουν όντως προστιθέμενη αξία στην εφαρμογή καθώς ικανοποιούν αποδεδειγμένα τα σημαντικά αυτά χαρακτηριστικά που θα πρέπει να έχει μια σύγχρονη διαδικτυακή εφαρμογή υψηλών απαιτήσεων. Τέλος γίνετε και μια αναφορά σε προσθήκες/βελτιώσεις που θα μπορούσαν να γίνουν σαν μελλοντική εργασία.

1. Introduction

In this introductory part the objective of this thesis will be presented. The problem that this thesis is trying to solve will be explained as well as the state of the art in similar cases in real world applications. Finally in this chapter the structure of this report is provided.

1.1. Complexity and Scalability Issues of Large Scale Web Applications

Nowadays it is quite obvious that the requirements and features of web applications have increased to a great degree given the wide spread adaptation and use of the World Wide Web in most people's everyday life for personal (education, entertainment etc) and professional activities. This trend but also necessity has created a race in terms of who will build the most complete web application to support the user's needs and provide satisfaction to the best possible level given today's technologies.

This situation has lead to the development and progress of software engineering and distributed computing to a great degree but also has created an important problem that has to be solved. The problem is that today's requirements have raised the complexity and multiplied the challenges of building and maintaining a large scale web application.

Concepts and attributes of such an application include availability, scalability, performance, maintainability, ease of use. To be more precise such applications are required to have a minimum downtime per year which means that their services should be available almost all the time. Scalability is also an important issue in order to be able to expand the services provided to an even bigger audience or even add new ones in a relatively easy way without having to restructure the whole existing application. Performance is very important and achieving it while having to serve millions of requests each day is a very difficult challenge. On top of all these, such applications require maintenance that is also a troublesome issue and has to be taken into account from the initial stages of the application's development. Finally the ease of use is of outmost importance to the end user that will use the application and has also to be taken into account.

Having in mind all the above it is now clear that the development of such an application is an extremely complex issue and requires great effort from the initial stage of the design

and development. All of those well known large scale web applications did not start in the form that are known today, they were much simpler but were designed in a way that allowed further improvements and supported the attributes previously described and in case they weren't they were redesigned this way. So in the present thesis an attempt to provide a solution to this problem will be made by using the MVC pattern and J2EE architecture in order to build a simple application that supports such features and explain the choices that were made in the technologies used and the design of the application in the context of the attributes previously described.

1.2. State of The Art

After explaining the problem that is faced in such applications in this part real world cases will be presented from some of the biggest applications in the world like Google, eBay, Amazon, YouTube and Flickr. For each of the above cases some general technological info will be presented and some stats regarding their performance and achievements in each business model they support since all of them provide different services and are the pioneers at what they do. Of course these cases are extreme examples of the best architectures used combining multiple programming languages and great distribution all over the world as well as custom solutions optimized to solve their specific problems. The case presented in this thesis is much simpler but tries to incorporate the basic principles and design methods that would allow its further development, in other words to have the potentials for further improvement. All the info provided below are from a web site that focuses on high scalability issues [9] and has gathered information on some of the biggest web applications in the world. It has extremely interesting contents regarding system architectures and technologies used in each of them.

Google's operating system platform is Linux and uses a large diversity of languages to achieve its business goals; these are Java, Python, C++. In 2005 Google had indexed 8 billion web pages, by now this should be even bigger. In 2006 it was estimated that they were using 450.000 low cost servers. Google also has created its own file system GFS (Google File System) for data retrieval and there are more than 200 GFS clusters in Google. Such a cluster can have 1000 or even 5000 machines. Pools of tens of thousands of machines retrieve data from GFS clusters that run as large as 5 petabytes of storage. Aggregate read/write throughput can be as high as 40 gigabytes/second across the cluster.

EBay uses also various technologies but the basic ones include: Java, Oracle, WebSphere, Mix of Windows and UNIX. Architecture is strictly divided into layers: data tier, application tier, search, operations. Now some stats concerning eBay which are really impressive will follow:

- On an average day, it runs through 26 billion SQL queries and keeps tabs on 100 million items available for purchase.
- 212 million registered users, 1 billion photos
- 1 billion page views a day, 105 million listings, 2 petabytes of data, 3 billion API calls a month
- 99.94% availability, measured as "all parts of site functional to everybody" vs. at least one part of a site not functional to some users somewhere
- 15,000 application servers, all J2EE

Amazon as all large scale web applications also uses a great variety of technologies like Linux, Oracle, C++, Perl, Java, Jboss. It started as one application talking to a back end. It was written in C++. Amazon is not stuck with one particular approach. In some places they use Jboss/Java, but they use only Servlets, not the rest of the J2EE stack. Now here are some stats also:

- More than 55 million active customer accounts.
- More than 1 million active retail partners worldwide.
- Around 100-150 services are accessed to build a page.

YouTube was founded on 02/2005. In 2006 it was acquired by Google and the number of requests it served almost tripled within several months after Google took over.

Technologies used include Apache, Python, Linux, MySQL, psyco (a dynamic python to C compiler), lighttpd for video instead of Apache. Here are some stats:

- Supports the delivery of over 100 million videos per day.
- 3/2006 30 million video views/day
- 4/2006 100 million video views/day

Flickr is the web's leading photo sharing site. It also uses a big variety of technologies to support its services, some of them are PHP, MySQL, MemCached for a caching layer, Squid, Linux, Perl, Java for the node service and many more. Here are some stats for Flickr too:

- More than 4 billion queries per day.
- Around 470 million of photos, and keeps 4 or 5 sizes of each
- Over 400.000 photos being added every day

1.3. Structure of the Thesis

In this part after presenting the problem, the objective of the thesis and the state of the art, the structure of the rest of this report will be presented. Then there is the part where the choice of architecture and framework is explained based on the attributes a large scale web application must have. After this a general overview of the selected framework and its features is presented. Next part consists of a very quick presentation of the application that will be developed in technical level as well as a more extensive high level presentation of what the application can do and the possible use cases it supports. This is done in order to help someone to understand what all the technical details that will follow try to achieve and what is their real world context. Following is the part where the technologies used in each application's layer are presented and also related examples are given based on what has been implemented in the application. The package structure and code organization then are provided to show how concepts like scalability and separation

of concerns happen in each layer and how this helps. Then main concepts of the application are presented and how they are dealt with by also providing configuration info on how to integrate multiple technologies together. After all the technical info and software architecture has been presented a full round trip of a request is presented in order to show how the various parts integrate with each other in a real sequence of a request. Then an interesting chapter with performance metrics is provided to show in practice the capabilities of the application and its design. Some info then on how to set up the application of this thesis and the configuration required is presented. Finally there is the conclusion part along with possible future work that can be done.

2. *Choice of Architecture and Framework*

In order to develop the application that will be presented some choices had to be made concerning the architecture and the framework that would be used. The choice of the framework is a very important matter because it is highly associated with the qualities of an application, for example scalability, expandability, ease of development and maintenance, current and future support, licensing issues, documentation, activity of the community that supports the framework and also stability and availability which are qualities related to the framework's maturity. All the above are very important issues someone has to take in account when choosing a framework in order to develop an application since this choice will be reflected on the application's value and performance. Finally the most important factor in this choice is the ability of the framework to support the system architecture you intend to use.

Show Face application is based on J2EE application architecture. J2EE technologies can be used to apply a well known architectural principle/design called MVC. MVC stands for Model View Controller and started with Smalltalk [1], that was originally used to match the input, processing, and output tasks with the graphical user model. However, it is straightforward to map these concepts into the domain of multi-tier enterprise applications. MVC divides an application in 3 different but collaborating layers.

- **Model** - The model layer has to do with the enterprise data and the business rules that perform actions on them (access, edit data). Often the model serves as a software approximation to a real-world process, so simple real-world modeling techniques apply when defining the model.

- **View** - The view shows in a graphic way the contents of a model. It requests for enterprise data from the model and specifies how that data should be presented. The view layer should also adapt to model layer changes. This can be achieved by using a push model, where the view registers itself with the model for change notifications, or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data (e.g. using Ajax technology).

- **Controller** - The controller intercepts view layer's requests and directs them to the suitable actions that access the model layer. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in a Web

application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

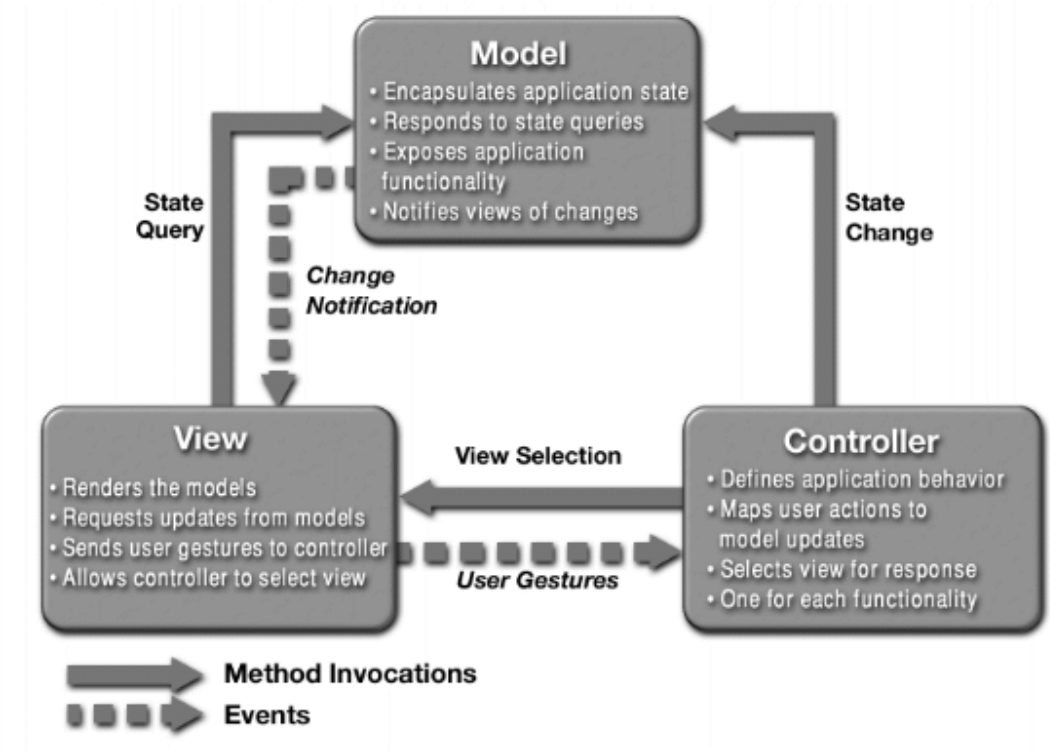


Figure 1. MVC Model Diagram [5]

The MVC design pattern has a lot of design benefits. MVC separates design concerns (data persistence and behavior, presentation, and control), decreases code duplication, centralizes control, and makes the application easier to modify. MVC also helps developers who are more able at a specific layer to concentrate only at that.

This pattern can be applied easily in J2EE architecture, and thus provides all the benefits to the application that will be developed. The J2EE platform was intended to solve the complexities that had to do with distributed and multi-tier application development and had great success in standardizing system services. The main issue though is that the fundamental problem of a simplified programming model was overlooked[2] and despite of the great adoption it had in the late 1990s and early 2000s, developing multi-tier

applications on the J2EE platform was a difficult and complex task and required great effort.

The choice of the Spring framework for the development of the Show Face application has a lot to do with the complexity of the J2EE platform as well as many other advantages it provides. The core J2EE platform consists of many technologies and APIs which are really complex and usually have a long learning curve. The difficulties encountered for the above reasons lead developing communities to look for other alternatives. For this reasons a lot of new frameworks have appeared and were built on various core J2EE APIs [2]. So today there is a great variety of frameworks to choose in order to implement a multi-tier application based on the MVC pattern.

The Apache struts framework for example (one of the most well known and used frameworks), helps to implement MVC architecture on the Controller layer and it is built on top of J2EE's servlet API. It provides various services and implementation out of the box in order to ease and speed-up development of a web application. Some other well known frameworks of this layer are Tapestry, Ice Faces and the newer Struts 2; all of these mainly provide the same features with some differences that should be taken into consideration when choosing one. But all of these are single-tier frameworks that address the needs of one of the 3 MVC layers.

Now on the other side on the Model layer there are frameworks like Hibernate that was created to solve the complex issues associated with J2EE's entity beans [2]. With Hibernate you can save data using POJOs (Plain Old Java Objects, simple classes) with minimal configuration and effort. These POJOs since they are not distributed objects like entity beans apart from being less complex also to lead to better application performance.

Spring framework also tries to address this complexity problem but unlike all the other single-tier frameworks described before, it provides a comprehensive and multi-tier framework that can be used in all tiers of an application [2]. Spring helps to compose the whole application and provides many out of the box services in order to speed up and ease an application development. Apart from these it also integrates with the best single-tier frameworks and gives great flexibility and choices. These are the main reasons that Spring framework was chosen for the development of the Show Face application.

Apart from those reasons above, Spring is one of the best choices not only because of its unique nature to support all MVC layers in a J2EE application, but also it is a wide-spread framework with great support and documentation as well as a big active community that work on it in order to improve it even more.

3. Overview of Spring Framework

Spring is an open-source framework, created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development* [3]. As it was previously said Spring's purpose is to address the complexity of enterprise application development. Spring makes it possible to use simple POJOs and make things that were previously possible only with EJBs, which are more complex and not always necessary. Any Java application can benefit from Spring in terms of simplicity, testability and loose coupling.

As an answer to the question of what Spring is the following statement is pretty descriptive and meaningful, Spring is a lightweight inversion of control and aspect-oriented container framework [3]. This is not an easy statement and in order to conceive it better, the meaning of its attributes will have to be analyzed and connected with the Show Face application in order to see how each of them affects the application.

- *Lightweight.* Spring is lightweight in terms of both size and overhead. The entire Spring framework can be distributed in a JAR file whose size is just over 1 MB. Also the processing overhead required by Spring is negligible. This can be seen in Show Face application by checking the response time of the application. With Spring's lightweight attributes, use of POJOs instead of EJBs and several application specific design choices the performance of Show Face application is at a good level.

- *Inversion of control*. Spring promotes loose coupling through a technique known as inversion of control (IoC). When IoC is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. This will be presented in the xml configuration files of the Show Face application where dependency configuration will be demonstrated.

- *Aspect-oriented.* Spring comes with rich support for aspect-oriented programming that helps development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they're supposed to do, follow just the business rules and are not mixed with other system services. They are not responsible and aware of other system concerns, such as logging or transactional support. In Show Face application the transaction management happens with Spring AOP's help, the transaction management configuration is done on xml level thus leaving the business objects to do their own job and be more readable. This way it obvious that separation of concerns truly happens.

- *Container*. Spring is a container in the sense that it contains and manages the life cycle and configuration of application objects. How many times each of the beans should be created (create one single instance of a bean or produce a new instance every time one is needed) and how they should be associated with each other is

completely handled by Spring based on xml configuration files defined by the developer. That is why Spring is also a container and this can be seen in Show Face application by the fact that a simple web server (servlet container) such as tomcat has been used and not an application server.

- *Framework*. Spring makes it possible to configure and compose complex applications from simpler components. Application objects are composed declaratively, typically in an XML file as it will be shown in Show Face application. Spring also provides much infrastructure functionality, leaving the development of application logic to the developer and makes things less complex.

All these attributes of Spring help to write code that is cleaner, more manageable, and easier to test. They also set the main stage on top of which other frameworks or Spring sub-frameworks can be easily integrated. This will be done in the next section where the sub-frameworks/frameworks that were integrated with Spring in order to build the show face application, will be presented.

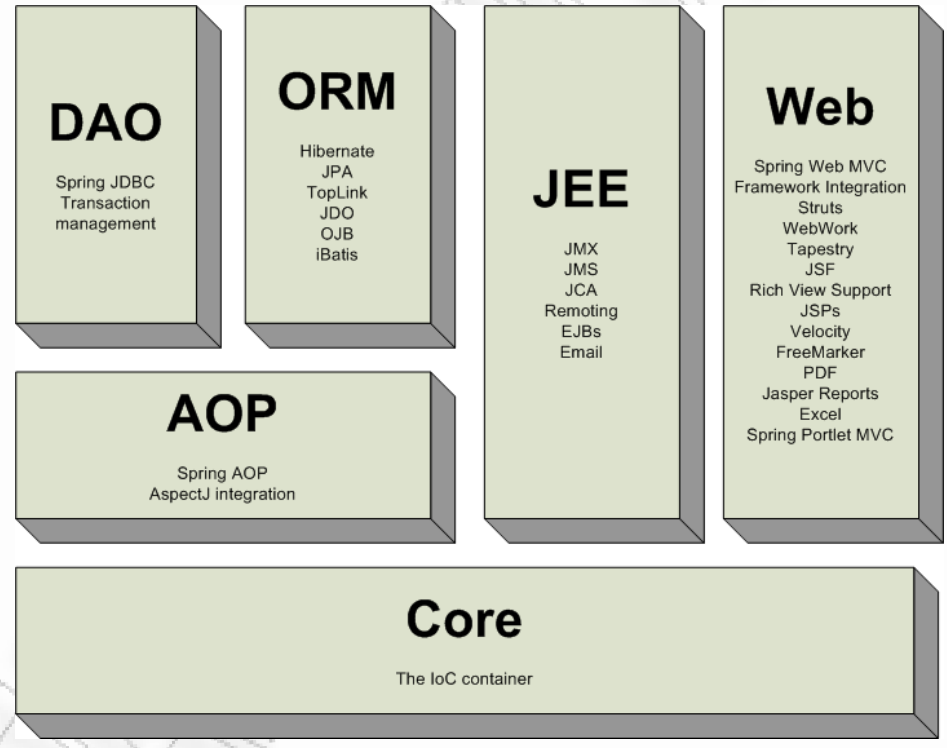


Figure 2. General View of Spring Framework [6]

4. Show Face Application Presentation

In this part a brief overall presentation of the application that was developed will be made. As previously noted the whole circle of the application's development will be presented in the rest of the document trying to point out how concepts like availability, scalability, performance and maintainability affect the choices made on the technologies used, on how the integration happens between them and the design issues taken into account to built an application in such a way that decreases overall complexity and still supports the concepts previously noted. At the first part of this chapter a brief description of the application will be given and the technologies that were selected will be noted. Then a high level presentation of the application will be provided with screenshots and use cases to show what it can do and how it looks like.

4.1. Introduction to Show Face

The application's name is Show Face and the name has been chosen based on the name of another large scale application named Facebook, of course there is no real connection between these two apart from the general idea of social services they provide. Show Face is a small application developed in the context of this thesis that provides several simple social networking services. The application gives you the possibility to create an account and then to upload and manage your images online, it supports most of the well known image types. Apart from this you can also make on line friends from other users that have subscribed to the application. When you make a friend then you can see his photos as well as he can see yours. Through the presentation of this application various J2EE related technologies will be demonstrated and also the integration between them will show how to make a scalable and highly available web application.

Show Face has been implemented using a J2EE system architecture, and more specifically with the use of Spring Framework version 2.5.5. The database used to manage the data storage needs of the application is hsqldb which is a lightweight java-based database. Finally the server that hosts Show Face is an apache tomcat (version 6) distribution. The choice of a simple web server (servlet container) such as tomcat and not an application server like Bea Weblogic or Jboss, was made because of the possibilities that Spring Framework offers. With Spring (and Hibernate's help too combined with Aspect Oriented Programming expressions) it is possible use POJOs (Plain Old Java

Objects) and manage transactions and db interaction actions without the need of a resource-demanding and cumbersome Application Server. Of course application servers provide a lot of other facilities and integrated solutions but they are not always the best choice, it depends on the project's requirements. For more info and details on the technologies used you can refer to the links page of Show Face application. The architecture and design choices will be analyzed further after a high level presentation of the application so as to have an overall view of how this application seems and the possible use cases it supports.

4.2. High Level Presentation

In order to begin with the presentation you can see below (3) the first screen you encounter when accessing the Show Face application. It is the login screen where you can do certain actions. Log in if you are already a subscribed user, register yourself so that you can Log In and have access and finally to see some info by clicking on the about link. On the top of the page there are also certain links where you can see some info for the creator of the application or sent him a mail. Of course if the login fails or a field is missing during login, an appropriate message is being shown

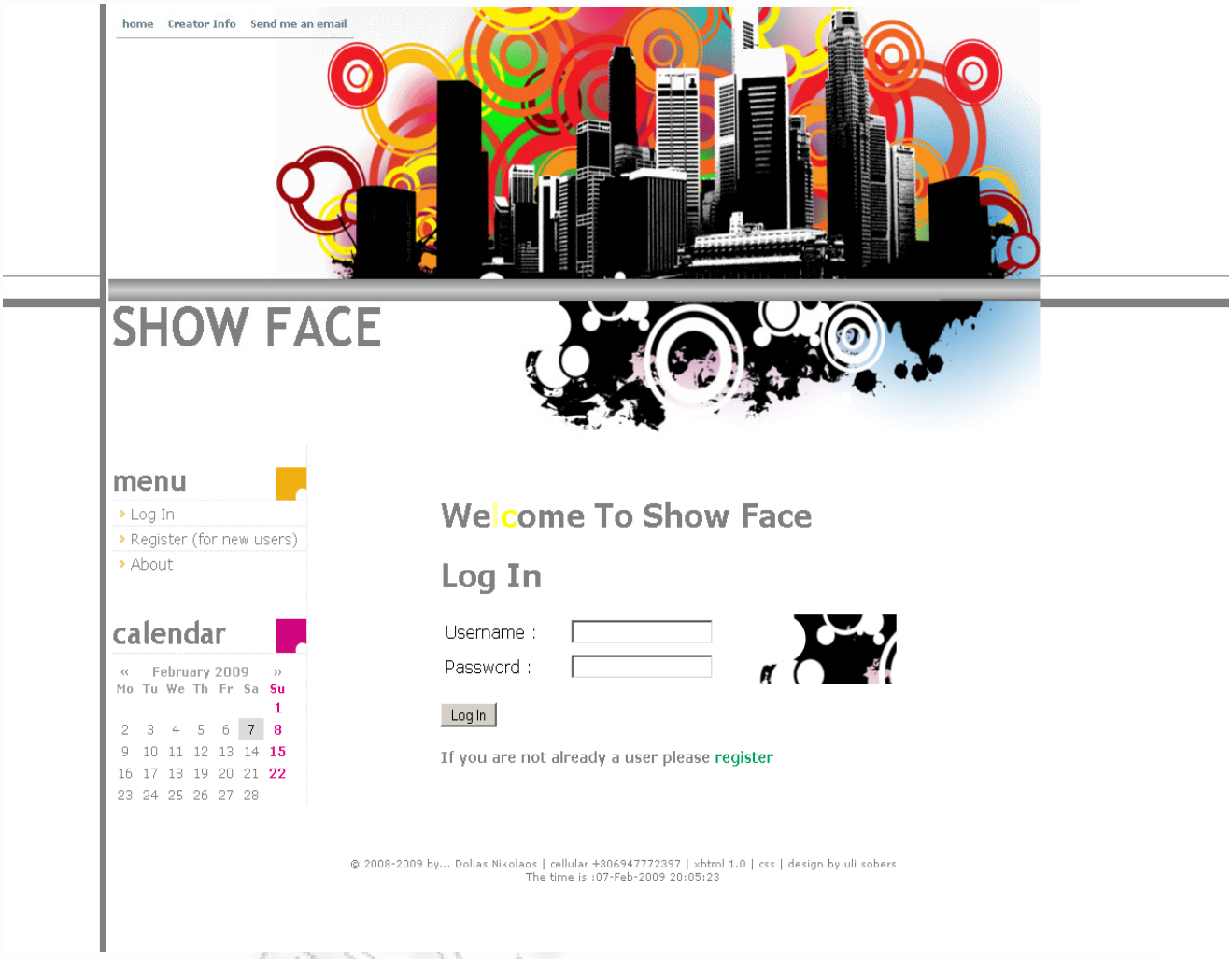


Figure 3. Log-In Screen

Now as a next point the register functionality will be reviewed. The first (4) of the 3 below screenshots shows the register page as it is when you access it to subscribe. Then at the second screen (5) you can see the same page with validation errors, from incorrectly filled fields. Finally at the third screen (6) you can see the register form correctly filled, and by submitting it your user account is created.

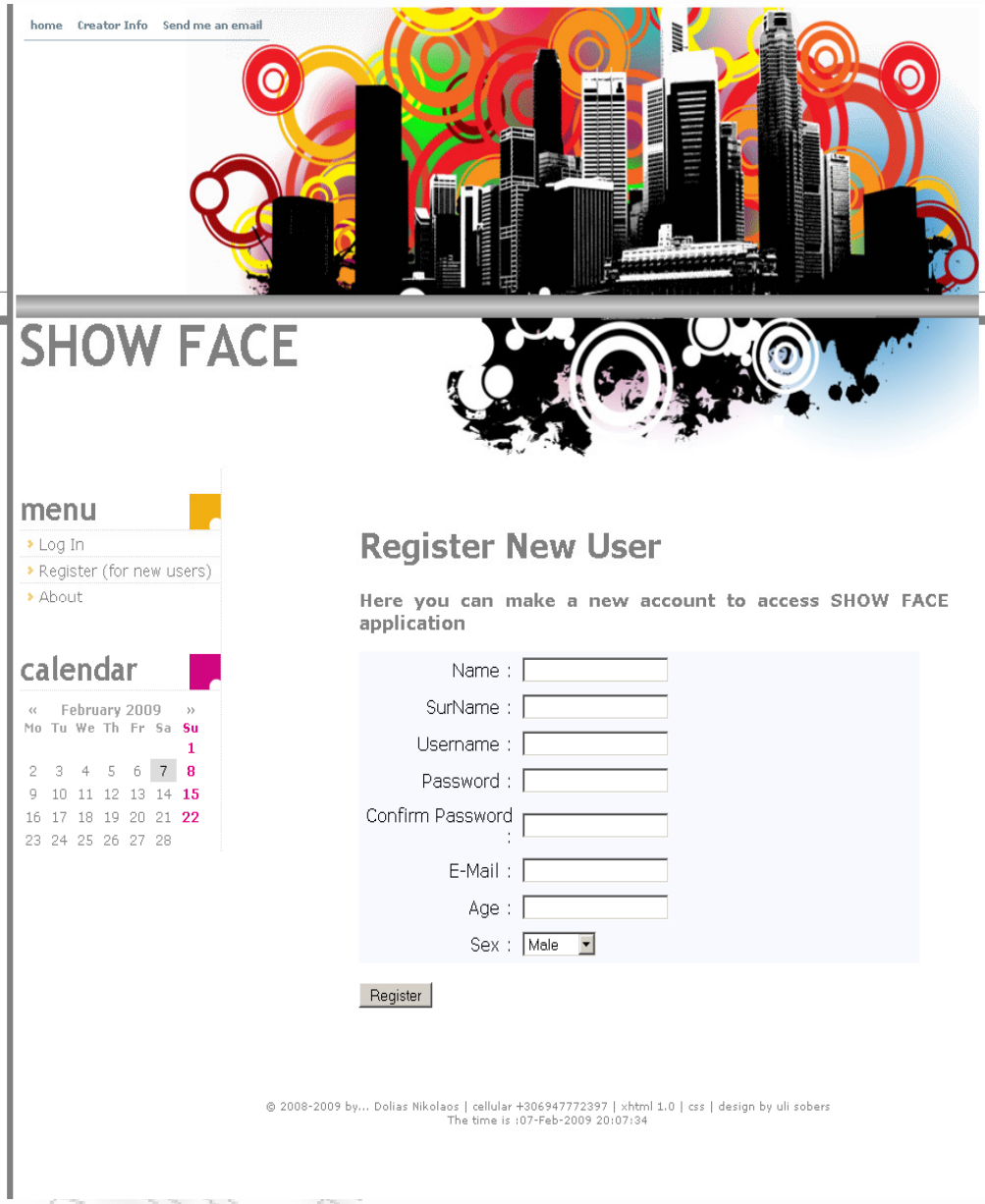


Figure 4. Empty Register Screen

Figure 5. Failed Register Screen

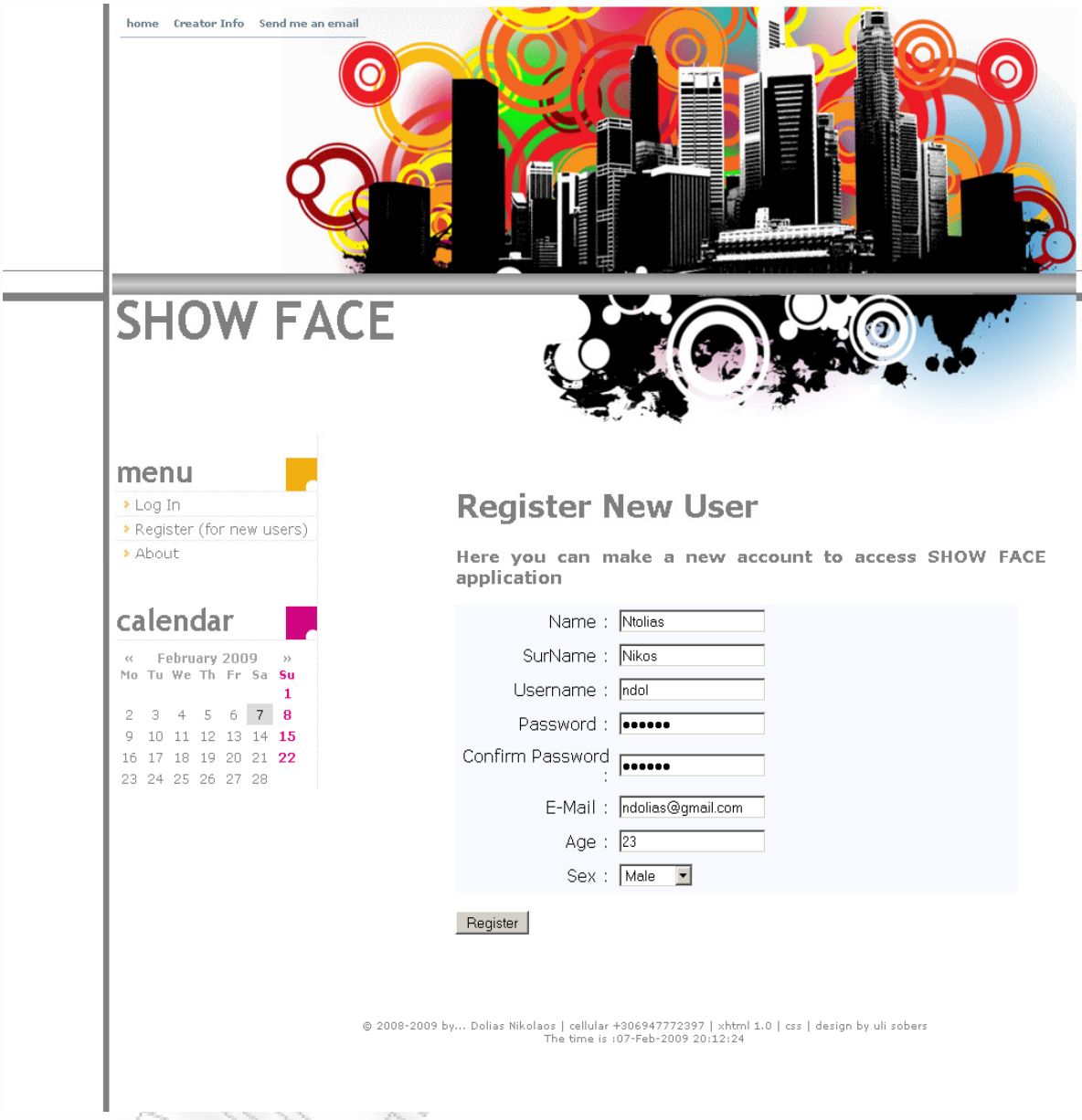


Figure 6. Correctly Filled Register Screen

After viewing the register section, the next thing that will be presented is the about page. In that page (7) some general information for the application is provided, as well as the context in which Show Face was created. Finally at the bottom of the about page, since

Show Face is part of a post graduate thesis, you can see the links of the Supervisor of the thesis and of the student. These links are connected to their respective bio info pages.

Figure 7. About Page Screen

After reviewing the pre-login pages and since now a valid account has been created, it is possible to log into the application. Below (8) you can see the initial screen, the home

page, after the first login. Here there are three sections at the home page; the first one shows the latest images that you have uploaded. The second one shows friend requests that others have sent to you and you have to accept or reject them. The third section shows friend requests you have sent but have not been answered by the other user yet, if they are answered they will be removed and if your request was accepted this user will be visible in View Your Friends link, if they rejected your invitation then you will be able to sent them an invitation again from Make Friends link. These links and the others of the post login menu will be later reviewed one by one.

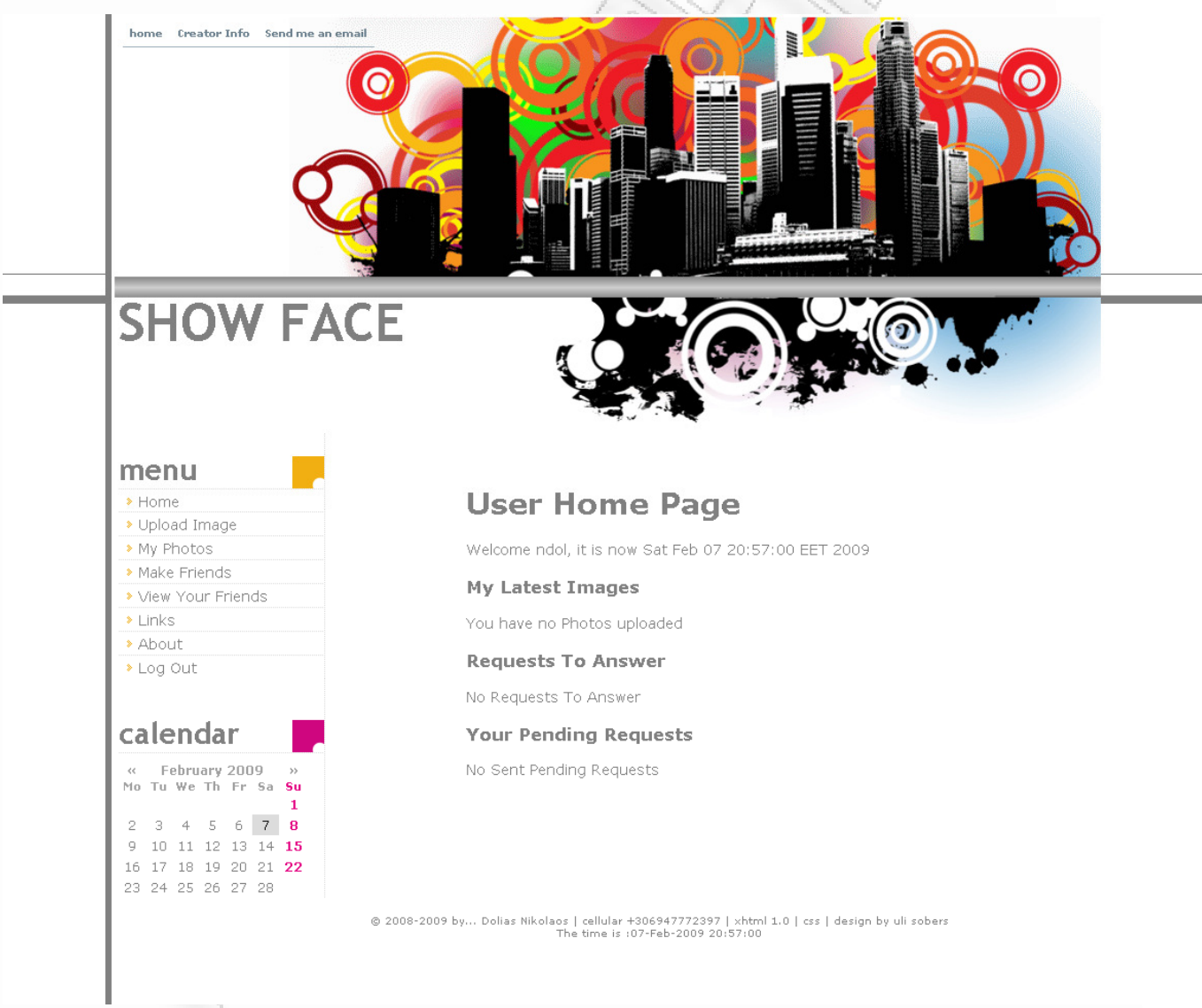


Figure 8. Home Page after First Log In

As it is shown in the above screen (8) no action has been taken yet and that is why after the first log in all the sections are empty. In order to start using the services the application offers, the upload image functionality will follow next. In order to get to that screen click on the respective link of the home page menu "Upload Image". Then the following screen (9) appears, there a name for the image has to be provided and give the image file's path in order to upload it. In case the file you provide is not an image file, a validation error will appear on the page, like in screen (10) where an attempt to upload a PDF file failed.

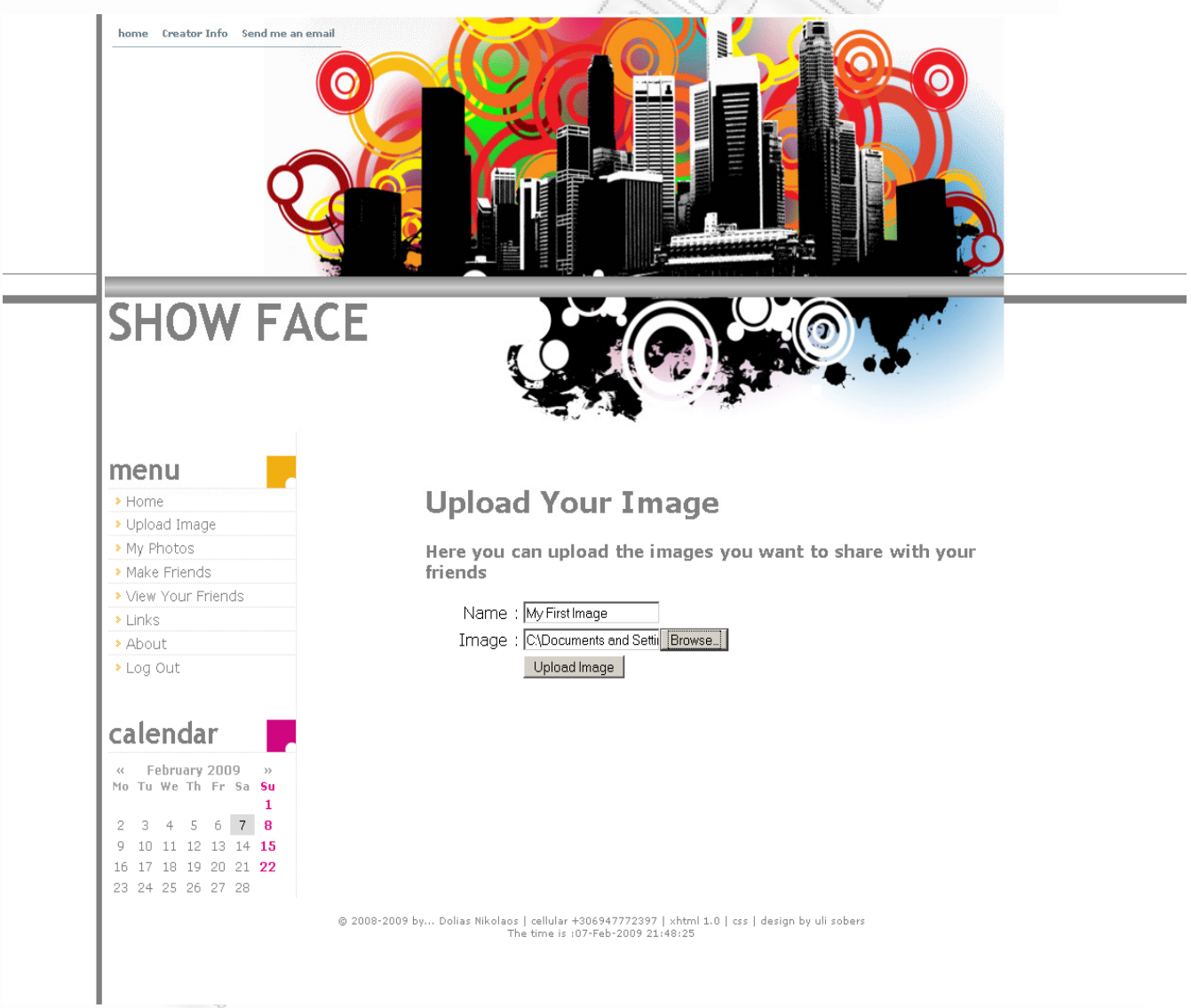


Figure 9. Upload Screen with Valid Input

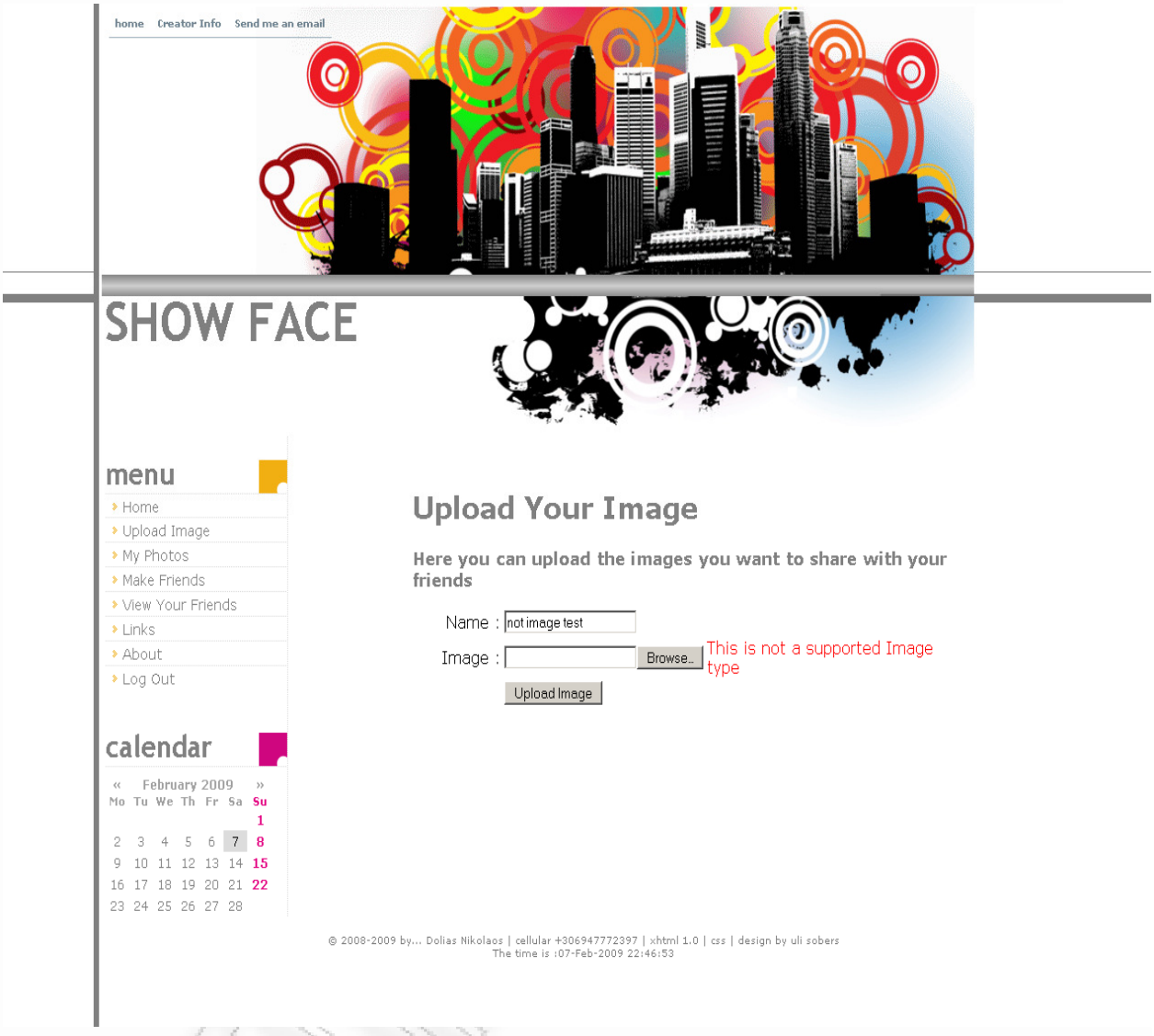


Figure 10. Failed Attempt to Upload a PDF (Not-Image) File

Finally if the image upload is successful you will be redirected to the home page, and in your latest photos section you will see the thumbnail of the image you uploaded, as you can see below (11) along with a confirmation message on the top of the page.

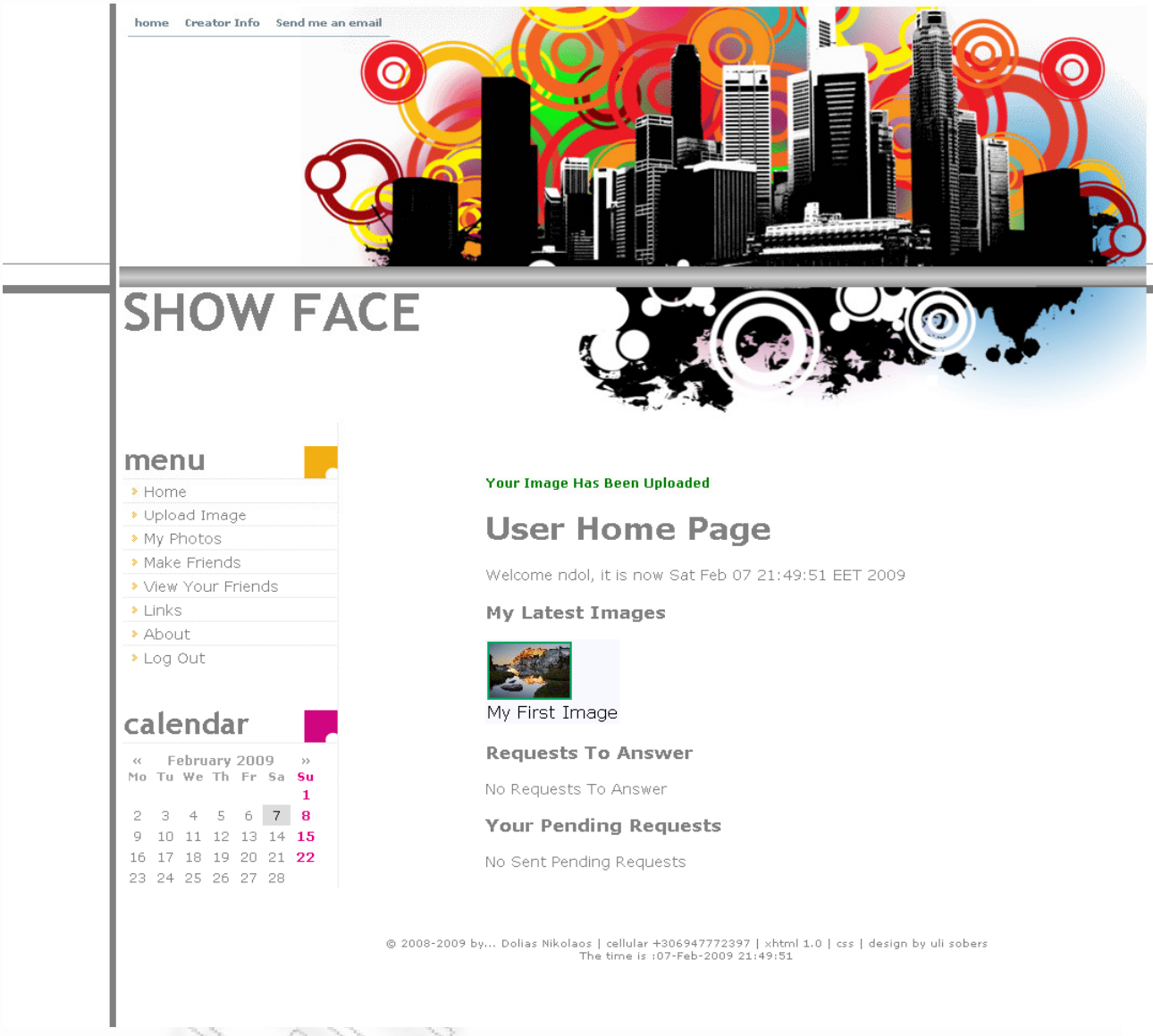


Figure 11. Home Page after Successful Image Upload

As it has been noted before, the image section of the home page only shows the latest 5 images you have uploaded, in case you want to see all your images you can click on My Photos link from the menu, and you will be forwarded to the following screen (12). In this page you can see all your images and also delete any image you do not want any more by pressing the delete link.

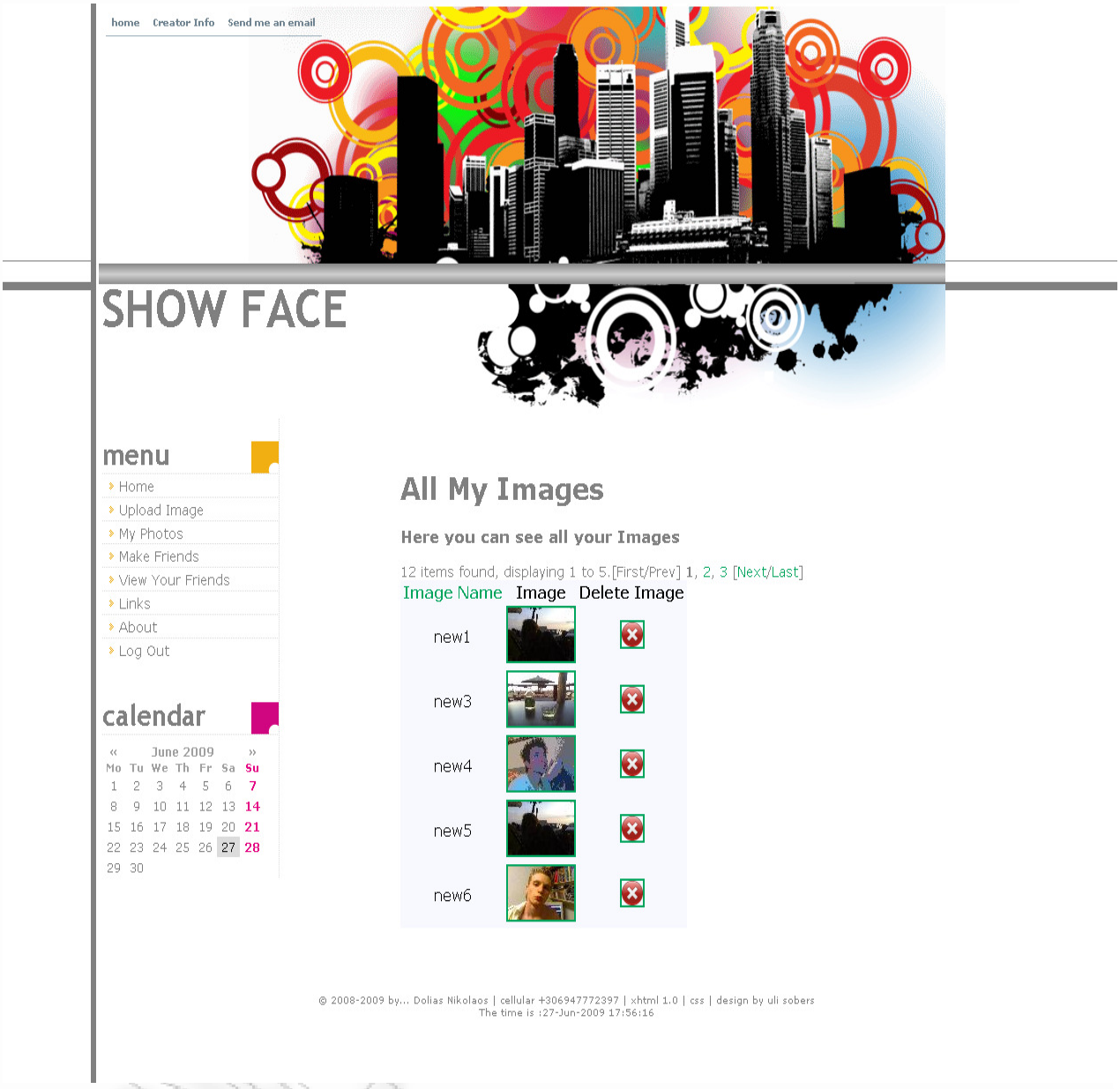


Figure 12. My Photos Screen

The next functionality of the menu that will be reviewed is the mechanism to make friends. By clicking on the menu link named Make Friends you will see the following screen (13). There you can see the users you are able to invite to become your friends (Send Friend Request). You will be given the ability to send a friend request to users that are not already your friends or to users for which no request is pending between you and him/her (neither accepted nor rejected). Finally by clicking the link "Send Friend

Request" a request will be sent to the user of your choice (you can send of course multiple requests). After this action the corresponding section in home page will be updated as you will see later.

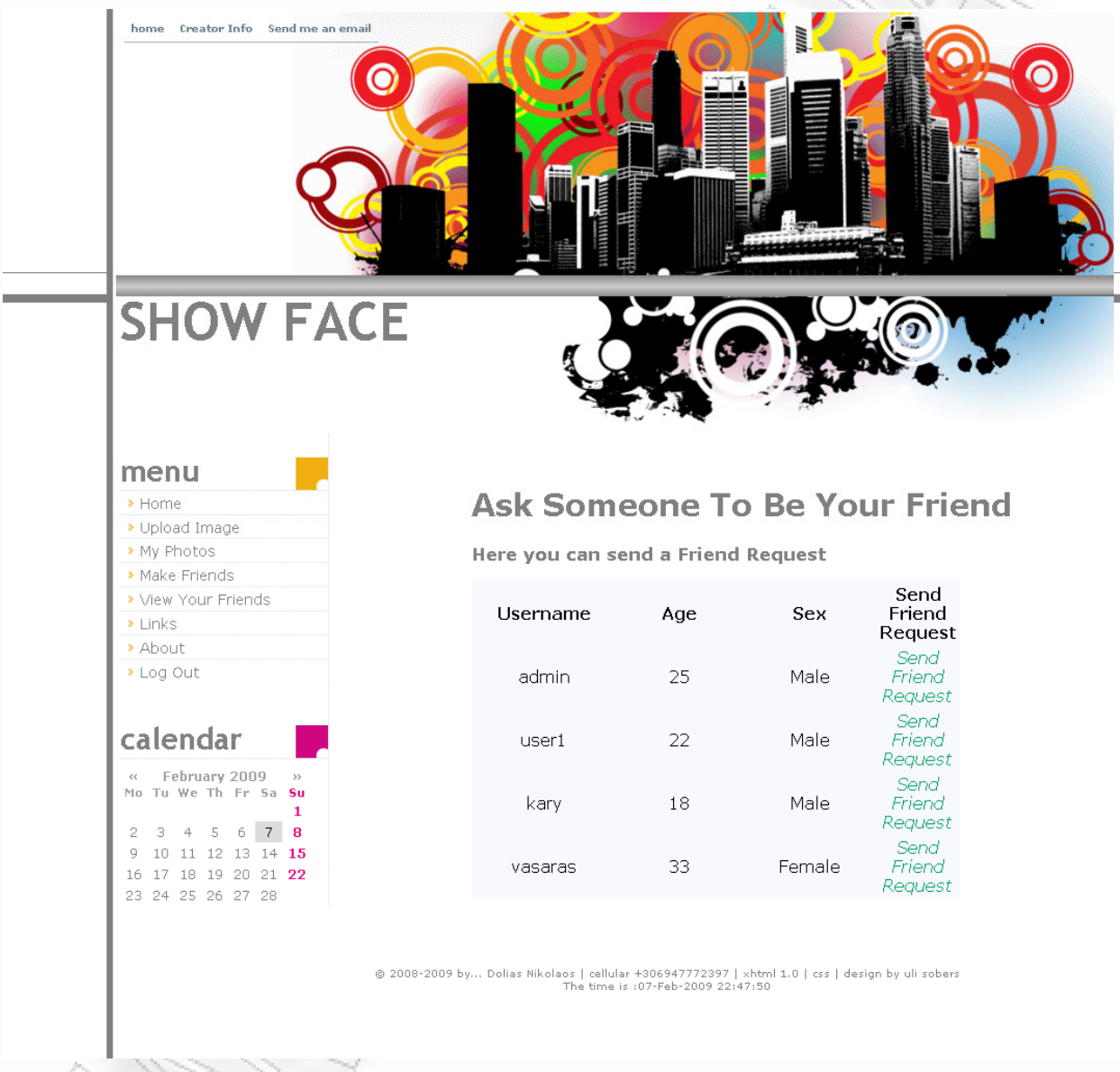


Figure 13. Make Friends Screen

Now at the home page screenshot you can see below (14), the photos section has been updated with more images. Also there is a request "user1" has sent you and asks you to become friends, you can reject or accept it. Finally at the third section you can see that

there is a pending request you have sent to user “admin”. If he accepts it you will become friends and you will see him in the next link that will be presented, in case he rejects your request you will be able to send him a new request from Make Friends link which was previously examined.

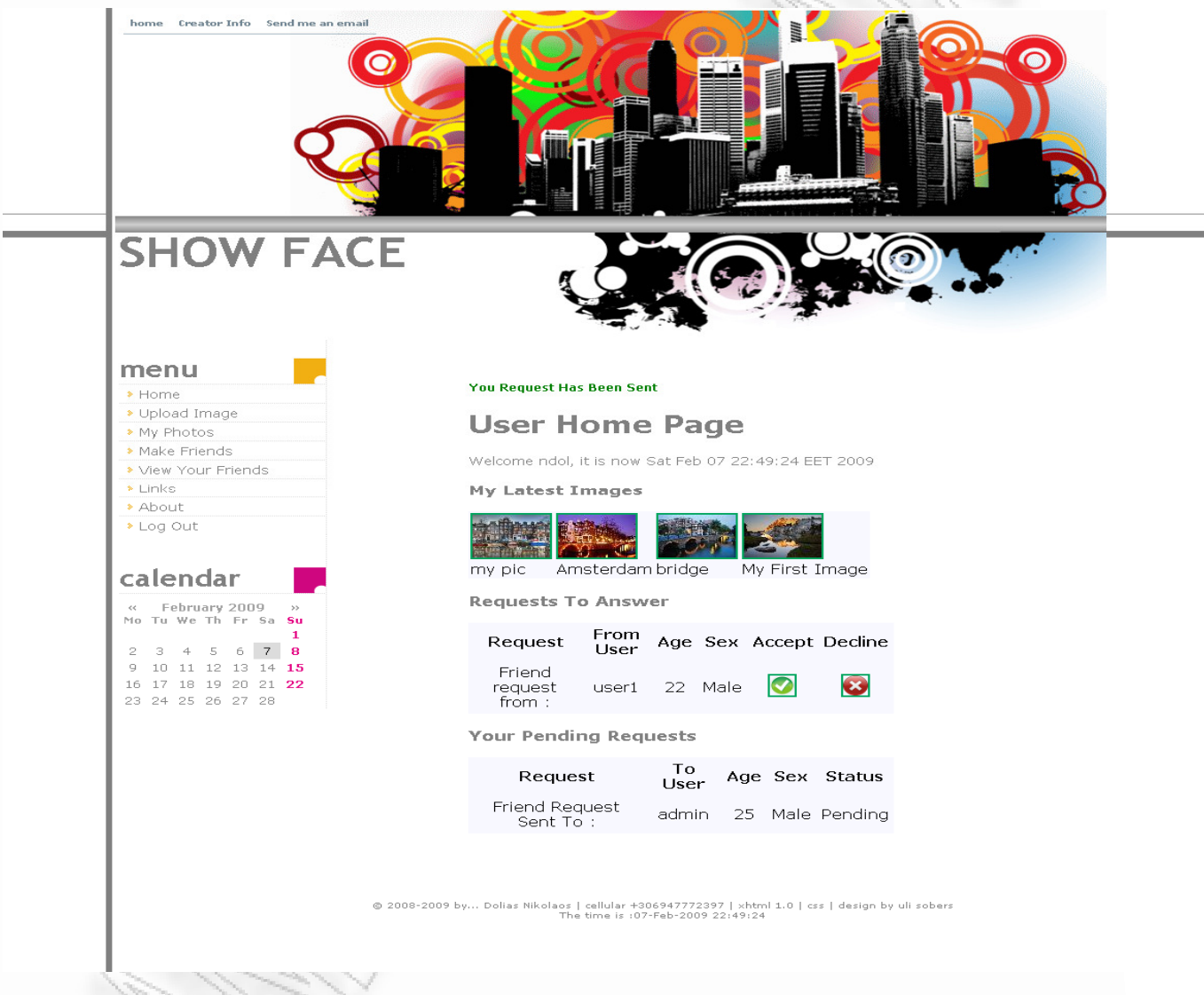


Figure 14. Home Page Screen with Requests, Photos

The next thing to examine is the View Your Friends link. In this section you can see all those that have accepted your friend invitation or you have accepted theirs. You can see for each one of them the 5 latest images they uploaded and some more info on them. If you want to see all their images you can click on the links “here” or “more photos…” Also in case you change your mind at some point you can delete them from being your

friends and you will stop sharing images. If you delete someone from your friends you won't be able to see him in View Your Friends link but you will be able to send him again a friend request in Make Friends link. Here is a screen (15) that depicts the above case.

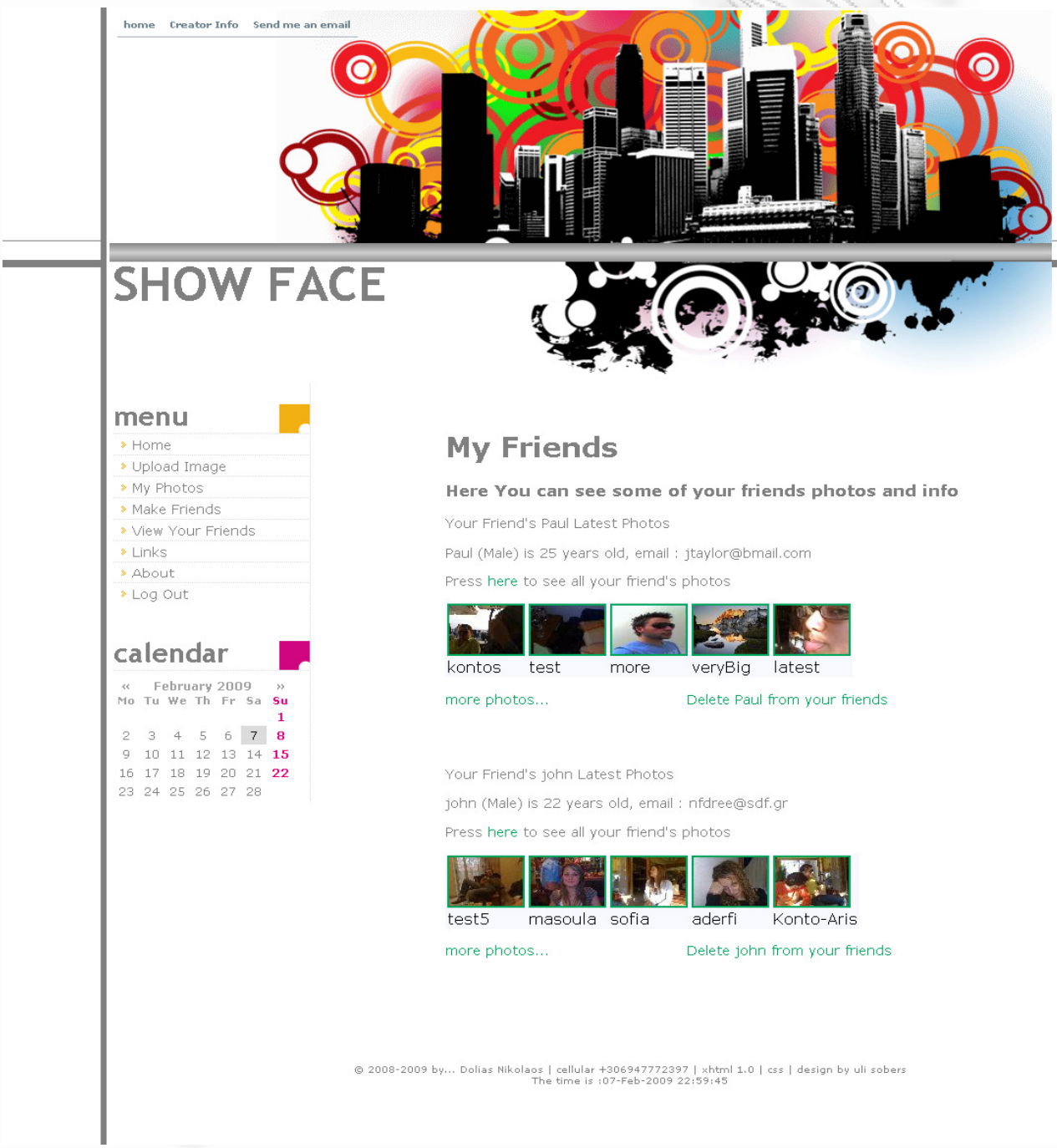


Figure 15. View Your Friends Screen

Then if you click on the link more “photos…” of screen (15) for a user, let’s say for John, you will see all his pictures as you can see at the following screenshot (16).

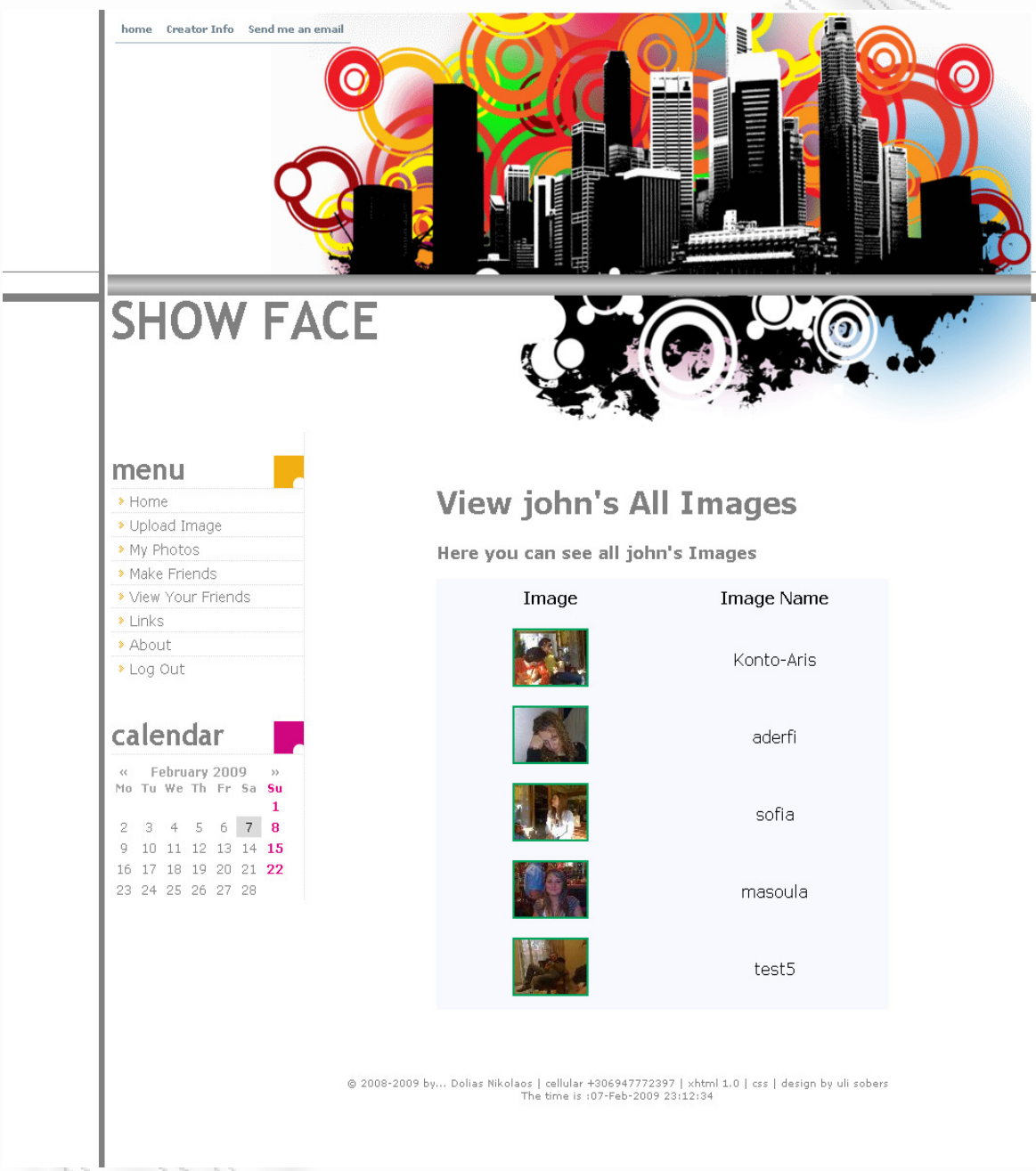


Figure 16. View Your Entire Friend’s Images Screen

The final screen that will be reviewed is the Links page that can be accessed from the menu. In this page there are links to the official sites of the tools – technologies used to implement the Show Face Application. Here is the links page (17).

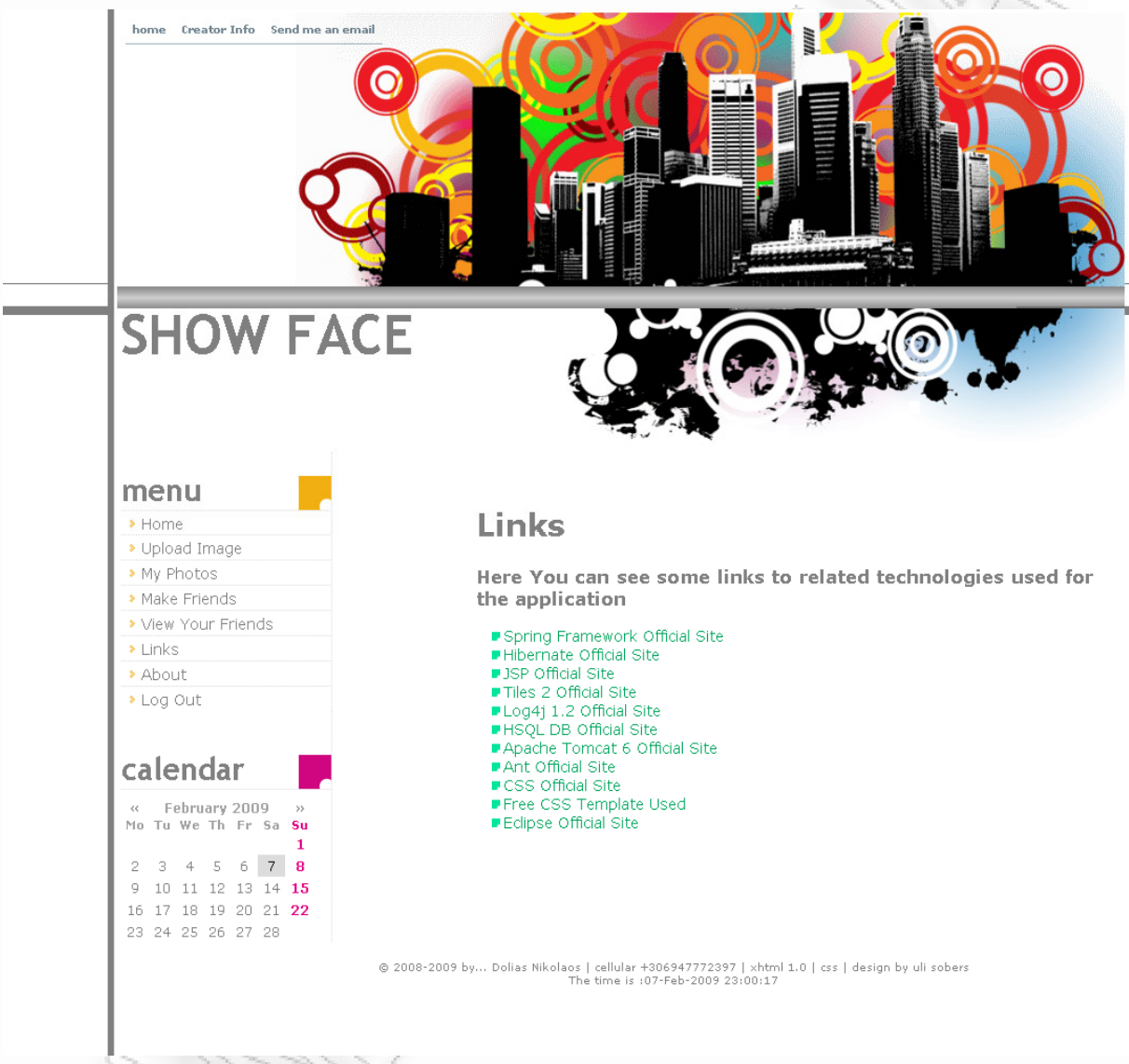


Figure 17. Links Page Screen

The only action in the menu of the Show Face application that has not been yet presented is the Log Out link. As you may already know this action is used to exit the Show Face application and in case you press it you will be redirected to the log in screen. After

logging out you are required to provide again your credentials in the log in screen if you want to access the services that Show Face provides.

This was a high level presentation of Show Face application where only minimal technical and architectural information is disclosed. The next part will be the technical analysis. In that part the software architecture of Show Face will be presented in detail as well as the reasons for several design choices that were made. Also it will be more obvious which part and requirements each of the selected technologies support in order to have the final result.

5. *Layered Frameworks of Show Face*

As it has been mentioned Spring will be used in Show Face application as the general multi-tier framework choice. Now it will be examined which framework will be integrated with Spring in each layer of the MVC pattern in order to build the application.

5.1. Model Layer

In the Model layer one of the most important features of Spring's Container will be used. This feature is its ability to manage application objects (POJOS) on its own without the use of an application server. Spring's container and the xml configuration will handle the lifecycle of the objects that are created by the user and also of those created by Spring. Moreover thanks to the Spring's built in AOP support transactions with the database can be managed in an easy way and keep transaction management decoupled from the business objects, since transaction configuration will be in the xml files in the form of AOP expressions.

Now apart from the main model of the application a choice has to be made as to which one will be the data access framework which is very important since it is the way to interact with the database. Some of the most well known data access solutions to interact with a database are jdbc, hibernate and ibatis. Jdbc is the well known API of java that defines how a client can access a database. When using jdbc it is the developer's responsibility to create the database connection, open and close it when required and do manually the mapping of the java objects to the database, which means that a lot of manual sql statements will have to be included in the code thus the complexity of the code and the coupling of an application to the specific jdbc based implementation will be high. This is something not desirable when implementing a scalable and relatively big application with many objects and associations; on the other hand it is the simplest way to interact with a database. Now on the other hand hibernate and ibatis are ORM (Object Relational Mapping) persistence frameworks that associate directly a java object to a relational database. Their main difference [4] is that hibernate allows the creation of an object model by the user, and creates and maintains the relational database automatically (based on configuration files), ibatis takes the reverse approach: the developer starts with an SQL database and ibatis automates the creation of the Java objects. Ibatis approach is mainly preferred in cases where the developer does not have full control on the database schema. In Show Face application that is not the case since there is full control over the database. Apart from this hibernate was chosen because of the great built in support that Spring provides for it.

With Hibernate templates of Spring it is possible to access the database and retrieve or insert populated java objects in very few lines of code without having to worry about

opening or closing the database connection or manually write sql statements (except from cases where there is need to improve performance for example) [8]. Also you do not have to surround with try catch statements every database interaction as in jdbc because exception handling is taken care by Spring. It is also worth to explain how spring handles exception handling concerning the db connection and how this helps. When jdbc forces the developer to surround with try catch statements every db interaction this makes the code cumbersome and more difficult to read, also this try catch block has no meaning since it should be used to remedy an erroneous situation but if the database server is off line there is nothing that can be done. This is what Spring has taken into account in the interaction with hibernate and this way there is no need to surround with try catch blocks every database interaction unless it is explicitly required to.

All these features concerning AOP, hibernate and the whole configuration of the model layer will also be presented in following chapter with code samples from the Show Face application.

5.2. Controller Layer

Concerning the web layer and more specifically the controller layer the choice for Show Face is the Spring MVC framework. Spring MVC is a sub-framework inside the Spring Framework that someone can choose to use in the web layer of an application. Of course Spring integrates very well with most of the well known web frameworks available. The choice for Spring's proprietary web framework was made because of the ease in the integration between them since they are made for the same project as well as the flexibility to use Spring's characteristics such as dependency injection etc in the web layer, generally it offers whatever is required for building most kinds of web applications. Since there are various web frameworks for J2EE applications each framework has its advantages and disadvantages and it is not possible to conclude which one is the best. Also there are usually workarounds to solve some of the differences they have. For example Spring MVC does not support directly Ajax related tags to implement easily such services, other frameworks do (Struts 2 for example). The solution in these cases is to use one of the numerous Ajax frameworks that exist and manually integrate it with Spring which will just need a little more manual configuration than the web frameworks where this support is already built in.

Let's explain now how Spring MVC handles a client request and see in more detail the components that take part in such an action, meaning what happens when the client (a web browser) sends an http request to the server that hosts the Show Face Application. The flow will be described as shown in the picture below. The flow begins when a client

sends a request (1), then Spring's MVC Dispatcher Servlet is the point of entrance for every request and delegates it. As in other web frameworks Spring receives the requests through a front controller servlet. This front controller servlet will receive the request and then dispatch it to the correct component in order to process it, in this case the front controller is the Dispatcher Servlet. In order for the request to be handled the Dispatcher Servlet has to send it to the correct Controller (the Controller is the component that will handle the request). The way to find the correct Controller is for the Dispatcher Servlet to query another component (2) the HandlerMapping. The HandlerMapping usually maps a URL object (the client's request) to a Controller object that will explicitly handle this request. After the Dispatcher Servlet has the correct Controller by asking the HandlerMapping, it sends the request to the Controller (3) in order to process it (a well-designed Controller performs little or no business logic itself and instead passes responsibility for the business logic to one or more service objects). Once the controller processes the request it returns (4) a ModelAndView object to the Dispatcher Servlet. The ModelAndView object may contain a View object or a logical name of a View object. If it contains a logical name of the View object and not the object itself (as in Show Face Application) then the Dispatcher Servlet will have again to query another object(5) to learn which one is the actual View object. This object is the ViewResolver that based on the logical name it will point to the actual View object that should be used, this is used in Show Face while using Apache Tiles 2 and a call to the ViewResolver is required in order to get the reference to the actual View. Finally since now the Dispatcher Servlet knows the View object, it dispatches the request to it (6) and now the View Object is responsible for rendering a response back to the client.

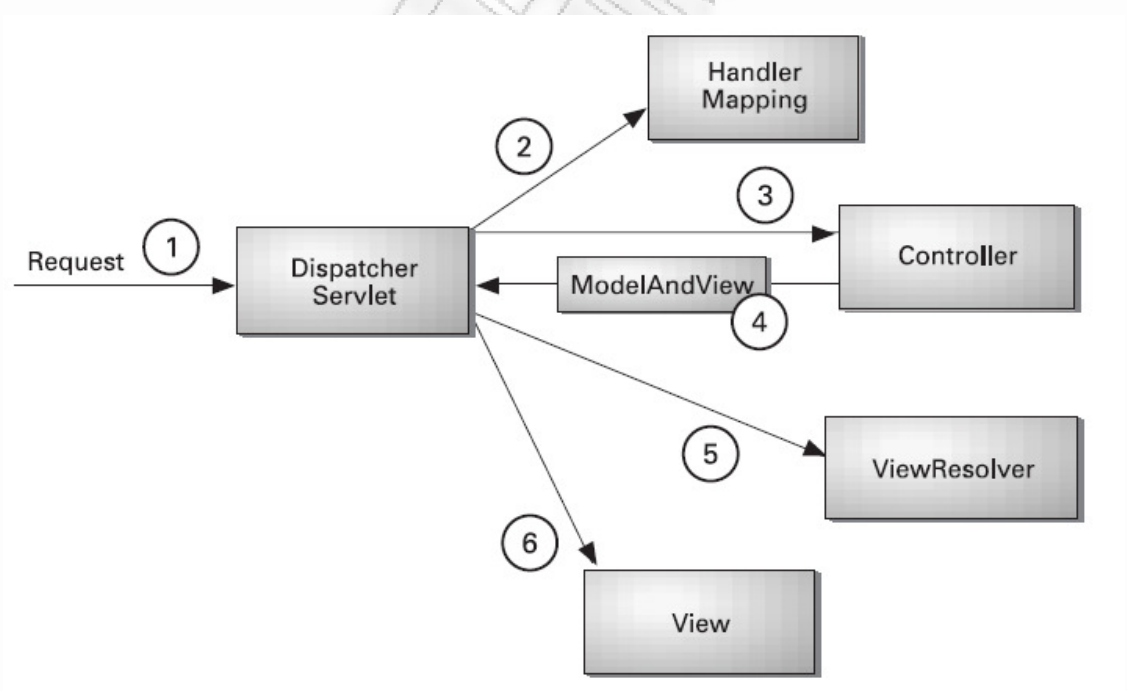


Figure 18. Life cycle of a request in Spring MVC [3]

This whole process in order to be better understood will be described based on an example of the Show Face Application. The example that will be used is based on what happens when the user presses the register submission button in the register page of the application (see screenshot 2). When the link is pressed the user's browser will send a URL request like this *http://<Server Name>:<Port if not 80>/showface/registerUser.htm* plus the registration data in the post request that are not visible in the URL. As it has been described the request will be handled by Spring's MVC front controller which is the Dispatcher Servlet. Then in order to find to which Controller this request should be sent to it asks the HandlerMapping. In Show Face application the HandlerMapping has this association (defined in the configuration xml files) "<prop key="/**/registerUser.htm">/registerUser.htm</prop>". This means that for any url of the type /**/registerUser.htm the controller that will handle this request is named "/registerUser.htm". After checking in the xml servlet configuration file used for Show Face for a defined controller named "/registerUser.htm" the entry below will be found:

```
<bean name="/registerUser.htm"
class="showface.web.RegisterAccountFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="account"/>
        <property name="commandClass" value="showface.domain.Account"/>
            <property name="validator">
            <bean class="showface.validators.RegisterUserValidator"/>
        </property>
        <property name="formView" value="registeruser"/>
        <property name="successView" value="login.htm"/>
        <property name="accountManager" ref="accountManager"/>

   </bean>
```

The above xml part of the configuration file has many parameters that are not required to explain the flow of a request in Spring MVC , so only the needed ones will be analyzed. The controller with the name "/registerUser.htm" has been found. The class attribute nearby shows the java class that is the controller in reality and will do the job of the registration action (not in controller level since it requires database access, it will propagate the request in model layer via interfaces that are used in Show Face design). When the controller finishes its work if everything goes fine and there is no validation error (for example passwords do not much), then this command in the controller will be executed:

```
return new ModelAndView(new RedirectView(getSuccessView()));
```

This command will return the new ModelAndView object to the Dispatcher Servlet. To be more specific a redirect (not a forward) is requested in order to avoid double submission errors. Let's show the differences between these 2 actions, since they are fundamental concepts in web applications.

Forward

- a forward is performed internally by the servlet
- the browser is completely unaware that it has taken place, so its original URL remains intact
- any browser reload of the resulting page will simple repeat the original request, with the original URL

Redirect

- a redirect is a two step process, where the web application instructs the browser to fetch a second URL, which differs from the original
- a browser reload of the second URL will not repeat the original request, but will rather fetch the second URL
- redirect is marginally slower than a forward, since it requires two browser requests, not one
- objects placed in the original request scope are not available to the second request

In general, a forward should be used if the operation can be safely repeated upon a browser reload of the resulting web page.

This redirect is requested to be done on the result of the method getSuccessView(). Now as it can be observed in the xml excerpt provided above there is one attribute called "successView" and has as value "login.htm". So method getSuccessView() will return the URL that the browser should request. Now since redirect consists of 2 steps (2 browser requests) this is the second one (the first one was the register related URL that checked up to now) and asks the browser to call the */**/login.htm* URL. The whole procedure up to now will be repeated since now a new request is made and the result will be the final URL (the second one) that the browser will request to get its final View. Checking the controller configuration of the second request */**/login.htm* the following can be found in the configuration file:

```
<bean name="/login.htm" class="showface.web.LogInFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="login"/>
```

```
        <property name="commandClass" value="showface.domain.Account"/>
            <property name="validator">
            <bean class="showface.validators.LogInValidator"/>
        </property>
        <property name="formView" value="loginUser"/>
        <property name="successView" value="hello.htm"/>
        <property name="accountManager" ref="accountManager"/>

    </bean>
```

The attribute that will give the name of the final View Object is the one called "formview" and has value "loginUser". Attribute "successView" that was previously used is only for submission actions when the user requests something. In this second request just a view is needed and not to submit data. So the logical name of the final View Object is "loginUser".

Now in order for the Dispatcher Servlet to find the actual View Object the help of the ViewResolver is required. In Show Face application the use of Apache Tiles 2 makes this case a little bit more complex but again in the end the ViewResolver based on his mapping will return the real view. The thing with the use of Tiles is that changes a little bit the configuration file and it seems like this:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
       <property name="definitions">
            <list>
                <value>/WEB-INF/tiles-defs.xml</value>
            </list>
        </property>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass"
value="org.springframework.web.servlet.view.tiles2.TilesView"/>

</bean>
```

With a few words these lines of xml configuration show how easy it is to integrate Spring MVC with Tiles and the most important is that they show the resolver's mapping is in the file "`tiles-defs.xml`". So finally the ViewResolver integrated with Tiles mapping file shows the connection of the logical name "loginUser" with its actual view in the part below:

```
<definition name="loginUser" extends="main.layout.pre.login">
      <put-attribute name="title"  value="login.user.title"/>
      <put-attribute name="body" value="/WEB-INF/jsp/login.jsp"/>

 </definition>
```

Tiles 2 will be explained in more detail later but for now what is important is the value of attribute "body" which shows the real view object that the server will send to the browser. This one is the login.jsp file, which will take the user to the login page after performing a successful registration submission. To sum up the request's sequence from the beginning:

- *http://<Server Name>:<Port if not 80>/showface/registerUser.htm* that will instruct the browser to call a second URL.
- *http://<Server Name>:<Port if not 80>/showface/login.htm* that will return to the server finally the real View Object which is the login.jsp page.

So through this redirect action it has been shown how Spring MVC internally handles requests and which of its components are involved and what their part in this procedure is.

5.3. View Layer

Concerning now the view layer things are much simpler than the other 2 layers. In order to generate the view to the user html is mainly used. Of course in order to enrich pages with other features and make them dynamic jsp code is also used to dynamically adjust the view as well as jsp, spring and display tags that make certain things easier to implement. With the use of the tags the code is cleaner and can be reused in multiple pages as imports avoiding code duplication. Finally css is used to improve the looks of the pages. A sample jsp page of Show Face application will be presented in order to explain in more detail the above. The presented jsp page is "images.jsp" and it is used to show the user the photos he has uploaded as thumbnails. Also it provides the functionality of deleting them or to view them in a new tab full sized, check screenshot 10 to see how it really looks.

```
<%@ include file="/WEB-INF/jsp/include.jsp"%>
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>


<style type="text/css">
table.test {
	background-color: #f8f8ff;
	text-align: center;
}
</style>

<html>

<head>
<title><fmt:message key="images.page.title" /></title>
<style>
.success {
	color: green;
}
</style>
</head>

<body>

<h5 class="success"><c:out value="${model.success}" /></h5>

<h1>All My Images</h1>

<h3>Here you can see all your Images</h3>
<c:out value="${model.info}" />
<display:table name="model.photosTable" id="testit" pagesize="5"
sort="list" requestURI="/viewMyImages.htm" class="test">

    <display:column title="Image Name" property="imgname"
sortable="true"/>

	<display:column title="Image">
		<i><a
href="<%=request.getContextPath()%>/imageRetriever.htm?img=${testit.id}
&full=yes" target="_blank">
			<img
src="<%=request.getContextPath()%>/imageRetriever.htm?img=${testit.id}&
full=no"/>
		</a></i>
	</display:column>

	<display:column title="Delete Image">
		<i><a
href="<%=request.getContextPath()%>/deleteImage.htm?img=${testit.id}">
			<img
src="<%=request.getContextPath()%>/css/grafics/delete.png"/>
		</a></i>
	</display:column>
```

```
</display:table>
</body>
</html>
```

Now starting from the top there are 2 imports. The first one calls the “include.jsp” page that contains the imports of the basic tags that will be used and it looks like this:

```
<%@ page session="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

The first 2 ones are jsp tags and the next 2 are Spring specific tasks. All of them are used to ease things up as it will be shown. The second import in “images.jsp” page is a very useful tag library called displaytag that makes very easy the presentation of tables with dynamic contents as well as its has built in paging functionality when correctly configured.

Then something like this follows the2 imports:

```
<style type="text/css">
table.test {
	background-color: #f8f8ff;
	text-align: center;
}

</style>
```

This is css styling used to give a specific appearance to the table that will be created afterwards. Other important tags related parts are:

```
<fmt:message key="images.page.title" />

<c:out value="${model.info}" />
```

These are both jsp tags that are imported by “include.jsp” as it was previously shown. The first one shows the title of the page by referencing to a key value. When this will be compiled by the web server in order to be sent to the browser as html code this is what happens. This key value will be checked in a file named messages.properties, Spring knows where to resolve these values as in the xml configuration there is this entry:

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource"
>
        <property name="basename" value="messages"/>

    </bean>
```

So by checking this key in the messages.properties values it will find:

```
images.page.title = View All My Images
```

And thus this value will be passed to the html page. The advantages of this is that you can make easily such changes without having to search the whole code since all these are accumulated in this properties file. Also this way it is possible to support internationalization which means the ability to see the whole site in another language. Show Face does not support this functionality but can be easily done. For example Spring can be configured to search such messages in 2 files one for Greek called messages.gr.properties and one for English messages.en.properties. Then based on the locale of the system or if the user wishes to change it, the appropriate messages of each language will be used.

The second entry that was noted above is used to easily present a dynamic value that came from the controller action to the html page. This c:out will output the real value to put into the “model” Map for the key “info”. To make it more understandable let’s see the code in the real action controller that sets this value:

```
Map<String, Object> myModel = new HashMap<String, Object>();
String info = "You have no Photos uploaded";

myModel.put("info", info);

return new ModelAndView("images", "model", myModel);
```

As it is obvious a HasMap is created and an entry is inserted where the key is called “info” and has as value the contents of the String variable created above. Then when returning the ModelAndView (as it was shown in the flow of a request in Spring MVC) three arguments are used, the first one is the logical name of the View Object that was analyzed previously and the second one is the name of the map that will be used in the view layer and populate it with dynamic data and the third argument is the real Map as an object. This way with using the c:out tag it is much easier to present dynamic data sent from the server in the html final view.

The last interesting thing to present in the view layer is the use of the displaytag that makes presentation of dynamic tables very easy and also with minimal configuration it is possible to add more features like sorting of selected columns and pagination which is really important for performance other than practical issues, here is the particular point of interest to analyze:

```
<display:table name="model.photosTable" id="testit" pagesize="5"
sort="list" requestURI="/viewMyImages.htm" class="test">

<display:column title="Image Name" property="imgname" sortable="true"/>

</display:table>
```

As it is obvious the specific tag prefix “<display:” is used in the beginning of the expressions and then use the specific functionality for the creation of a table element. The first attribute in the expression is `name="model.photosTable"` this shows to the dislpaytag library where to find the info that will populate the table. In the respective controller action that provides the view layer with dynamic data as shown for a previous tag there are the following related lines:

```
while(iter.hasNext()) {
                Object[] data = iter.next();

                TableData tblData = new TableData();
                tblData.setId(new Long(data[0].toString()));
                tblData.setImgname(data[1].toString());
                newList.add(tblData);
```

```
            }
myModel.put("photosTable", newList);
return new ModelAndView("images", "model", myModel);
```

The data that will be passed to the displaytag are populated as shown and to be more specific a class TableData has been created which is a Pojo. This class has an id and an imgname attribute and these are possible columns for a row that will be presented by displaytag. All these TableData objects accumulated in an ArrayList are the rows of the table. As previously shown this data is passed to the view layer through the ModelAndView object. Now the table in displaytag knows where to search for them since the name attribute has been defined.

Next attribute of the tag is `id="testit"` this way it is possible to use the testit variable as a reference to the TableData object directly through jsp code if there is the need to do something that displaytag cannot do and you need this value to be processed by jsp code.

Attribute `pagesize="5"` is used to say that pagination will be used and that the size of each page that will presented to the user will contain 5 rows of the table, in order to see the next page with other 5 rows he will have to navigate there by pressing the link that is generated by displaytag.

Attribute `sort="list"` is used to tell to displaytag that the implantation of sorting in the columns that will support it will be done by itself and not by an external custom sorter. Displaytag has built in support to automatically sort most well known object types, external sorting can be used to achieve custom results.

Attribute `requestURI="/viewMyImages.htm"` is used for the feature of pagination. With pagination as explained the user will see the first 5 rows of the table, in order to see the next fine ones, if they exist; the link to next page should be used. But in order to do this and fetch the next five rows dislpaytag has to know from which controller to request this data and that is why this attribute is used.

Attribute `class="test"` is used to link this specific table with the css style that was defined in the begining of the jsp page, to give it specific attributes on how it will be presented.

Now after all the attribute of display:table tag have been discussed let's check the next display tag provided above named display:column that is used as it is obvious to define the attributes of a column.

Attribute `title="Image Name"` simply defines the name of the column as it will be presented to the user.

Attribute `property="imgname"` is used to tell the column which exact attribute of the TableData object will be shown in the specified column. In the presented case this column is assigned to show the attribute imgname of the TableData POJO.

Attribute `sortable="true"` finally tells diplaytag whether this column should be sortable or not.

All the info on how to use displaytag and to get the jar that provides all these and more functionalities have been obtained by its official site [7].

6. Application Package Structure and Design

In this section the java package organization of the application will be presented. Also it will be explained why it was done this way and the benefits this has. A small reference to the database model will also be made since it is related directly to the domain objects of the application. Hibernate is an ORM (Object Relational Mapping) framework (as previously noted) which directly maps java objects to database tables, for example Account.java object is connected directly to the Accounts table in the db. Finally a brief description of what each java action controller does will be provided, this way it will be easier for someone to read the code or make a change.

6.1. Package Structure

In order to start with the package overall structure a screenshot will be provided that depicts the overall schema. This schema only concerns the java files, of course there are other files too like jsp, css, tag libraries and configuration files but for them their packaging is more or less standard in the final deployment war file that is generated. The war (web archive) file is a standardized structure of the files that is used by web servers to deploy an application. So here is the java package structure (19).

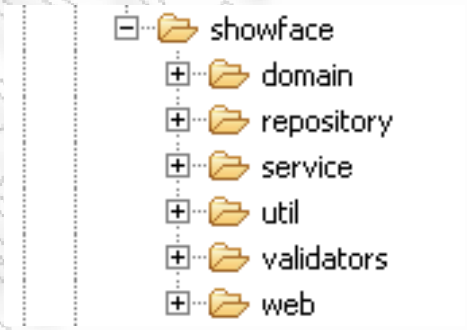


Figure 19. Java Package Structure

As it is shown the java classes have been separated in 6 packages. Of course this has been done for a reason as all classes in a package have similar functionality or serve the same layer. This way it is easier to keep java code organized, it is easier to make changes and

find a file with a specific functionality once the meaning of each package/layer is understood.

Starting with the domain layer, here are its contents in the screen below (20)

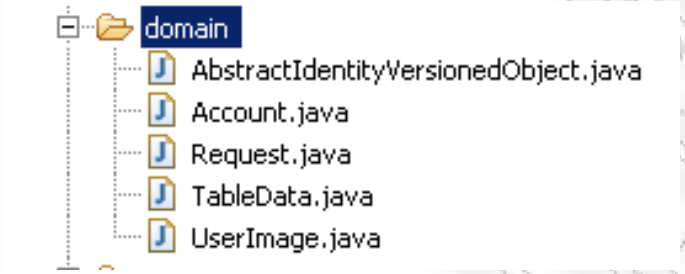


Figure 20. Contents of Domain Package

These classes provide the domain level, the basic units that the application uses to store and handle its data. To be more precise these are:

- The Account class that holds the info for an account, it has an account's attributes as class variables, like name, surname password etc. Also the Account object is directly related to the Account table in the database with Hibernate's help and more specifically with ORM (Object Relational Mapping).

- The Request class models the relationship between 2 Accounts in the application. For example a Request that correlates 2 users can be in 3 statuses. Pending if one of the users has sent a friend request to the other and the other has no yet replied (in order to update the status of their connection). Accepted if a user has sent a friend request and the other user has accepted it. Rejected if a user has sent a friend request and the other one rejected it. As in the case of the Account this object also represents a table in the database, but also has some extra helper fields in order to present data. For example the attribute `verbalStatus` that shows the relation between 2 accounts but with a String representation for the user to view it. In the database level though it is more effective to just use an integer instead of a whole String to model three different states.

- The UserImage class holds all the data related to a picture that is uploaded in the application. These can be the image's name, the owner of the picture, the data (bytes) of the image and also the data of the image's thumbnail. All these info of course also map directly to the third database table of the application that holds

the pictures. Now it is worth to note that the data of the image's thumbnail are also kept in the database when a user uploads an image and are not generated on the fly when a thumbnail is presented. This has been done like this in order to improve the performance of the application when the user browses a page where there are a lot of thumbnails that have to be loaded. This way by using a little more database space for each image, the application's performance has been greatly improved.

- The AbstractIdentityVersionedObject is a class that is extended by all the above 3 domain classes and this has to do with the database. This class has fields that are common for all the database tables used in this application so instead of adding these attributes to every class, they are created once and are extended by all the domain objects used for database persistence. More specifically the first one is the id of the model object. Since each model object is directly mapped to the database (class attribute per column) the id of an entry is almost obligatory in all relational db models so that is the job of the id attribute of AbstractIdentityVersionedObject class that is given to all the domain objects that extend it. Then the second attribute is called version and is also very important. It shows the state of the model object in the database, for example the Request class has some attributes and these are stored in the database as an initial version. If the class itself changes by adding a new attribute or by changing the type of an existing one, it is now different from what it was and this is a new version. This helps to understand data inconsistencies or stale data because of a new version. It is very helpful in the stages of development where changes may happen often to adjust the domain objects to the application's requirements and points out inconsistencies that elsewhere would be really difficult to spot.

- Finally the TableData object is not really used for persistence in the db but it is a helper class that is used for the displaytag library. This helps with the data presentation in the view layer and is used there to present tables in an easier way.

Then the next package that will be examined is the repository package, here is a screenshot.

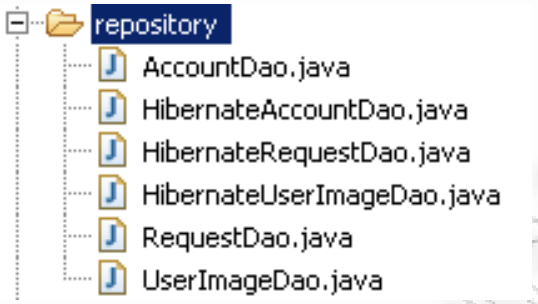


Figure 21. Contents of Repository Package

As it is shown in this package there are the DAO files of the application. The Data Access Object design pattern provides a technique for separating object persistence and data access logic from any particular persistence mechanism or API. There are clear benefits to this approach from an architectural perspective. For example if you want to change the way data access has been implemented it is very easy to be done by changing only these files. If this pattern was not followed it would be more difficult to implement such a change, it could require restructuring the whole application.

Now to be more specific in this case there are the HibernateXXXXDao files and the XXXXDao files. So there are 3 such pairs and this happens because use of interfaces is done. The XXXXDao files are the interface classes that provide the available methods that can be called to the classes of other layers. The HibernateXXXXDao files implement the interfaces defined in the interface classes. So there is a pair of classes for managing accounts in the database, another one for managing requests where as explained this shows the connection between 2 accounts and finally the classes for managing images in the databases. As it has already been noted this is done by using Hibernate (the implementation of the Dao files), and more details will be given in the next chapter as to how this exactly happens.

Now let's move to the next package which is the Service package.

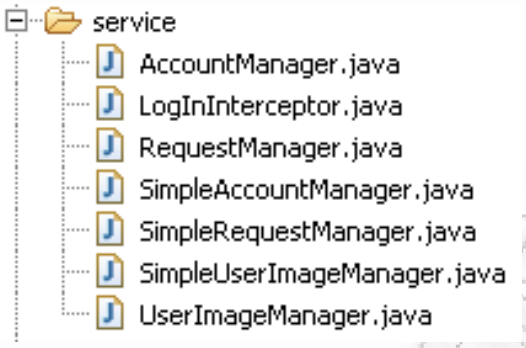


Figure 22. Contents of Service Package

This package is used in order to gather all the possible services that can be provided to the controller layer. So here again the interface design pattern is used for the Manager classes that contain all possible services that may be required for accounts, requests and images. These managers connect to the dao files via the interfaces previously noted in the repository layer, but apart from db access services they may provide other ones like sorting, calculating something etc.

Apart from the manager classes there is also here the LogInInterceptor file that is used to check if a user that requests something is signed in the application. More details of how the interceptor work will be presented in the next chapter.

Moving on to the utils layer here is a screenshot.

Figure 23. Contents of Util Package

This package as its names implies is used for utility classes that may provide a general service for the application and are usually out of the main business scope. In this case ThrowableRenderer is a utility class that is used with log4j logging mechanism. This class helps to present in a better way exceptions in the log files thus helping in the developing and maintaining of the application.

The next package that will be checked is the validators package.

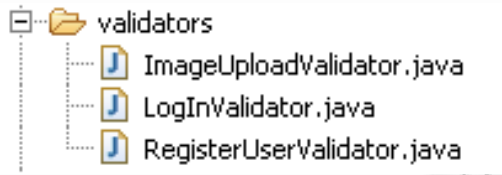


Figure 24. Contents of Validators Package

In this package the classes used for validating user input data are kept. These classes are connected to the respective controller classes. For example in the page where the user registers himself to the application some data have to be filled in the form. Then by submitting this request the appropriate controller will be chosen to fulfill the request. But when there is the need to check the data provided (if the email is valid or a field was left empty etc) the validator kicks in to make these checks and then control is given to the actual controller. This way all such actions requiring user input data validation are kept separated from the other parts of the code and this has its obvious advantages regarding separation and organizing of the code and the responsibilities each class has. The assigning of the validator class to the controller is made by the xml configuration file and here is such a case in the application.

```
<bean name="/registerUser.htm"
class="showface.web.RegisterAccountFormController">
        <property name="sessionForm" value="true"/>
        <property name="commandName" value="account"/>
        <property name="commandClass" value="showface.domain.Account"/>
            <property name="validator">
            <bean class="showface.validators.RegisterUserValidator"/>
        </property>
        <property name="formView" value="registeruser"/>
        <property name="successView" value="login.htm"/>
        <property name="accountManager" ref="accountManager"/>
   </bean>
```

As it is shown in the configuration of `RegisterAccountFormController` that performs the register action there is the validator property that assigns the validator of this controller and in this case it is the RegisterUserValidator class. In Show Face application there are 2 more validator classes as it can be seen from the package screenshot. The

LogInValidator that checks user input data in the login page and the ImageUploadValidator that checks input data of the form user fills to upload an image.

Finally here is the web package of the application.

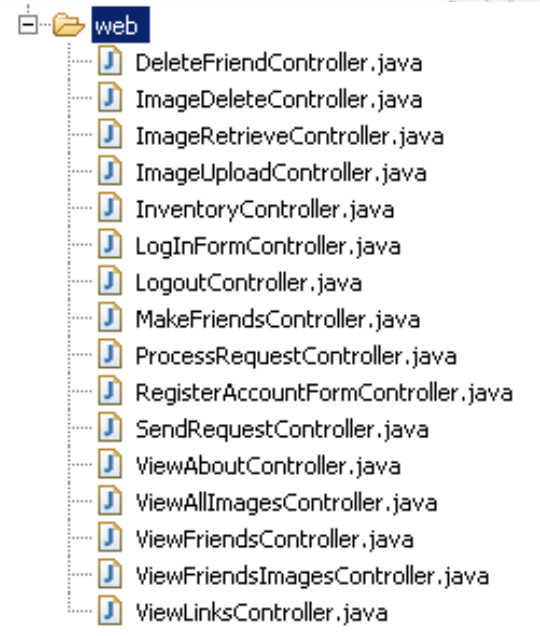


Figure 25. Contents of Web package

In this package as it can be deducted by the naming convention of the classes all the controllers of the application are kept. As previously described these classes are assigned with a job to do when the user request is sent to the server and it has been explained how this works in Spring MVC section. Below a table is provided with each controller (16 in total) and a brief description of what it does.

a/a	Controller Class Name	Brief Description
1	DeleteFriendController.java	Controller that performs the deletion of someone as your friend
2	ImageDeleteController.java	Controller that performs the deletion of an image
3	ImageRetrieveController.java	Controller that performs the retrieval of image data as a byte array, it returns thumbnail or full image depending on request parameter
4	ImageUploadController.java	Controller that performs the uploading of an image
5	InventoryController.java	Controller that populates the home page of the user with data required, 5 latest photo links, pending request info
6	LogInFormController.java	Controller that performs the login action
7	LogoutController.java	Controller that performs the logout action
8	MakeFriendsController.java	Controller that populates the Make Friends page with the available users that you can send a friend request
9	ProcessRequestController.java	Controller that performs actions on a Request, accepts a request or deletes a request in case it is rejected
10	RegisterAccountFormController.java	Controller that performs the registration of a new user to the system
11	SendRequestController.java	Controller that sends the request to a user which is in state Pending initially

12	ViewAboutController.java	Controller that populates the About page
13	ViewAllImagesController.java	Controller that populates My Photos page, it constructs all the image links and the deletion links too
14	ViewFriendsController.java	Controller that populates View Your Friends page, constructs 5 latest images links for each Friend, the deletion link and the link to see all friends images
15	ViewFriendsImagesController.java	Controller that populates the page with all the images of one of your friends
16	ViewLinksController.java	Controller that populates the Links page

7. Main Concepts and Integration Points-Configuration

In this part since the main structure has been presented from an object oriented view, more details will be presented about configuration and integration issues of the technologies used. First Hibernate will be examined regarding its setup and integration with the application. The transaction management mechanism will then be presented. More details on Tiles 2 will be given and explain how they help by building the web pages in a modular way. Finally the login interceptor setup and use will be shown.

7.1. Hibernate Setup and Use

Since Hibernate is the data access framework that is used in the application, the first thing that is required is the datasource that it will use to perform all its operations, in other words info so as to connect to the database. In this application in order to decouple database specific info and configuration from the application, the database has been configured as a jndi resource in the Apache Tomcat server configuration. To be more specific in Tomcat's (version 6) conf folder in the file context.xml the following has been added:

```
<Resource name="jdbc/MyDB"
            auth="Container"
            type="javax.sql.DataSource"
            username="sa"
            password=""
            driverClassName="org.hsqldb.jdbcDriver"
            maxActive="20"
            maxIdle="10"
            url="jdbc:hsqldb:hsql://localhost"
/>
```

So in this part a jndi resource with name "jdbc/MyDB" is configured, its type is a DataSource which shows that this is a sql database, the username and password for db access are provided as well as the database specific driver. In this case the database is an HSQL database so the driver required for the connection is "`org.hsqldb.jdbcDriver`". This driver exists in hsqldb.jar file that must be provided to the server in its lib folder in order to be able to make the connection. Properties maxActive and maxIdle have to do with the maximum number of active connections that can exist in the same time and with the maximum number of idle connections that can be available. Finally the url of the

database is provided, this is the database's location that in this case is localhost since the server and the database are on the same machine.

All this info about the database connection exists on the server's configuration as a jndi resource. In the application the only thing that is needed in order to acquire a reference to this database is the configuration of the datasource bean based only on its jndi name and here is how this is done in spring's configuration xml:

```
<bean id="dataSource"
class="org.springframework.jndi.JndiObjectFactoryBean">

      <property name="jndiName" value="java:comp/env/jdbc/MyDB"/>
</bean>
```

As it is shown with just a few lines and based on the full jndi name *"java:comp/env/jdbc/MyDB"* the reference to the database is obtained and this is the only dependency with the database without caring about its type, url etc.

Now that is clear how the reference to the datasource is obtained, let's examine the main configuration part of Hibernate, which is the configuration of the hibernateSessionFactory bean.

```
<bean id="hibernateSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

      <property name="dataSource" ref="dataSource"/>

      <property name="mappingLocations">
                  <list>
                        <value>classpath*:*.hbm.xml</value>
                  </list>
      </property>



      <property name="hibernateProperties">
                  <props>
<prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
                  </props>
      </property>
    </bean>
```

This bean is injected in all the HibernateXXXXDao files that were examined in the previous chapter since they provide the Hibenrate's session in order to perform actions on the database. Back to the configuration now the first property is the datasource and it is connected with the datasource bean that was previously shown. This is how Hibenrate is now aware of the database info in order to perform actions on it. Next property is mappingLocations and shows to a list of files that have the ".hbm" extension. These files map the domain classes (Account, Request, UserImage) to the database tables, this is how Hibernate manages to give populated java objects from database tables. Last property is "hibernate.dialect" that is a specific sql dialect that hibernate uses(HSQL), an example will be shown later.

An ".hbm" file will now be presented to show how this object-relational mapping is defined.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping default-lazy="false">
<class name="showface.domain.UserImage" table="IMAGES">

<id name="id" column="IMG_ID" type="long" unsaved-value="-1">
      <generator class="sequence">
                <param name="sequence">PHOTOSEQ</param>
    </generator>
</id>

<version name="version" column="version" unsaved-value="null"
type="long"/>
<property name="name" column="NAME" not-null="true"/>
<property name="contents" column="CONTENTS" not-null="true"/>
<property name="thumb" column="THUMB" not-null="true"/>
<many-to-one name="ownerid" column="OWNERID"
class="showface.domain.Account" not-null="true"/>
</class>
</hibernate-mapping>
```

In this configuration file that is used by Hibernate the domain class UserImage is mapped to the database table IMAGES `<class name="showface.domain.UserImage" table="IMAGES">`. Then each of the class attributes is mapped to a database column. For example the id attribute is mapped to the database column IMG_ID, moreover for this case a SEQUENCE (an id generator in sql level) is created in the database and is associated with this column in order to dynamically generate the id values for each new

entry. A last thing that is worth noting is that Hibernate can also handle more complex issues like many to one relationship cases and more. Such configuration files also exist for the other 2 domain classes: Account and Request.

All the above had to do with configuration/integration issues of Hibernate with Spring. Now that it is ready to be used from the HibernateXXXXDao classes it will be demonstrated how easy it is to access and alter data with a few lines of code. All the HibernateXXXXDao classes extend `HibernateDaoSupport` class and they implement their corresponding interface. By extending this class provided by Spring it is very easy to access the database. First thing to note here is that HibernateDaoSupport requires a SessionFactory according to the documentation of the class and this is why there is the need to inject the hibernateSessionFactory bean to all the HibernateXXXXDao files of the application. This is simply done like this:

```
<bean id="userImageDao"
class="springapp.repository.HibernateUserImageDao">
        <property name="sessionFactory" ref="hibernateSessionFactory"/>
</bean>
```

Now the full class of HibernateUserImageDao will be provided as an example in order to show how it works and to pinpoint some features this kind of implementation has.

```
public class HibernateUserImageDao extends HibernateDaoSupport implements UserImageDao {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    public void saveUserImage(UserImage usrimg) {
        logger.info("Saving image: " + usrimg.getName());
        getHibernateTemplate().saveOrUpdate(usrimg);
    }

    public void deleteUserImage(UserImage usrimg) {
        logger.info("Deleting image: " + usrimg.getId());
        getHibernateTemplate().delete(usrimg);
    }

    public List<UserImage> getUserImageById(Long id){
        logger.debug("getUserImageById");
        Long[] crits = new Long[1];
        crits[0] = id;
        return (List<UserImage>)getHibernateTemplate().find("from UserImage image where image.id = ?", crits);
    }

    public List<byte[]> getUserThumbById(Long id){
        logger.debug("getUserThumbById");
        String query = "SELECT image.thumb from UserImage image where image.id=?";
        Long[] crits = new Long[1];
        crits[0] = id;
        return (List<byte[]>)getHibernateTemplate().find(query, crits);
    }

    public List<byte[]> getUserFullImgById(Long id){
        logger.debug("getUserThumbById");
        String query = "SELECT image.contents from UserImage image where image.id=?";
        Long[] crits = new Long[1];
        crits[0] = id;
        return (List<byte[]>)getHibernateTemplate().find(query, crits);
    }

    public List<Object[]> getImgIdByUserId(Account userid){
        logger.debug("getImgIdByUserId");
        String query = "SELECT image.id, image.name from UserImage image where image.ownerid=?";
        Account[] crits = new Account[1];
        crits[0] = userid;
        return (List<Object[]>)getHibernateTemplate().find(query, crits);
    }
}
```

So it is obvious in the class declaration in the beginning what was previously discussed about extending the HibernateDaoSupport class and implementing the interface which is UserImageDao in this case. The first method called saveUserImage is just 3 lines long and is used to save a new UserImage object in the IMAGE table in the db. With a closer look it is clear that all the job is done by one line "`getHibernateTemplate() .saveOrUpdate(usrimg);`". This HibernateTemplate is provided by the class that is extend and makes inserting an object very easy.

Unlike most traditional operations of accessing a database in this case there are several things that should be pointed out. First of all there is nowhere in the code statements to open and close the db connection, then there are not any obligatory, as in other cases, try catch statements when accessing a database, no manual sql code is written in order to perform the insert of the new object and finally no sign of statements or annotations about transaction management. All these lead to a clean and readable code whose only concern is to define what is needed by the database and does not have to worry about all the above. All these are handled by Spring and Hibernate based on the configuration and setup that was provided. As for the transaction management a configuration has also been provided but it will be examined right after.

Of course in cases where the whole object is not needed from the database but only the value of an attribute/column in order to reduce I/O actions and improve performance, specific queries in HSQL can be written as it is shown in method getUserThumbById and in some others.

7.2. Transaction Management with AOP

After checking how the data access framework operates in the context of the application concerning database I/O transactions, it is also very important to see how these transactions are managed.

To begin with transaction management let's see where it is needed based on a simple example. Imagine that there is a method that transfers money from one bank account to another. In order to perform such an action two update statements in the table with the bank accounts should be done. One to remove the money from the first account (decrease account's balance) and one to add the money to the second account (increase account's balance). In case the first update is executed and then there is a network problem for example and the second update is not executed, the money of the first account is lost. This is an inconsistent case that could cause big problems. In cases like this a good

transaction management strategy is essential. In a managed transaction environment if something like the above had happened the transaction would roll back. This means that if the first update is executed and the second fails, all the changes made by these 2 updates which are in the same transaction will be reverted and everything would be as before. In case though both updates were successful then the transaction would be committed which means applied in the database.

In Show Face application the transaction management strategy is applied with the help of AOP (Aspect Oriented Programming) expressions which are built in Spring. All the transaction handling can be configured using AOP expressions in Spring's configuration xml thus having centralized control of transaction management and keep java code clean.

Fist of all the bean transactionManager is defined and the necessary attributes to handle transactions are injected to the bean, the sessionFactory which is the hibernateSessionFactory and the datasource which was defined previously in this chapter.

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
<property name="sessionFactory" ref="hibernateSessionFactory"/>
<property name="dataSource" ref="dataSource"/>
</bean>
```

The next thing that will be checked is the AOP configuration which is based on a pointcut and an advice. In AOP pointcut is a set of joint points which means specific points in the program's code execution. Advice is this extra behavior that is applied in the specified pointcuts. In order to perceive this better the rest of the show face case will be presented. Below is the part that defines the pointcuts (specific points in the program) where the extra behavior (transaction management) will be applied.

```
<aop:config>
<aop:advisor pointcut="execution(* *..AccountManager.*(..))" advice-
ref="txAdvice"/>
<aop:advisor pointcut="execution(* *..RequestManager.*(..))" advice-
ref="txAdvice"/>
<aop:advisor pointcut="execution(* *..UserImageManager.*(..))" advice-
ref="txAdvice"/>
</aop:config>
```

As it is shown three pointcuts are defined. The points of interest are the execution of any method in any one of these three interfaces AccountManager, RequestManager, UserImageManager. These are the "places" of interest because all database operations begin from these classes and this is where the transaction should start. Moreover it can be noticed that after each pointcut is defined, it is also assigned to an advice that will perform the extra actions. In all three cases the advice of the pointcuts is "txadvice" that will be shown below.

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="*save*" propagation="REQUIRED"/>
<tx:method name="*delete*" propagation="REQUIRED"/>
<tx:method name="*get*" propagation="SUPPORTS"/>
</tx:attributes>
</tx:advice>
```

This specific advice that is followed by all the pointcuts of the application is a transaction advice used for transaction management. The transactionManager bean is also provided to the txadvice configuration. Then there are the attributes that say: if the name of the method contains "save" or "delete" then a transaction is required, if the name of the method contains "get" then transaction is supported. In Show Face there are 2 different cases. When a user's action changes data in the database, the database related actions (related to the same user) should be in one transaction in order to be able to roll back in case of any problem. The same thing applies for methods that contain save or delete in their method-name, in such cases a transaction is required and if one does not already exist a new one is created. The other case is for methods that contain "get" in their name, these methods do not alter database data they just retrieve data. So transaction rules are more flexible in this case where transaction is supported. If for a client request a transaction already is running the "get" method execution will be made in that transaction else no transaction will be initiated since there is no danger for such methods ("get") to cause data inconsistencies.

7.3. Apache Tiles 2

Apache Tiles 2 is used in the application and it is integrated with Spring's MVC viewResolver component. This is done because of the benefits that Tiles offer and result

in the modularity of the web pages since they are separated in parts. What is more important is that the page is separated in parts which can also be re-usable thus decreasing the amount of same page parts that would otherwise have to be re-written. This pattern that Tiles 2 use is called "The Composite View Pattern" and allows to create pages that have a similar structure, in which each section of the page vary in different situations [10]. This will be better explained by the following example.

Tiles configuration had been quickly presented in the part where a request in Spring MVC was analyzed, but in this part some more details will be provided. Here is again the configuration that is required in Spring's xml files in order to integrate Tiles with the viewResolver.

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
       <property name="definitions">
            <list>
                <value>/WEB-INF/tiles-defs.xml</value>
            </list>
        </property>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass"
value="org.springframework.web.servlet.view.tiles2.TilesView"/>

</bean>
```

The first bean named `"tilesConfigurer"` is used in order to configure the final page view and contains a property that shows to the tiles-defs.xml file. In this file the configuration and layout of the jsp pages is included and each view's logical name is mapped finally to the real view object. The second bean named `"viewResolver"` is the one that integrates Spring's viewResolver with Tiles.

The main concept in Tiles is to separate each page in reusable parts and keep a certain layout. In Show Face application there are 2 main layouts called `"main.layout"` and `"main.layout.pre.login"`. Both of them define the main layout of the application with the difference that the fist one is used when a user has logged in and the other one is used before the login or after the logout. The configuration of these 2 layouts in tiles-defs.xml file is described like this:

```
<definition name="main.layout" template="/WEB-INF/jsp/main-layout.jsp">
<put-attribute name="title" value="Title of this page...should be
set!"/>
<put-attribute name="header" value="/WEB-INF/jsp/header.jsp"/>
<put-attribute name="menu" value="/WEB-INF/jsp/menu.jsp"/>
<put-attribute name="body" value="/WEB-INF/jsp/empty-body.jsp"/>
<put-attribute name="footer" value="/WEB-INF/jsp/footer.jsp"/>
</definition>


<definition name="main.layout.pre.login" template="/WEB-INF/jsp/main-
layout.jsp">
<put-attribute name="title" value="Title of this page...should be
set!"/>
<put-attribute name="header" value="/WEB-INF/jsp/header.jsp"/>
<put-attribute name="menu" value="/WEB-INF/jsp/preloginmenu.jsp"/>
<put-attribute name="body" value="/WEB-INF/jsp/empty-body.jsp"/>
<put-attribute name="footer" value="/WEB-INF/jsp/footer.jsp"/>
</definition>
```

As it is shown in the first layout the template page main-layout.jsp is used. This page contains the attributes title, header, menu, body and footer that are inserted in the page as defined in the definition of main.layout. The second definition called main.layout.pre.login is exactly the same and adds the same attributes except from the case of the menu attribute where another jsp is inserted. This is done because the menu of a user cannot be the same before-after the login action. Now since the 2 main layouts of the application have been created these are used all the time in order to present the various views of the application. For example after a successful login action the logical name of the view that will be requested will be “home”, in that case the viewResolver will find in tiles-defs.xml file the following:

```
<definition name="home" extends="main.layout">
      <put-attribute name="title"  value="home.page"/>
      <put-attribute name="body" value="/WEB-INF/jsp/hello.jsp"/>
</definition>
```

This means that in order to present the view "home" that the main.layout definition will be used since it is extended by the home definition (`extends="main.layout"`). So the page created according to main.layout's definition will be presented, but 2 attributes will be overwritten by those described in the "home" definition. The title of the page will be updated and also the body will contain a new jsp called "hello.jsp" which contains the data of the home page.

The same happens for all the other views since the menu, header, footer do not have to change every time and can be the same. This way the application keeps a consistent layout when presented to the user and certain jsps like menu.jsp, header.jsp etc are written just once and can be easily re-used thanks to Tiles.

7.4. Log In Interceptor

In this section the use of the login interceptor will be presented. In order for the user to access a page of the application a request has to be made but whether or not this request should be served depends on the user's state. If the user has signed in the application then all the services of the application should be available, if not only some. So in order to check this for every request that is made this means that this code check should be added in the beginning of all the 16 controllers of the application, which is not a good solution taking into consideration time required, code repetition, difficulty of change in such an implementation.

In order to solve this problem in an optimal way, avoiding all the above issues, the use of an interceptor in Spring's MVC handlerMapping configuration is required. As it was described in previous chapter concerning the Spring MVC, the job of the handlerMapping is to find the correct controller to serve a request based on the url of the client's request. Here is how the interceptor is configured first as a bean:

```
<bean id="loginInterceptor"

class="springapp.service.LogInInterceptor">

</bean>
```

And then how it fits in the overall handlerMapping configuration in Spring's xml:

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
>

<property name="interceptors">
<list>
<ref bean="loginInterceptor"/>
</list>
</property>


<property name="mappings">
<props>
<prop key="/**/login.htm">/login.htm</prop>
<prop key="/**/registerUser.htm">/registerUser.htm</prop>
<prop key="/**/hello.htm">/hello.htm</prop>
<prop key="/**/priceincrease.htm">/priceincrease.htm</prop>
<prop key="/**/uploadimage.htm">/uploadimage.htm</prop>
<prop key="/**/imageRetriever.htm">/imageRetriever.htm</prop>
<prop key="/**/logOut.htm">/logOut.htm</prop>
<prop key="/**/viewMyImages.htm">/viewMyImages.htm</prop>
<prop key="/**/deleteImage.htm">/deleteImage.htm</prop>
<prop key="/**/makeFriends.htm">/makeFriends.htm</prop>
<prop key="/**/sendRequest.htm">/sendRequest.htm</prop>
<prop key="/**/processRequest.htm">/processRequest.htm</prop>
</props>
</property>

</bean>
```

As it can be seen in the configuration file the first property is used to define the list of possible interceptors. In this application there is only one that was previously defined and called loginInterceptor. The next property called mappings helps the handlerMapping to find the correct controller based on the url of the request.

Now that an interceptor is defined what will happen is that whenever a request comes to the handlerMapping it won't be passed to the appropriate controller (via the DispatcherServlet) but it will be passed to the loginInterceptor. The loginInterceptor will do its checks and if it returns true (the user is logged in) then the suitable controller will be selected to fulfill the request, else in cases of false a redirect to the login page is

requested in order for the user to sign in. The initial request in case it required sign in will not be served if the user is not signed in.

Finally let's take a look at loginInterceptor's code:

```
public class LogInInterceptor extends HandlerInterceptorAdapter {
    /** Logger for this class and subclasses */
    protected final Logger log = Logger.getLogger(getClass());

    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler) throws Exception {
        log.debug("Inside LogInInterceptor" + request.getRequestURI());
        HttpSession sess = request.getSession(false);
        if(request.getRequestURI().equals("/showface/login.htm")
                || request.getRequestURI().equals("/showface/registerUser.htm")
                || request.getRequestURI().equals("/showface/viewAbout.htm")) {
            return true;
        }

        if(sess!=null) {
            Account logedin = (Account)sess.getAttribute("LogedInAccount");
            //log.debug("Account : " + logedin.toString());
            if (logedin!=null) {
                return true;
            } else {
                response.sendRedirect(request.getContextPath() + "/login.htm");
                return false;
            }
        } else {
            response.sendRedirect(request.getContextPath() + "/login.htm");
            return false;
        }
    }
}
```

In the declaration of the interceptor class as it is shown HandlerInterceptorAdapter should be extended, then the method preHandle of the extended class is implemented. In this

method if the url of the request is related to the login, register or about page then immediately true is returned without any further checks since these pages do not require log in to be accessed. For all the other cases the HttpSession is checked in order to find if there is a logged in account (when a user logs in his/hers account is put in the HttpSession). If the user account is found in the HttpSession this means that a log in has been performed so access to the specified request is allowed. In case the account is not found then the user is redirected to the login page in order to sign in and the request that was sent is not served since false is returned.

So with this simple and clean implementation an application universal login check has been added and can be managed just by one class, avoiding complexity and code duplication.

8. *Presentation of a Request's Full Round Trip*

Now since every aspect of the application has been explained a full round trip of a request will be presented in detail. This way most of the things described will be encountered in their physical order in a real example. The example that will be shown refers to what happens when the user presses the "Upload Image" button as shown in picture (7). In this specific case the user uploads a new image in Show Face application.

The data required to upload the image have to be completed as in screenshot (7). They are a name for the image and the full local path of the image file. Then user presses the "Upload Image" button and the following request is sent to the server: *http://<Server Name>:<Port if not 80>/showface/uploadimage.htm.*

At this point the request will be handled by Spring MVC once it reaches server side (Controller Layer). First of all the Dispatcher Servlet will have to handle control to the login Interceptor that was previously explained, every request has to be checked first there, this is done to check whether the specific request requires log in or not. In this case the uploading of an image requires the user to be logged in. If the user is not logged in or the session has expired then the user is sent to the login page else the flow continues normally. After a successful log in check by the interceptor control is given again to the Dispatcher Servlet that will now have to find to which controller to delegate this request. In order to do this it asks the HandlerMapping and as shown before this mapping exists in its configuration:

```
<prop key="/**/uploadimage.htm">/uploadimage.htm</prop>
```

This means that the controller that should handle such requests is this one:

```
<bean name="/uploadimage.htm"
class="showface.web.ImageUploadController">
<property name="sessionForm" value="true"/>
<property name="commandName" value="userimage"/>
<property name="commandClass" value="showface.domain.UserImage"/>
<property name="validator">
<bean class="showface.validators.ImageUploadValidator"/>
</property>
<property name="formView" value="uploadImage"/>
<property name="successView" value="hello.htm"/>
<property name="userImageManager" ref="userImageManager"/>
</bean>
```

As it can be seen the controller class attribute shows the real controller file that should handle this request, ImageUploadController. There are also other info and parameters like the validator (checks user input data) class, the success view (the url that will be asked if the request is successfully completed), the form view (the page the user sees and fills to submit) and the page data mapping to a java domain object UserImage, that will be populated by the user's input data.

In this specific case that the suitable controller has been found the Dispatcher Servlet will first pass control to the validator in order to check user's input and if no errors are found only then the control will be passed to the controller to serve the request. Here is the validator class.

```java
package showface.validators;

import org.apache.commons.logging.Log;

public class ImageUploadValidator implements Validator {

    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());

    public boolean supports(Class clazz) {
        return UserImage.class.equals(clazz);
    }

    public void validate(Object obj, Errors errors) {
        UserImage pi = (UserImage) obj;
        if (pi == null) {
            errors.reject("error.null-Image");
        }
        else {

            if(pi.getName()==null || pi.getName().length()<1) {
                errors.rejectValue("name", "required");
            }
            if(pi.getContents()==null || pi.getContents().length<1) {
                errors.rejectValue("contents", "required");
            } else if (pi.getContents().length>2097152) {
                errors.rejectValue("contents", "image-toobig");
            }

            logger.debug("TESETSET" + pi.getContents().length);

        }
    }
}
```

The first thing that is checked is if the model object that is associated with the upload image form is null or not. In that case a general error is added to Spring's Error object. Then the name of the image is checked in case it is not set. Now for the image file there is a check to see that the file that will be uploaded is not a zero sized file (which obviously is an error). An extra check is added to limit the maximum size of an image in order to avoid cases of abuse (in Show Face the limit is 2 MB per image). In any case that an error is found it is added in the Error object. This object if it contains any errors it will be automatically used by Spring MVC and will populate the form with error messages next to the erroneous fields like in screen (8) (this is also achieved thanks to Spring's Tag

library that is used in the jsp page). For example for attribute “name” of the UserImage class there is the following in the jsp to make the mapping and shows the errors:

```
<tr>
<td align="right" width="20%">Name :</td>
<td width="20%"><form:input path="name" /></td>
<td width="60%"><form:errors path="name" cssClass="error" /></td>
</tr>
```

In case though that no error is found in these checks, Dispatcher Servlet now delegates control to ImageUploadController that is presented below.

```
37    protected ModelAndView onSubmit(
38            HttpServletRequest request,
39            HttpServletResponse response,
40            Object command,
41            BindException errors) throws ServletException, IOException {
42
43        // cast the bean
44        UserImage bean = (UserImage) command;
45        byte[] file = bean.getContents();
46        try {
47            BufferedImage image = ImageIO.read(new ByteArrayInputStream(file));
48            Image resized = image.getScaledInstance(75, 52, Image.SCALE_FAST);
49            log.debug(resized.getWidth(null));
50            BufferedImage bufferedImage = new BufferedImage ( 75,52,BufferedImage.TYPE_3BYTE_BGR);
51            bufferedImage.createGraphics().drawImage(resized, 0, 0, null);
52            ByteArrayOutputStream out = new ByteArrayOutputStream();
53            ImageIO.write(bufferedImage, "jpeg", out);
54            bean.setThumb(out.toByteArray());
55            Account loggedin = (Account)request.getSession(false).getAttribute("LogedInAccount");
56            bean.setOwnerid(loggedin);
57            userImageManager.saveUserImage(bean);
58            out.close();
59        } catch (Exception e) {
60            log.error(e);
61            errors.rejectValue("contents", "img-invalid");
62            Map model = errors.getModel();
63            return new ModelAndView("uploadImage", model);
64        }
65        return new ModelAndView(new RedirectView(getSuccessView()+"?success=Your Image Has Been Uploaded"));
66    }
```

This is the main method of the Controller that will handle the request; its name is onSubmit and is inherited by Spring's SimpleFormController that all the controller classes of Show Face extend. The job of this controller is to get the user input data that is already mapped in the UserImage domain object and make the necessary transformations (generate the image thumbnail and set the owner of the image). After the UserImage bean is fully populated (this is finished in line 56 of the above image) then the controller will access the model layer in MVC architecture in order to persist the data in the database. This is done in line 57 of the above image. At this specific point the control is given to the back end layers of the application through the use of the interface class UserImageManager of service package. The interface method that is called is saveUserImage and this above is its implementation in service package:

```
public void saveUserImage(UserImage usrimg) {
userImageDao.saveUserImage(usrimg);
}
```

This is a method in service package whose only job is to save the UserImage model object to the database and no other actions are required. That is why the only method called is saveUserImage of interface class UserImageDao of the repository package, so control goes even deeper in the application's architecture. In this part also the transaction manager mechanism is taking care of the transaction. As it was shown in the configuration of the transaction manager every method of class UserImageManager that is called and contains the word "save" as this one, requires a transaction within which it will be executed. In case any problem comes up during the saving operation the transaction will roll back and no stale or inconsistent data will be kept in the database.

Now in the repository package since there is the implementation of the persistence layer of the application Hibernate will simply save the image in the database given the configuration provided as shown in the previous chapter. Here is the implementation of saveUserImage method in class HibernateUserImageDao:

```
public void saveUserImage(UserImage usrimg) {
logger.info("Saving image: " + usrimg.getName());
getHibernateTemplate().saveOrUpdate(usrimg);
}
```

Thanks to Hibernate the image is saved in the database with just one call. If everything has been executed successfully control returns from repository to service package and from there back to the ImageUploadController (web package). There since everything is ok and its job is complete it will return through a redirect action the final view to the user. In the final view that will be returned, the user will see the home page of the application with an additional image-upload-success message on its top as shown in screenshot (9).

This way the whole round trip of a request has been presented. The whole flow has been presented by showing the various layers/packages of the application to take the control in their real order to serve the request. Then control returns back the same way to the controller that will give through a redirect action the final view to the user.

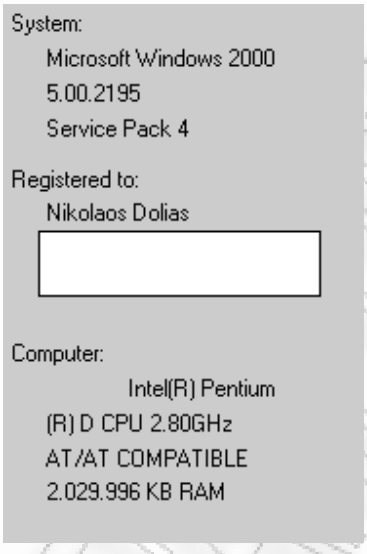
9. Performance Metrics

In this chapter since the whole architecture and technological features of the application have been thoroughly presented, it is now time to see what all these offer performance wise. What are the response times of the application with normal load? How much can it be stressed given the available hardware for the tests that will be conducted? How it uses system resources? Such type of questions will be answered and statistical data will be provided in this chapter.

9.1. Jmeter Setup and Tests

For testing this J2EE application the first tool that will be used is the well known in Java community jmeter tool. Jmeter is a tool that can be used to measure performance of both static and dynamic resources which include of course Show Face application. This tool can be used to simulate various loads (heavy or not) on a server, network or object to test its strength or to analyze overall performance under different load types [11].

All the tests are executed on the following Desktop PC:



This info is provided in order to show in which hardware context the tests will be performed, it is just a simple dual core windows PC and not a dedicated server that uses any high performing operating system (eg linux, solaris). Also all the components are being executed on the same machine (database server, apache tomcat, jmeter application, profiling tool) so the machine is already consuming a lot of resources to accommodate all

of them. In real life things are different with dedicated servers and so on but despite this fact such tests can still provide valuable data.

Apart from the info on the hardware certain tweaking actions took place to improve the application's performance regarding the system setup. The java memory heap size has been increased for Tomcat server in order to be more durable against heavy load testing and to avoid out of memory exceptions. Further on Tomcat's configuration has been tweaked a little in 2 files. In Tomcat's conf folder in file context.xml the maximum number of concurrent database connections has been increased to 50 (intial value 30) and the maximum number of pooled (idle) database connections has been increased to 30 (initial value 10). The other file in conf folder that has been changed is server.xml, there the maximum number of active threads that Tomcat can spawn has been set to 260 in order to be able to serve heavy loads.

To start with the testing of the application a test case has to be defined on jmeter and various configuration actions have to be done in order to create a good test case. To begin with the first test a screen is provided that shows the normal load test plan and it will be explained later on.

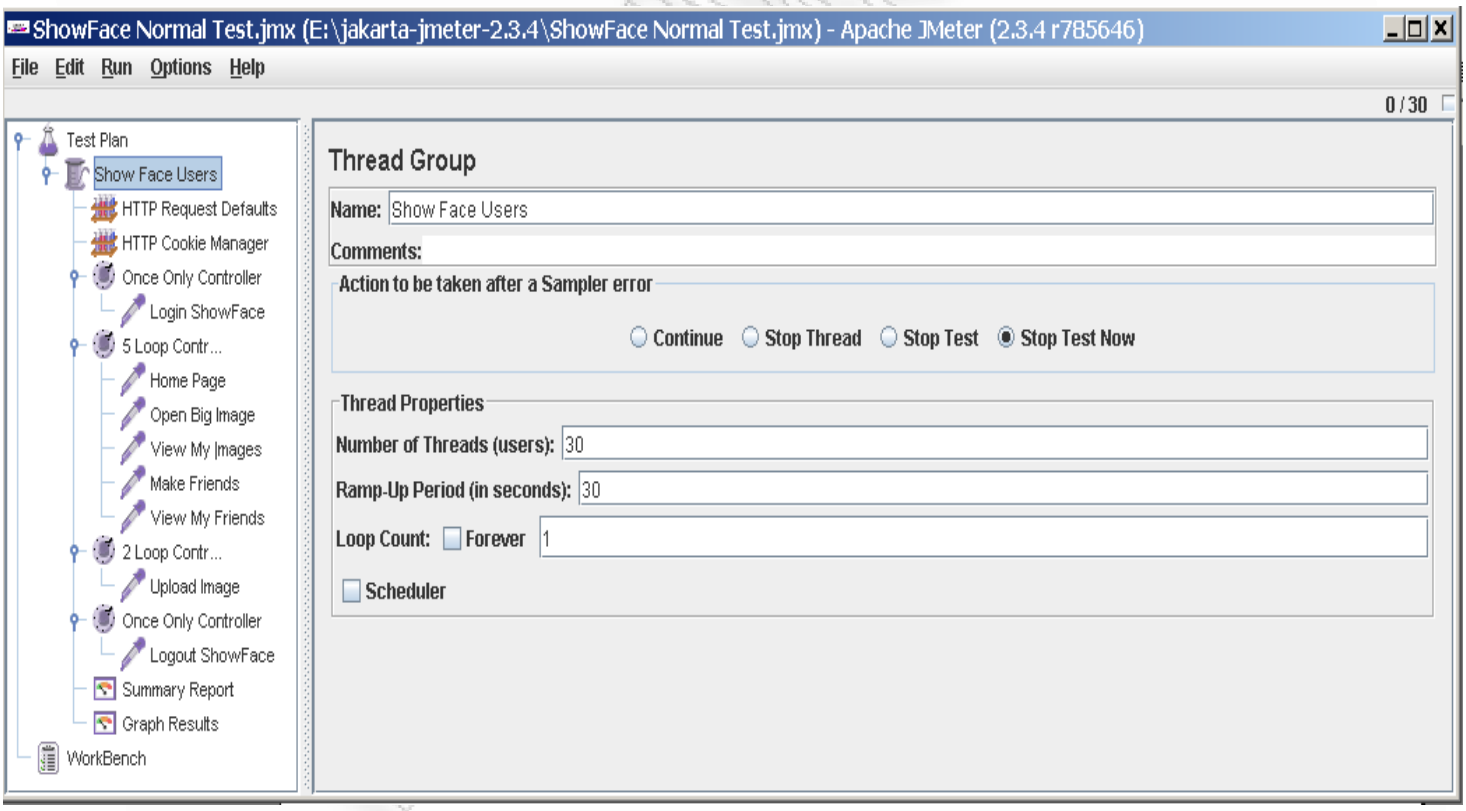


Figure 26. Normal Load Test Case

In the above image (26) jmeter and the normal load test case is presented. On top of the tree view is the element Test Plan that contains all the elements of a test. Right below this the element named Show Face Users is a Thread group. This element is for example the user or users that will perform the actions that are defined in the test. For this normal load test case 30 users (threads) have been used that will perform for one time all the requests defined below them, moreover within a period of 30 seconds all the threads/users will have started sending requests (if this attribute was 0 all 30 threads would start requests at the same time but this does not simulate well real life situations).

The main elements of the test case are all children of the Show Face Users element. The first one named HTTP Requests Defaults is used to configure the server URL that will be used in order to perform requests; this is a useful element in case the application is tested in another system, then only the server parameters will be changed there since the action specific URLs will be the same. The next element named HTTP Cookie Manager is used to handle cookies sent by server in order to keep the session etc in order to behave like a browser.

After this there are the loop controller elements. These elements are used in order to execute once or repeat more times the requests that are defined as their children in the tree view. The first controller that executes only once is used to send the log in request since it has to be done only once. Then there is a loop controller that repeats for 5 times the five requests it contains. These are the home page of the application, the opening of a big image, the request for view my images page, for make friends page and for view my friend's page. So each user after login executes 5 times these 5 requests in the test case. Then the upload image action is also tested but only for 2 times (to avoid cleaning the database every time a test is executed from hundreds of images) and finally the logout happens once as the login. Summing all these up this test case will make 30 users * 29 requests, which means 870 requests in total with no pause between them (even more stressful than real life cases).

Finally the last 2 elements named Summary Report and Graph Results are used to gather statistical info and create graphs that will be presented after the test has been executed. These types of elements are called listeners in jmeter. Now from the menu bar, once the application on Tomcat and the database server are up and running, select Run and then Start. After waiting some time for the test to finish, check that there are no exceptions (in the application log or database server). If that is the case it means that the load has been handled with no problem.

So after the above is done let's examine the Summary Report that was generated based on the application's performance (27).

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Login ShowFace	30	296	51	729	190,53	0,00%	1,0/sec	3,44	3470,6
Home Page	150	160	14	544	134,58	0,00%	4,3/sec	14,63	3471,3
Open Big Image	150	178	65	550	90,87	0,00%	4,3/sec	7043,07	1677248,0
View My Images	150	66	9	410	71,72	0,00%	4,3/sec	20,54	4888,3
Make Friends	150	148	12	692	149,77	0,00%	4,3/sec	9,16	2187,0
View My Friends	150	366	31	1139	274,61	0,00%	4,3/sec	31,96	7643,0
Upload Image	60	1808	651	2903	558,78	0,00%	1,6/sec	5,61	3501,2
Logout ShowFace	30	189	19	679	172,81	0,00%	51,4/min	3,72	4451,0
TOTAL	870	300	9	2903	472,68	0,00%	23,2/sec	6634,64	292831,4

Figure 27. Results of Normal Load Test Case

In the first column as it is shown there are the various requests defined in the test case. These requests cover almost all the application's actions so that the test is as close to a real case as possible. The next column shows the number of the requests for each case. Then the Average, Min and Max response times of each request in milliseconds are shown. As it can be seen the average response times are really good since almost all of them are under 300 milliseconds while in real life an acceptable limit would be 2 seconds for most cases, in these cases the response times are very fast, the only exception is the Upload Image request but this is normal since uploads always take more time. The total average response time for the test is just 300 milliseconds which is a very good result.

Then there is the standard deviation column which is relatively big but this depends on the load of the application, for example the initial requests were served faster than the later ones but this is normal for a web site while its load increases. Even the max average response times though are quick enough taking under 1 second. In the Error column there is no error of course since the application handled the load relatively easy.

The next column shows the Throughput of the application, this shows the amount of requests for each type that the application was able to serve for the given test; these values are expected to be even higher in a more stressful test. The last 2 columns finally have to do with the amount and rate of transferred data between the client (jmeter) and the server. The only notable thing for this case is the size of bytes for the Open Big Image request, this number is too big in relation to the other results but this is normal since in that request a big image is requested from the server.

After checking the report results let's move on to the graph result that is presented below (28).

.

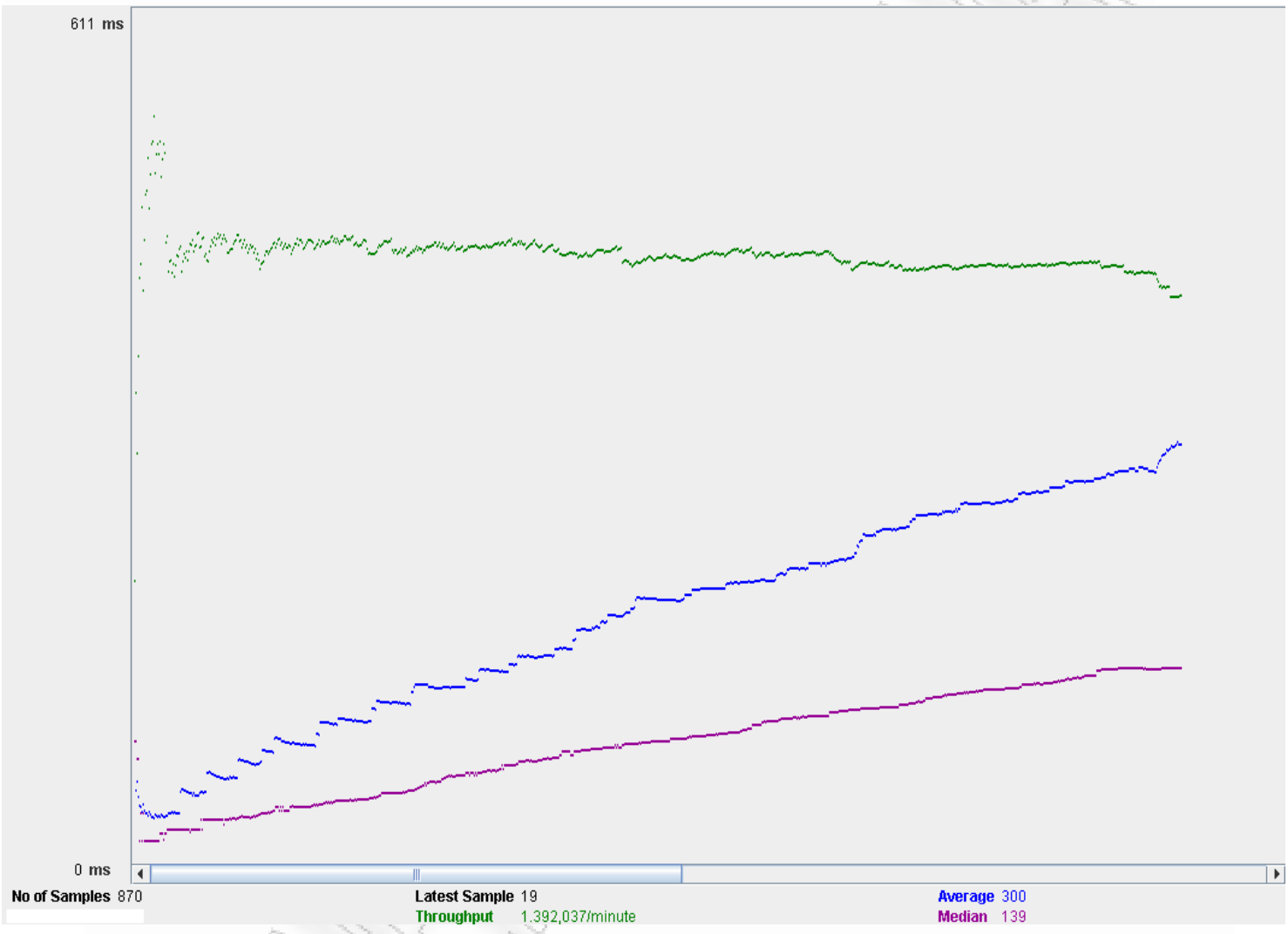


Figure 28. Graph of Normal Load Test Case

In the graph presented there are 3 lines. The green one shows the average throughput of the application in total based on the sample requests. As it can be seen the average throughput is high and relatively stable despite the load increase as the time passes in the test case (more threads/users are spawned to make requests). The blue line shows the average response time in total as the test is executed. Here it is obvious as it was previously stated that the response time increases as the load increases but again it goes up to 300 milliseconds which is very fast. Finally the purple line shows the median value

as the test progresses. The median is the limit that is used to separate the samples of this test in 2 teams that have the same number of requests. To be more precise since the overall median is 139 milliseconds this means that from the 870 samples/requests, half of them were executed in less than 139 milliseconds and half of them in more. This is another interesting statistical measure that shows the performance of the application.

Moving on to the second test something very interesting will be presented that will prove in practice the scalability of the application since the performance has already been shown. The test case will be exactly the same as previous but some simple JVM tuning will be performed to show how the application without any code change but based on setup and hardware configuration can improve its performance.

In the initial test the minimum java heap size that was allocated to Tomcat was set by this environmental variable: CATALINA_OPTS= –Xms256m –Xmx1024m. This means that the initial heap size used by the JVM was 256 megabytes and the maximum 1024. Now the new value that will be used for the next test will be: CATALINA_OPTS= –Xms512m –Xmx512m –Xmn256m. This now means that the initial heap size has been augmented to 512 megabytes and it is allocated from the beginning which is good and the maximum one has been set at the same value, it has been decreased. However there is a new parameter –Xmn which is associated with garbage collector execution to free memory, its correct value requires some testing in order to be selected for an application. In Show Face case 256 megabytes were used. This value mainly cleans the first generation of the groups that garbage collector handles (called Eden Age generation), this generation includes the newly created objects.

In order to begin the test again Tomcat server has to be restarted in order for the new value of CATALINA_OPTS to be taken into account. Also in order to obtain accurate results the test case has to be run once for one user in order for Tomcat to perform first time compilations in jsps when something is requested for the first time after the server has started, after this the response time becomes stable and quicker (this was performed for the first test too). So here is the Summary Report for the second test (29).

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Through...	KB/sec	Avg. Bytes
Login ShowFace	30	25	17	67	9,49	0,00%	1,0/sec	3,50	3470,4
Home Page	150	11	8	23	3,03	0,00%	5,1/sec	17,20	3471,2
Open Big Image	150	76	55	102	13,53	0,00%	5,1/sec	8294,47	1677248,0
View My Images	150	8	6	21	1,76	0,00%	5,1/sec	24,34	4909,2
Make Friends	150	8	7	20	2,05	0,00%	5,1/sec	10,84	2187,0
View My Friends	150	25	20	47	4,72	0,00%	5,1/sec	37,89	7643,0
Upload Image	60	435	294	920	154,97	0,00%	2,0/sec	6,92	3500,7
Logout ShowFa...	30	11	6	27	4,79	0,00%	1,0/sec	4,52	4451,0
TOTAL	870	53	6	920	114,22	0,00%	28,7/sec	8197,27	292835,0

Figure 29. Results of Normal Load Test Case 2

As it is shown above the same test with the same number of requests has been executed, also the average bytes are almost the same since the requests returned the same data. Apart from these two columns that are the same as expected, all the other columns with statistical data have been improved. Total average response times (as well as Min and Max) have been greatly improved, overall response time has been decreased from 300 to 53 milliseconds, this is a very big improvement (of course since both times are too small in practice the difference cannot be seen unless in very heavy load situations). Apart from this the total standard deviation has been decreased from 472,68 to just 114,22 which shows that the application behavior regarding response times is more stable. Again no error cases since such load can be easily handled. Finally total throughput and data rate also improved from the previous test. In general this time execution was much quicker. Below now the Graph Results are presented (30).

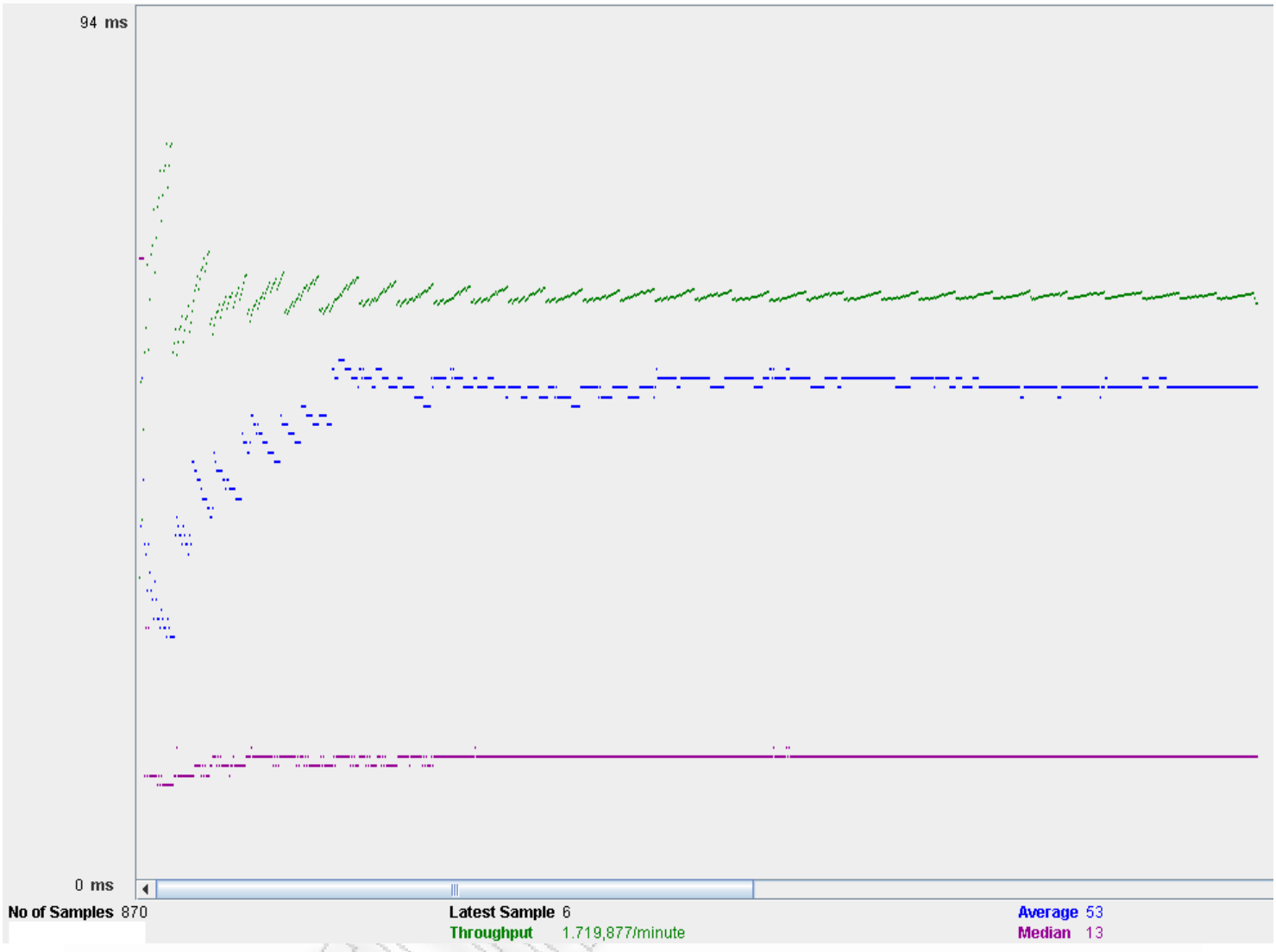


Figure 30. Graph of Normal Load Test Case 2

As it is expected the results are much better in the graph too. Since the application now is quicker the throughput has become more stable as well as the average response time and the median. All 3 lines show very stable behavior. This happens because now it is very easy to handle the load of this test case; it will be interesting to see what will happen when the load increases in the next test that will be performed.

To conclude with the second test, it has been shown that the application is really scalable since its performance has been increased and handles easier the same load without performing any source code change but by just improving resource related attributes like the JVM memory allocation of Tomcat.

Now in the third test there is going to be a much bigger load in the application that looks more like a denial of service attack than a real test case since the load that will be created is too big and it is created within just a minute. To be more precise in this test there will be again the same 29 requests per user without any pause between them (this is not like real life cases since user requests have some pause time between them), but instead of 30 users now there will be 100 concurrent users that within 1 minute they will have send all their requests to the server. This creates a total load of 2900 requests within a minute on a simple desktop pc that hosts the application. Moreover for this test since the database activity will be greatly increased compared to the previous tests there is also the need to allocate more memory to the database server. Unfortunately since available resources are limited only 512 megabytes will be allocated to the database server.

Now after executing the test here is the Summary Report (31).

Label	# Sam...	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Login ShowFace	100	1055	43	2932	646,32	0,00%	1,6/sec	5,53	3483,0
Home Page	500	555	9	2111	446,39	0,00%	5,5/sec	18,60	3483,2
Open Big Image	500	1733	60	8422	2219,19	0,00%	5,5/sec	8930,86	1677248,0
View My Images	500	283	10	1814	270,27	0,00%	5,4/sec	26,33	4962,6
Make Friends	500	421	9	2157	351,99	0,00%	5,4/sec	11,51	2188,0
View My Friends	500	1634	25	4900	1132,45	0,00%	5,4/sec	40,16	7644,0
Upload Image	200	3817	660	8604	1486,77	0,00%	2,1/sec	7,06	3512,9
Logout ShowFace	100	331	5	1607	449,58	0,00%	1,1/sec	4,70	4452,0
TOTAL	2900	1109	5	8604	1485,22	0,00%	29,5/sec	8448,49	292847,9

Figure 31. Results of High Load Test Case 3

As it can be seen 2900 requests have been made and the total average response time is about 1 second which again is a good result given the sudden and big load as well as the limited resources of the test system. Of course such a response time is not bad at all but in a better system it would be much quicker. Also in average response times it is worth noting that the actions that have the 2 biggest delays are those that have to do with database transactions that involve large amount of data like Open Big Image and Upload Image, this has also to do with the limited resources (memory) that were allocated to the database server. Apart from the response times again no error was reported and the throughput rate remained at around 30 requests per second which shows the stability and the limits of the system for this test. Finally the data rate has slightly increased from the previous test. Below also the graph of this test is provided (32).

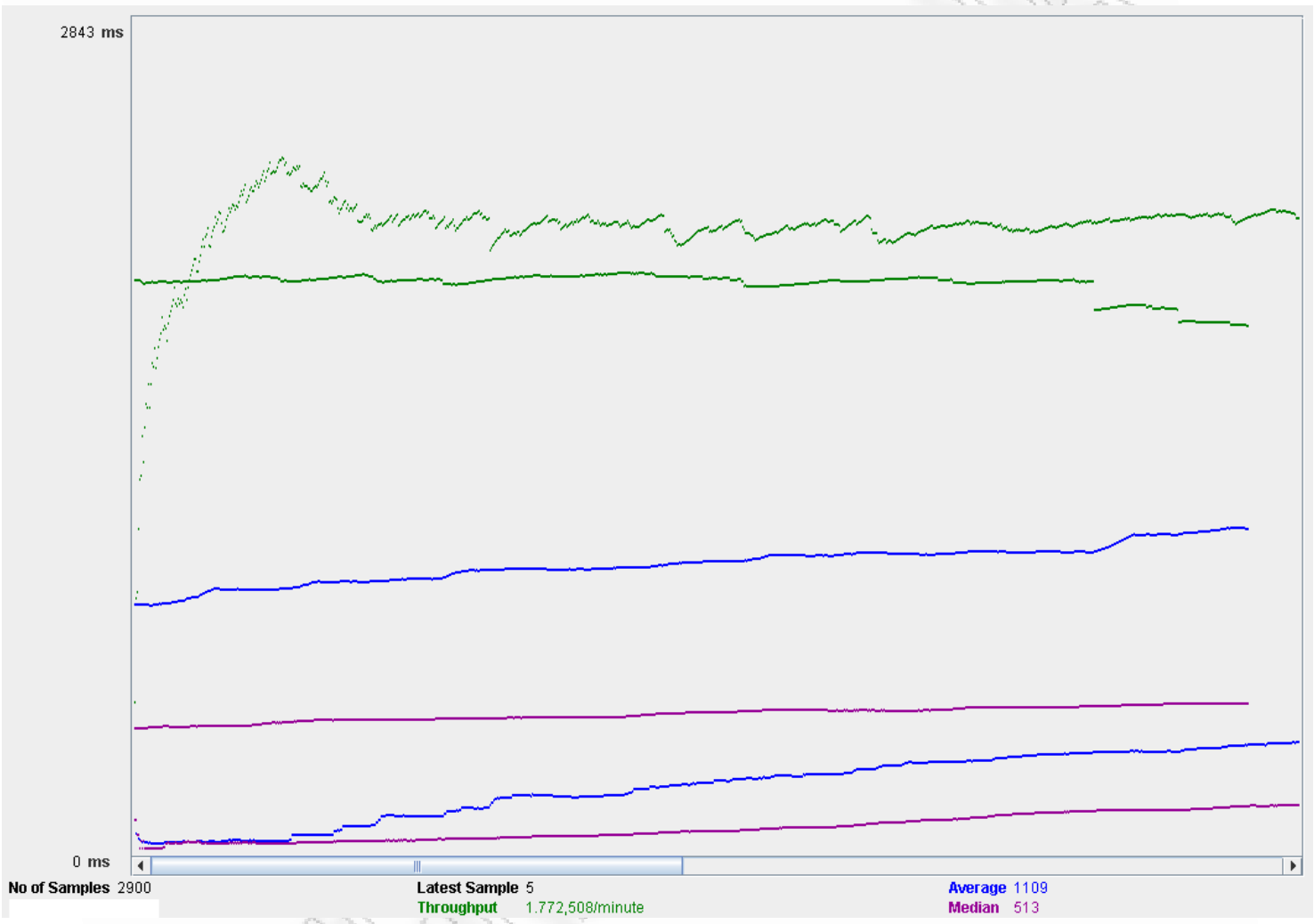


Figure 32. Graph of High Load Test Case 3

In the following test the previous test case will be modified in order to resemble more a real life case. The same amount of requests and users will be used but now between each user's requests there will be a gap of 500 milliseconds also the requests will start within a period of 3 minutes. To sum up the test case: In a period of 3 minutes 100 users will start sending requests (29 per user) and between each user's request there will be a gap of 500 milliseconds. Here are the report results (33).

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Login ShowFace	100	44	14	226	47,90	0,00%	33,6/min	1,90	3467,0
Home Page	500	11	8	122	7,67	0,00%	2,2/sec	7,42	3474,6
Open Big Image	500	121	56	668	73,39	0,00%	2,2/sec	3581,27	1677248,0
View My Images	500	10	6	189	12,83	0,00%	2,2/sec	10,49	4913,0
Make Friends	500	12	6	187	18,44	0,00%	2,2/sec	4,67	2188,0
View My Friends	500	41	19	337	42,40	0,00%	2,2/sec	16,32	7644,0
Upload Image	200	370	291	676	91,90	0,00%	1,1/sec	3,85	3512,5
Logout ShowFace	100	9	4	107	12,05	0,00%	33,9/min	2,46	4452,0
TOTAL	2900	61	4	676	103,21	0,00%	11,9/sec	3407,67	292837,3

Figure 33. Results of High Load Test Case 4

As it can be seen now that the load has been better dispersed in time and is more realistic the performance of the application has dramatically improved. Now the total average response time is only 61 milliseconds, the standard deviation is also small compared to other tests. No errors again and the throughput is now less since the requests are more dispersed in time, the same applies for the data rate. In general despite the big number of concurrent users the application is very stable and performing in a possible real life situation. The stability can also be deducted from the graph of the fourth test case which is provided below (34).

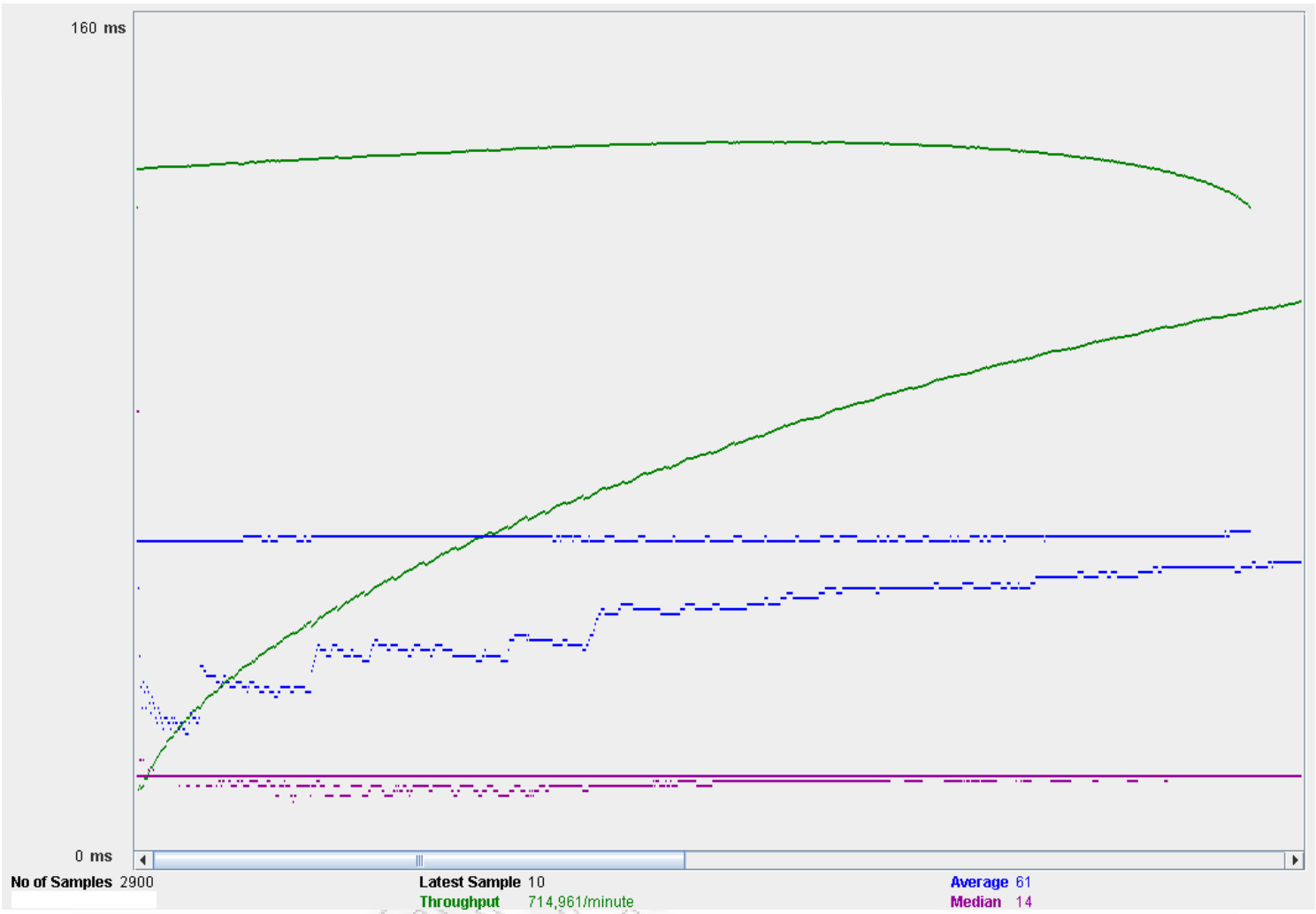


Figure 34. Graph of High Load Test Case 4

Now let's move on to the fifth and final test. In this final test the previous real life test scenario will be used exactly as it is concerning number of requests per user (29), delay between requests of a user (500 ms), time period within all users will start sending requests (3 minutes). The important change is though that this test will be executed for 200 concurrent users instead of 100, this means double load within the same time that requests were dispersed in the previous test. This test will show the application's performance in a really high load real life scenario. Here is the results report (35).

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	KB/sec	Avg. Bytes
Login ShowFace	200	359	14	2798	535,60	0,00%	1,1/sec	3,74	3467,2
Home Page	1000	419	9	2280	435,95	0,00%	4,2/sec	14,15	3474,7
Open Big Image	1000	1020	58	8874	1346,09	0,00%	4,2/sec	6829,69	1677248,0
View My Images	1000	156	6	1878	231,67	0,00%	4,2/sec	20,06	4924,2
Make Friends	1000	271	6	2490	378,77	0,00%	4,2/sec	8,91	2188,0
View My Friends	1000	743	20	3778	768,52	0,00%	4,2/sec	31,13	7644,0
Upload Image	400	1597	298	4879	1008,17	0,00%	2,1/sec	7,21	3512,8
Logout ShowFace	200	109	4	1928	213,57	0,00%	1,1/sec	4,64	4452,0
TOTAL	5800	576	4	8874	856,26	0,00%	22,8/sec	6512,95	292839,3

Figure 35. Results of Very High Load Test Case 5

The first thing note in the table above is that the number of requests doubled as expected to 5800 requests. The total average response time is much bigger than that of the previous test but still it is a great value and taking into consideration the number of concurrent users it is even better. Now the standard deviation has a big value but this is something to be expected as the load on the system was getting bigger and bigger the response times were increasing too. No errors again and the throughput rate is double that of the previous test, this is normal since the number of users doubled. The same goes for the data rate. Finally here is the graph results too (36).

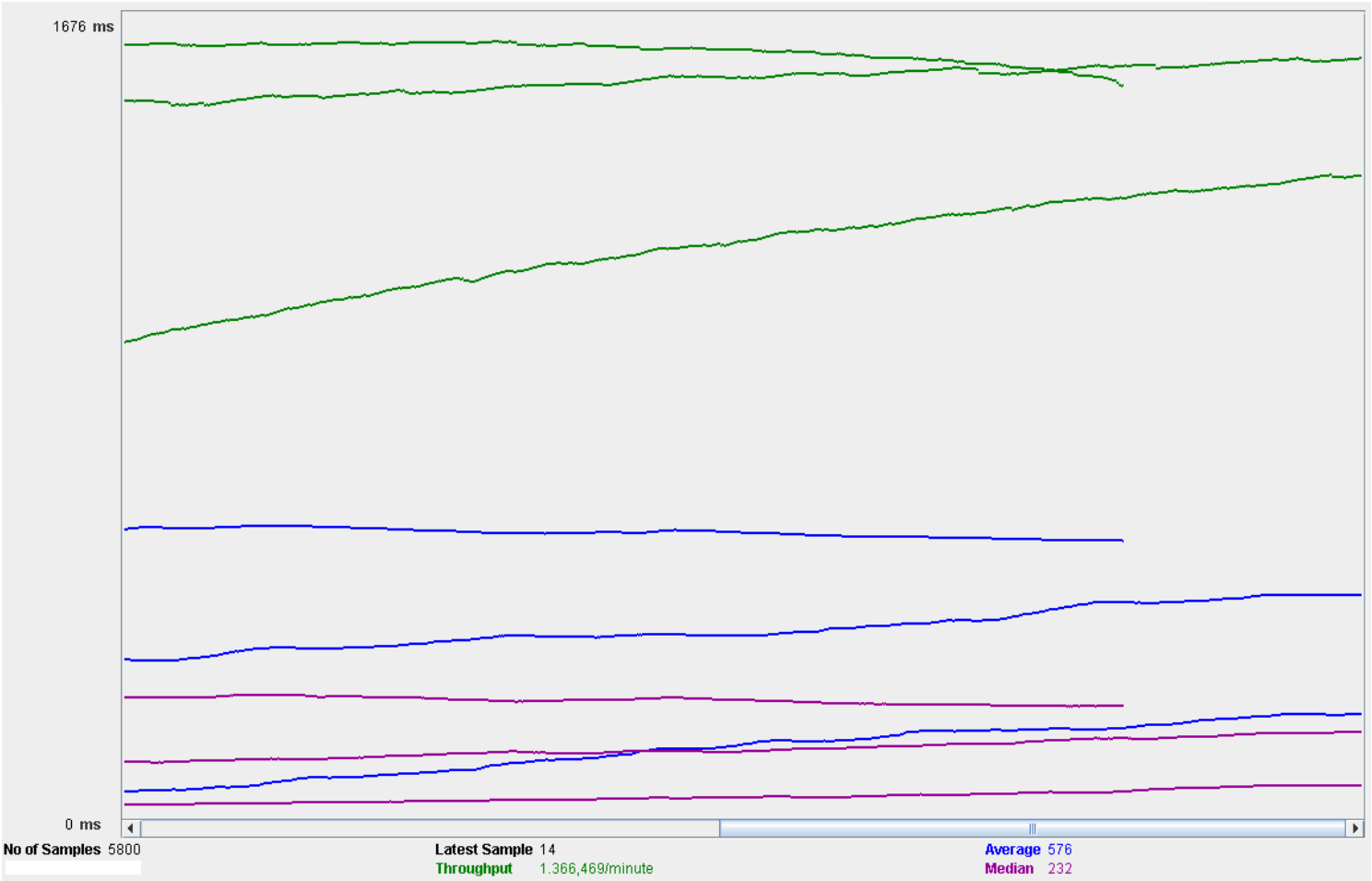


Figure 36. Graph of Very High Load Test Case 5

After all the above 5 test cases and the analysis that was provided on Show Face application results, it has been shown that the architecture and design choices of the application do have an impact on its performance and scalability. High load tests have been executed with successful results and concepts like the scalability of Show Face application have been presented in practice.

9.2. Profiling the Application with the Help of jmeter

Now after measuring the application's performance and behavior under different types of load, more info will be provided on the resources the application needs during runtime and the impact it has on the system. In order to provide such info a java profiling tool will be used. There are various profiling tools but the one that was chosen for this test is named YourKit [12]. This tool was chosen because it seemed really easy to use and integrate with Tomcat for profiling of j2ee applications. It also can be used under various operating systems and provides a nice help section with videos etc.

So how this profiling tool works and what info it provides? After installing the profiling tool there is an option to integrate it with a local web server, amongst the list of options, Tomcat 6 that Show Face uses, is provided. Then the startup.bat file of Tomcat is provided to YourKit and it generates another file to startup Tomcat that includes several options to support profiling. So when Tomcat is started through the edited by YourKit bat file then the server can be profiled by YourKit UI (user interface).

What YourKit does is that while the application runs on Tomcat various info regarding the application's use of resources is recorded and presented for example cpu usage, memory allocation, methods that are called and execution times of methods, hierarchy of calls and many more other data. These are used to monitor an application and find possible problems like memory leaks or method bottlenecks etc. Generally it provides detailed info during the execution time of an application.

In order to present the profiler results for Show Face application something is needed to work on the application to create some load in order for the profiler to be able to capture the data that is needed. Of course in order to achieve this, a jmeter test will be executed while the profiler monitors the application. The test that will be used is test number 4 from the previous section, to be more precise it is a test case with 100 users and 3 minutes ramp up period. Each user makes 29 requests to the application.

So after starting up the database server, tomcat from the edited by YourKit bat file, jmeter and YourKit UI, the profiling from YourKit UI is started then jmeter test is started and wait until all its threads are finished then from YourKit UI stop the profiling and here are the results. First graph (37) has to do with the CPU usage levels that tomcat needed in order to fulfill the 2900 requests that were made by the test case.

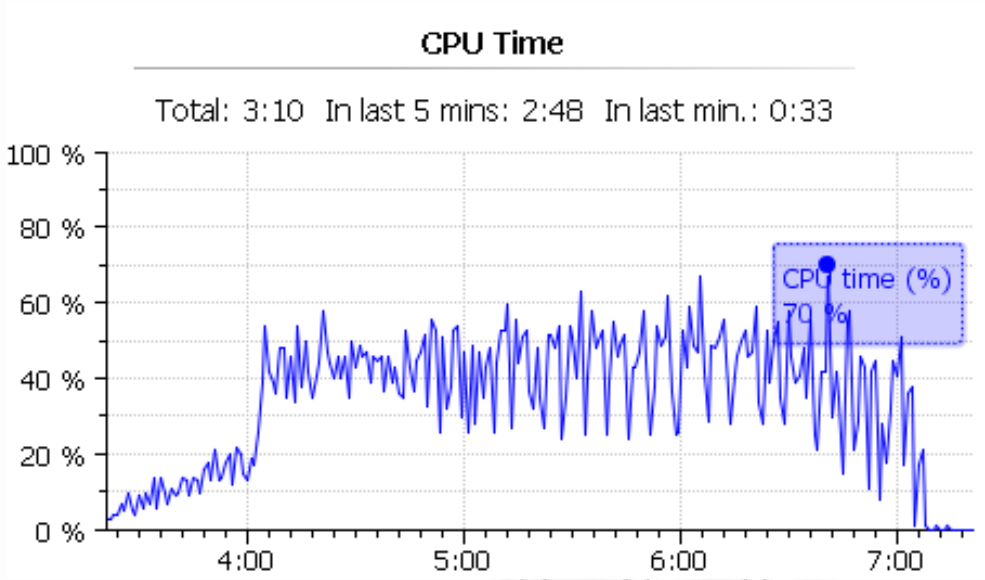


Figure 37. CPU Usage Graph

As it is show during the 5 minutes (approximate time) that were needed for the test to complete the CPU usage level is presented. It is obvious that as time passed and more user threads are spawned by jmeter the load gets bigger and so the CPU usage, of course at the end as the load decreases CPU usage decreases too. Below another graph that shows Tomcat's number of threads in time is also presented (38).

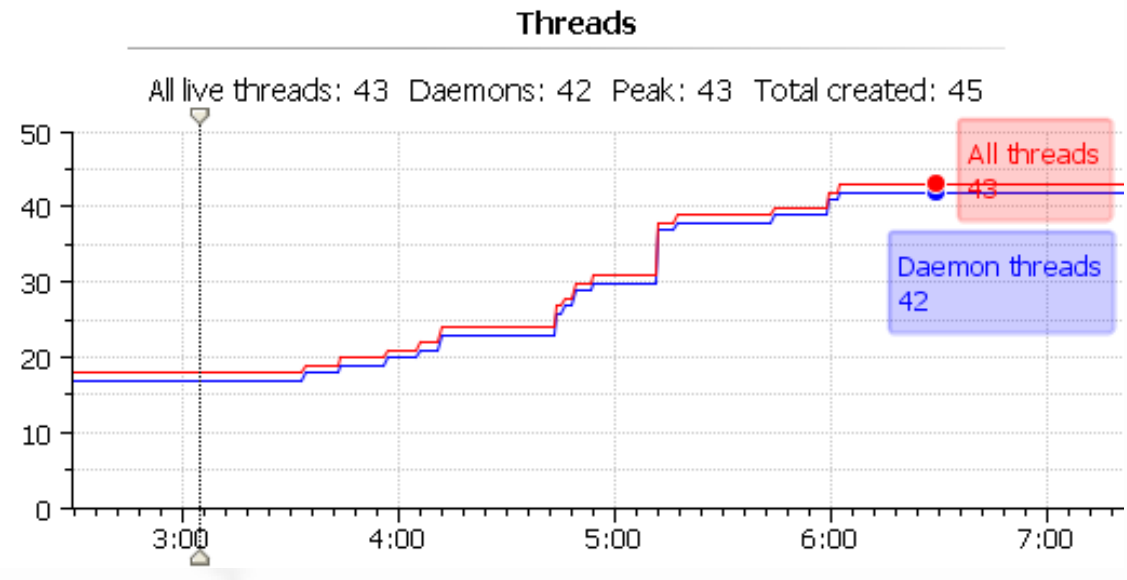


Figure 38. Tomcat's Threads Graph

In the graph above as it is expected the count of threads increases as the load of the application increases in order to be able to serve more requests. Now let's see some statistic results that the profiler tool gathered during the execution of the test. In the following screen (39) there is some info on the SQL queries of the application regarding the total execution time each needed the average execution time as well as the number of times that this query was invoked.

Call	Time (ms)		Avg. Time (ms)	Invocation count
SELECT	30.128	3 %		
select userimage0_.CONTENTS as col_0_0_ from IMAGES userimage0_ where userimage0_.IM	21.999	2 %	43	500
select userimage0_.IMG_ID as col_0_0_, userimage0_.NAME as col_1_0_ from IMAGES userim	3.352	0 %	1	3.300
select account0_.ID as ID0_, account0_.version as version0_, account0_.NAME as NAME0_, .	2.324	0 %	1	2.000
select request0_.REQ_ID as REQ1_2_, request0_.version as version2_, request0_.FRIENDA a	877	0 %	1	800
select request0_.REQ_ID as REQ1_2_, request0_.version as version2_, request0_.FRIENDA a	647	0 %	1	500
select request0_.REQ_ID as REQ1_2_, request0_.version as version2_, request0_.FRIENDA a	483	0 %	0	800
select request0_.REQ_ID as REQ1_2_, request0_.version as version2_, request0_.FRIENDA a	262	0 %	0	500
select account0_.ID as ID0_, account0_.version as version0_, account0_.NAME as NAME0_, .	133	0 %	0	500
select account0_.ID as ID0_, account0_.version as version0_, account0_.NAME as NAME0_, .	46	0 %	0	100
Other	22.305	2 %		
org.apache.tomcat.dbcp.dbcp.**DelegatingStatement.close**()	9.977	1 %	1	9.400
org.apache.tomcat.dbcp.dbcp.**PoolingDataSource$PoolGuardConnectionWrapper.clos**	8.911	1 %	0	9.200
org.apache.tomcat.dbcp.dbcp.**DelegatingStatement.executeBatch**()	2.362	0 %	11	200
jdbc:hsqldb:hsql://localhost	374	0 %	15	24
org.apache.tomcat.dbcp.dbcp.**PoolingDataSource$PoolGuardConnectionWrapper.con**	341	0 %	1	200
CALL USER()	137	0 %	0	200
org.hsqldb.jdbc.**jdbcConnection.createStatement**(int, int)	93	0 %	0	200
call next value for PHOTOSEQ	77	0 %	0	200
org.hsqldb.jdbc.**jdbcStatement.close**()	31	0 %	0	200

Figure 39. Profiler SQL Statistics

In the first tree there are the "SELECT" queries that the profiler recorded and the "Other" tree contains mainly database actions performed automatically by Hibernate-Spring. As it is show execution times are very quick and consist only a small part of the overall execution time.

Moving on the next screen (40) there is info from the profiler on the requests for jsp-servlet files. Again the execution times seem quick enough.

Call	Time (ms)		Avg. Time (ms)	Invocation count
/showface/uploadimage.htm	74.106	8 %	370	200
/showface/viewFriends.htm	36.466	4 %	72	500
/showface/hello.htm	25.800	3 %	32	800
/showface/makeFriends.htm	7.147	1 %	14	500
/showface/viewMyImages.htm	4.343	0 %	8	500
/showface/login.htm	2.283	0 %	11	200
/showface/logOut.htm	78	0 %	0	100

Figure 40. Profiler JSP-Servlet Statistics

Now let's check the memory related graphs produced by the profiler tool (41).

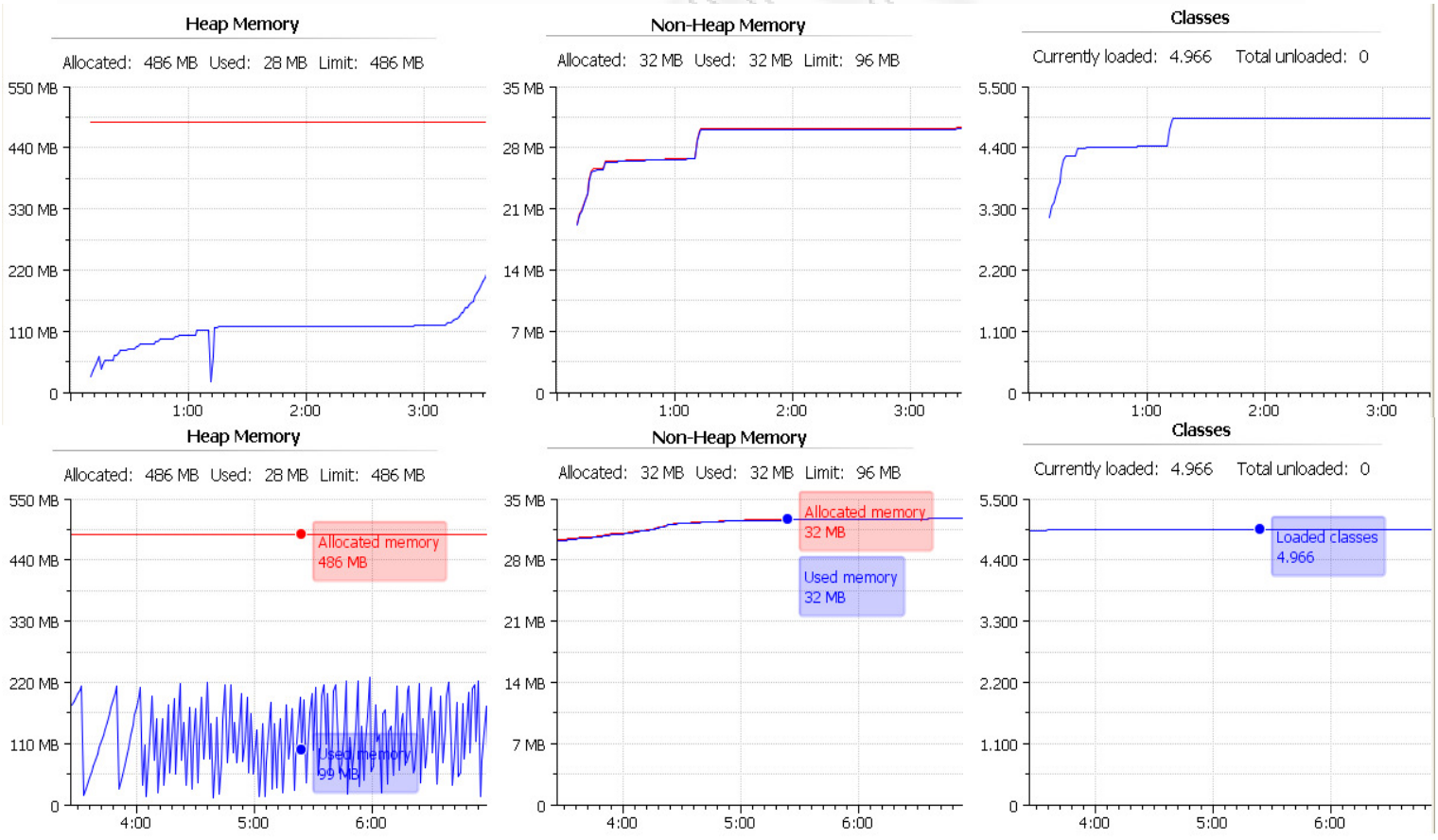


Figure 41. JVM Graphs

In the figure above (41) there are 3 graphs in 2 different time periods. First line has the three graphs from 0 to 3.5 minutes of the profiling period and the second line contains the same graphs from 3.5 minutes to the end of the profiling period. In order to check better the 2 memory related graphs (heap, non heap) some further info on the JVM will be provided. The JVM memory consists of the following segments:

- Heap memory, which is the storage for Java objects
- Non-Heap memory which is used by Java to store loaded classes and other meta-data
- JVM code itself, JVM internal structures etc.

Now taking into account the above heap memory graph, it shows the runtime usage of the heap memory during the application's execution. As it is obvious in the beginning the memory usage is low but it augments as time passes and the jmeter test increases the test threads it spawns to make requests. This makes the application to require more memory and so it does use. Then in the second line graph for heap memory when the memory size reaches around 220 MB the line drops down and increases again all the time creating this pattern that is shown. This happens because of the –Xmn 256m parameter that was added for JVM tunning in CATALINA_OPTS environmental variable. This attribute as it was explained, when the heap size reaches the specified value, it calls the garbage collector to clean up memory releasing resources from the Eden Age generation (newly created objects). So in sum the usage of heap memory of Show Face application is normal and no memory leak or unexplained behaviour is to be observed in the graph. Finally something very important to note is that value of –Xmn parameter has to be chosen very carefully and after testing and profiling of the application. A very small value in relation with an application's memory needs can lead to excessive calls to the garbage collector and excessive usage of the garbage collector consumes resources and can degrade the application's performance. In this case that the application was profiled under a stress situation there were 74 garbage collector calls that took 1 second of the total processing time (if it was not a stress period profiling, garbage collector calls should be considerably less).

Finally the other 2 graphs, Non-Heap Memory and Classes are identical and there is a reason for this. As it has been explained previously non heap memory is used to store mainly loaded classes and the Classes graph shows the number of loaded classes in the memory. So they depict approximately the same thing but using other metrics. As it can be seen the vast majority of classes are loaded from the start up of tomcat so already in the beginning 3300 classes are loaded. As the application starts serving requests some additional classes are loaded in the non heap memory that are required and from that point on the line remains stable since whatever was needed has been loaded, the multiple

requests are just asking the same things for multiple users so no new class is needed to be loaded in the non heap memory.

Finally after checking the application's behavior and resources it uses through this profiling tool, everything seems normal concerning CPU and memory usage and execution times seem good and in accordance with the response times of the jmeter tests in the previous sections of this chapter. The application's performance and scalability has been measured and its design and architecture benefits show their real meaning in this area.

10.Application Configuration and Setup Instructions

The application requires some configuration and certain steps for someone to set it up, in this chapter this info will be presented. To begin, java version that is required for the application but also for the web server is 1.5 or bigger. To check this in Windows command line type java –version and the current system's info concerning java will appear. For example:

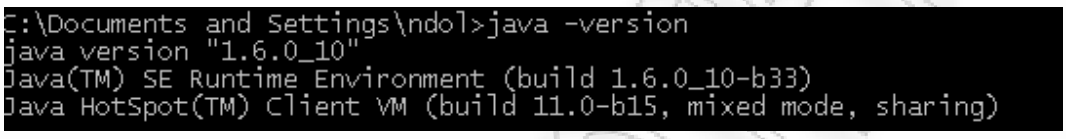


Figure 42. Java Version Check

Now the files that will be needed to setup the application are:

- An Apache Tomcat version 6 binary distribution preferably in zip format, for example apache-tomcat-6.0.18.zip.
- The folder that will be provided which contains the database files of the application already set up with some initial data.
- The WAR (Web ARchive) file of the application. This contains the Show Face application ready to be deployed to the web server (showface.war).

Starting with the web server installation the only thing needed to be done is to unzip the distribution preferably to a path without any spaces, for example C:\apache-tomcat-6.0.18, in the rest of the instructions this path will be considered as the root path of Apache Tomcat. Now since the web server is installed what has to be done is to configure the datasource as it was stated in a previous chapter. Go to C:\apache-tomcat-6.0.18**\conf** folder. There open file "context.xml" and within the <Context>(HERE)</Context> tags add the following:

```
<Resource name="jdbc/MyDB"
auth="Container"
type="javax.sql.DataSource"
username="sa"
password=""
```

```
driverClassName="org.hsqldb.jdbcDriver"
maxActive="20"
maxIdle="10"
url="jdbc:hsqldb:hsql://localhost"
/>
```

This will provide via jndi the reference to the database connection. After this is done, a last step to configure the web server is to add hsqldb.jar file in the lib folder of Tomcat, C:\apache-tomcat-6.0.18**\lib**. This is required as in there is the database driver that was previously defined in "context.xml" file. Now Tomcat is ready to be started and this can be done in C:\apache-tomcat-6.0.18**\bin** folder by executing **startup.bat**. A successful start up screen shot should be like this:

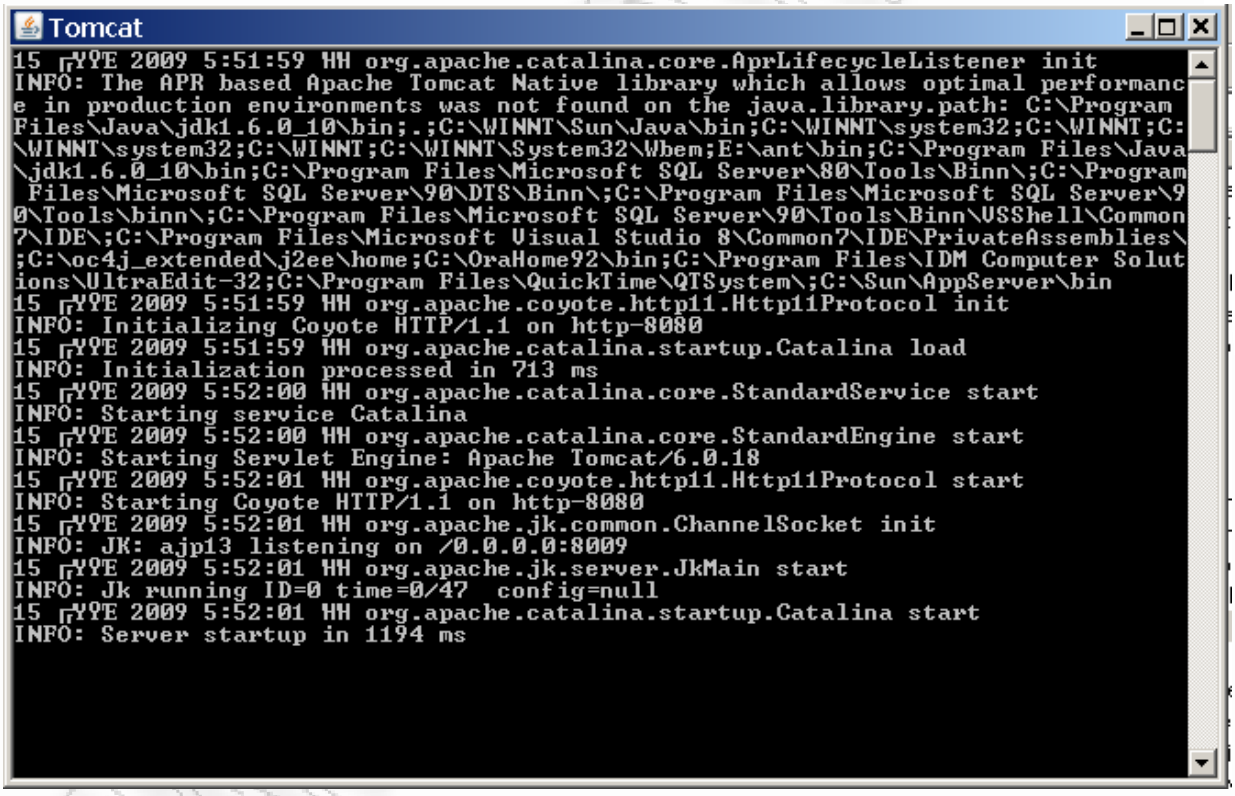


Figure 43. Tomcat Successful Start Up

Now that Tomcat is up and running the database server should also be started. Go to the provided folder in the cd named "showface". This folder contains the source code of Show Face as an Eclipse project ready to be imported and the database folder. Copy the

whole “showface” directory to a local disk and go to folder **“...\showface\hsqldb**”. Inside this folder execute “**starthsqlserver.bat**” and the database server should be up and running. A successful start up screen shot should be like this:

Figure 44. HSQLDB Server Successful Startup

At this point both web and database servers are up and running so now it is time to deploy the war file (showface.war). The deployment is very simple, the only thing required to do is to copy the provided showface.war file in this folder of Tomcat: C:\apache-tomcat-6.0.18**webapps**. When this is done check the Tomcat console that will find the new application and deploy it. If everything works fine something like this should be shown at the end of the console logs:

Figure 45. Show Face Successful Deployment in Tomcat

If everything has been done with no problems the application can now be accessed from a web browser by writing: **http://localhost:8080/showface/** (or any other port that Tomcat might have been configured to run, 8080 is the default one).

11.Conclusion and Future Work

Throughout the whole thesis Show Face demonstration application has been presented in great detail regarding technological and architectural choices trying to give an insight in solving the problem that refers to complexity and scalability of large scale applications. As it has been shown great care has been taken in every step of the application's design concerning various issues. A clean and well defined architecture has been defined following the MVC pattern. This resulted in having several discrete packages or abstraction layers that each of them had its assigned responsibilities. Moreover the idea of separation of concerns and use of centralized control for certain issues like transaction management has been followed wherever possible, this way the maintainability and expandability of the application are improved. Dependency points on configuration issues like database setup have been decreased as much as possible. All the above combined with various new and performing technologies that have been integrated and used show how to build an application that can meet today's needs and can adapt in quickly changing environments and needs.

Now apart from the work that has already been presented there is more that can be done regarding future work, which could improve the application even more and introduce some new concepts. One such idea is the use of a caching mechanism that could improve even more performance by reducing database Input / Output actions. The setup and modification of the application required to operate in a clustered architecture would also be a modification of great interest. Apart from performance related improvements new features could be added in order to integrate more technologies like the use of Ajax model to implement a simple web chat mechanism in the application. Of course these are some suggestions and there are many things that can be done in the above context.

12. References

1. http://java.sun.com/blueprints/patterns/MVC-detailed.html
2. D. Kayal . *Pro Java EE Spring Patterns.* Apress. 2008.
3. C. Walls, R. Breidenbach. *Spring in Action.* Manning. 2005.
4. C. Begin, B. Goodin and L. Meadors. *iBATIS in Action*. Manning. 2007.
5. http://java.sun.com/developer/technicalArticles/J2EE/despat/
6. Rod Johnson et al. *The Spring Framework – Reference Documentation.* 2008.
7. http://displaytag.sourceforge.net/1.2/
8. J. Machacek et al. *Pro Spring 2.5.* Apress. 2008.
9. http://highscalability.com/
10. http://tiles.apache.org/2.0/framework/tutorial/pattern.html
11. http://jakarta.apache.org/jmeter/index.html
12. http://www.yourkit.com/