That One Special Shot

Critical Design Review

TOSS TEAM

Gregory LaFlash
Patrick O'Loughlin
Zachary Snell
Joshua Howell
Hao Sun
Kira Jones

Department of Computer Science and Engineering
Texas A&M University

3/3/2014

Table of Contents

1 Introduction

1.1 Problem Background

At any large gathering – whether it is a wedding, convention, or sporting event – attendees will want to both take photos and see photos taken by others. With the proliferation of smartphones, taking photos has never been easier. Retrieving other' photos, however, remains cumbersome. It is cumbersome because no one wants to engage in the time consuming task of contacting photographers individually (this assumes, of course, we have their contact information). A way to easily create and access a crowd-sourced photo database would be beneficial. The proposed application, upon completion, will be such a database. It will give users a variety of storage and access options as well as include numerous security features.

1.2 Needs Statement

Current software which attempts to create these databases are woefully inadequate: Once our project has been completed, no one will come close to having its range of functionality. Our application will be distinct from current applications in the following three major ways.

First, storage comes with a variety of options. Current applications only support one type of storage, severely limiting the consumer's choice and thus making it less likely that the application will suit their needs.

Second, our program will implement numerous measures to ensure and maintain our users' privacy and security. Our research into other applications has shown that, as of yet, this is not even a passing concern. However, as internet users become more concerned with their privacy online, any form of social media (including crowd-sourced photo databases) must take this concern into account.

Third, and perhaps most important, our application will be free. Any application currently on the market which comes close to offering our range of functionality is either inherently for profit or must charge the users' a fee to recoup the money needed to store all of its users' photo.

1.3 Goals and Objectives

We plan to create an application which will support a crowd-sourced photo database. It will provide various methods of storage, privacy, and security for free. Using our app, anyone with the proper permissions will be able to upload a photo they have taken in real time to one of the numerous databases our application will support.

In order to keep the program free, our method of storage must be free as well. To do this, TOSS will leverage the APIs of a number of social media websites, which will service as our makeshift databases. These social media websites include Facebook, Photobucket, Twitter, Instagram, Pinterest. Users will also be allowed to store their photos using Dropbox, an FTP server, or private server that they must create and host.

1.4 Design Constraints

Our primary constraints derive mostly from the fact that it is our aim to keep our application free. For an application to be free, our storage must be free, and to keep our storage free we will leverage the APIs of social media websites. It is here where we will encounter the most constraints. Our constraints will fit into two categories: documentation problems and media throttle limits.

First, because the APIs for these social media site change so quickly their proper documentation is often not kept. Many tutorials are therefore out of date and useless. Second, many APIs have method throttle limits, which restrict how many times we can call a function of the API. For instance, Photobucket will only allow our application to upload 10,000 photos a day to a specific album.

1.5 Validation and Testing Procedures

The overall system must have several different points tested. Technical points as well as user interaction points must be evaluated.

- Android application usability
- Python web service (frontend) usability
- Python web service response time
    - Under normal conditions
    - Under heavy load on web service (many users browsing existing event albums)
    - Under heavy load on photo service (many users uploading photos)
- Java web service response time
    - Under normal conditions
    - Under heavy load (many users uploading photos)
- Database security
    - Are usernames and passwords stored properly?
    - Are unauthorized users prevented from viewing photo details or event details?
    - Are the event IDs secure and being recycled correctly?
- Are the users' Facebook, Photobucket, and Dropbox account credentials secure?

Obviously, the most important feature to test would be the basic functionality of the system. Does it perform the task it is meant to do? Our team will take pictures with several Android phones simultaneously and upload them using an event ID. We will test uploading them to Facebook, Photobucket, Dropbox, and a private server. If this test passes, we shall test uploading photos concurrently to multiple events.

Usability testing will be performed by polling a large portion of the CSCE 482 Senior Design students. Students will have a chance to vote on a Lickert scale just how usable the interface is, in addition to providing a short description of what they found intuitive and what was difficult. This method for usability testing can be done for both the Android application and the main website.

Load testing will be organized by measuring response times of both web services as well as resource allocation of the server while many users are uploading photos. The large number of users will again come from the students of CSCE 482. Multiple users will upload multiple photos to different events while we record response time of the services.

Security testing will be accomplished through attempts to break into our system. Examination of permissions concerning the database will be crucial.

2 Proposed design

2.1 Updates to the proposal design

Since the initial design there have been a few changes in the design.

- We originally planned to implement the REST calls on the Android app. However, we found that due to limitation to the Java libraries included with Android we could not use Jersey. We decided to instead use a Spring API for the Android REST calls.
- We originally planned to use a REST call from our photo service to the web service to signal that a photo was uploaded for an event so that the webpage could update. However, after considering the overhead of this approach we have decided to go with a time interval refresh.

- We originally planned to have the photo service handle saving photos to the various API services. However, we decided that it would be a better design to abstract the API access into a layer that could be accessed by both the photo service for saving and the web service for retrieving photos.
- We updated the budget by adding an Android phone. The member of the team developing the Android app did not have an Android phone to test on and the emulator was not sufficient for testing.
- We noticed a new product that had been released called Wedpics. From this app we got the idea to do guest card printing, email invites, and albums.

2.2 System description

TOSS will be comprised of six main parts: an Android phone, two services, an API abstraction layer, a storage medium, and a database. The Android phone will be the main method of interaction for users. The first web service will be the other method of interaction for the user (frontend), and will control user account creation, login, and photo displays. The second web service will be strictly backend-oriented, handling incoming photos, authenticating device IDs and event IDs, and communicating with the API abstraction layer. The storage medium will be decided by the user at the creation of each event. Storage mediums may be Photobucket, Dropbox, Facebook, or a user's private server. The database will store user accounts, event IDs, and links to photos.

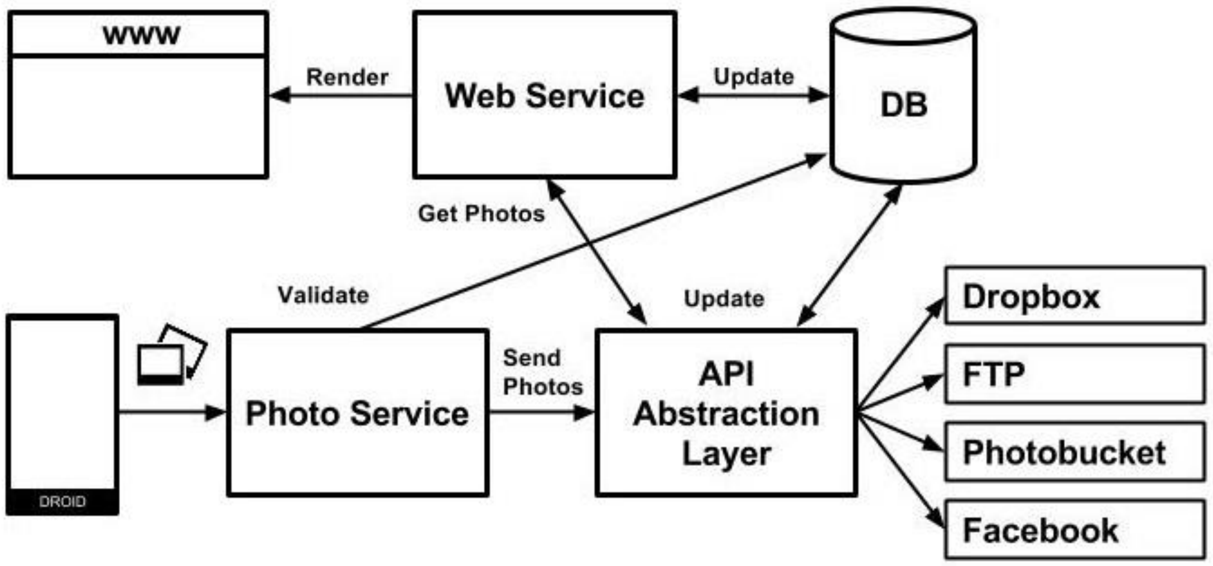


Figure of the Basic System Architecture.

The Android application will be developed by Gregory. The app will consist of the default camera component, a screen to input an event ID code, and a screen to choose which pictures should be sent to the web service using REST. The Android application will send each photo as a REST call, with image data, tags and comments, device ID, event ID, and time taken present.

The frontend web service will be built using Python and the Django Web Framework. It will handle the creation of user accounts and events, the display of photos, and the downloading of all pictures taken at an event. Kira will handle the technical design of the web service front end. Hao will be in charge of 'branding' the website and creating a uniform look and feel throughout the project. The Python web service will interact with the database to manage account and event creation. It will also interact with the API abstraction layer to retrieve photos from their respective storages. Event creation will assign a unique identification code to each event, generated through an md5 algorithm. The host of an event will distribute a QR code to his guests. The QR code will translate into an event code, which maps to the unique event ID.

The photo service (backend) will be implemented in Java and Jersey. Jersey is a framework for creating REST applications. Maven will be used to manage dependencies and builds. The web service will be created using Grizzly, a lightweight Java server. Patrick will be working on the photo acceptance and event ID authentication. Josh will be managing the storage of photos in the user's Dropbox, Facebook, or Photobucket account. The decision for splitting the photo-handling service and the user interface service was made by Gregory, to reduce load on the UI while many pictures were being uploaded. This service will communicate with the Android application, database, and Python service. The Android app will send the image, device ID, and event ID, which will be read by the Java service. We will look up the event ID in the database to determine its authenticity. The device ID may be logged to prevent spammers from abusing our system. After the event ID has been verified, the photo will be sent to the API abstraction layer to be placed in its respective storage.

The inclusion of several different storage options for the user allows for more flexibility and privacy than other services may allow. Josh will be in charge of integrating our web services with Facebook's, Dropbox's, and Photobucket's APIs. The list of planned storages is as follows:

- Facebook
- Photobucket
- Dropbox
- Private Server

The standard services (Facebook, Dropbox…) will be simple to set up. The host for the event will provide his credentials to the specific account to allow our services to upload the guests' photos directly to the account. The private server option is a more technical option for the truly privacy-conscious user. This option will allow the users to bypass all of our services. Instead of giving his guests an event ID, he will distribute a link to his private server, which will collect and display photos sent to it.

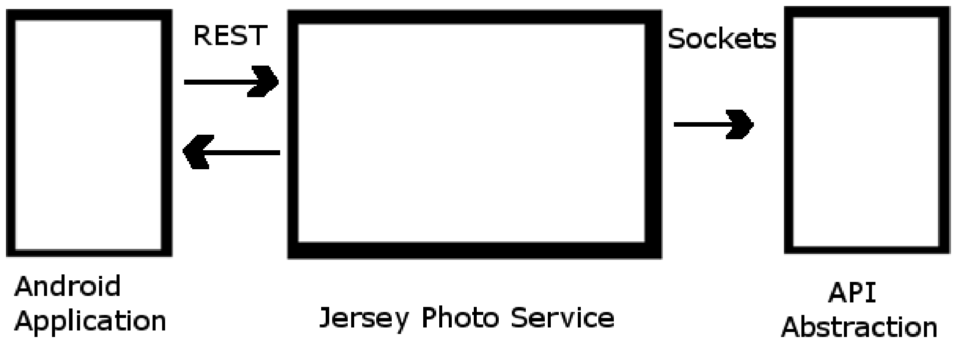
The mySQL database will contain information related to user accounts, event IDs and time constraints, and photo links. Zach will set up and maintain the database. The Python web service, Java web service, and API abstraction layer will communicate with the database. The Android apps will never interact directly with the database.

Zach purchased a server NFOservers.com and installed 12.04 LTS Ubuntu distribution. Gunicorn, Nginx, Supervisor, and Virtual Environment are installed on the Linux machine. Most of our development work will be done through this server.

2.3 Complete module-wise specifications

Android Application

- Requires Android version 4.0 and above because we are using slider switches in our application which is only supported in version 4.0 and above.
- Using Spring API for performing REST calls.
- Using SimpleXML for serializing app data.
- Using ZXing Barcode scanner in order to read QR codes.
- Decided to use a camera intent vs a camera API because the camera intent provides a user interface that is already similar to the default camera app and takes the least amount of effort to implement. The only downside is that we are unable to customize the camera screen.



The relationship between the Android Application, Photo Service, and API Abstraction Layer is shown.

Photo Service

The photo service has been being developed using Java, Jersey (REST), and Grizzly (server). Maven is being used as a dependency handler and build environment. The decision to use Java (as opposed to Python, for example), was based on Java's speed and Patrick's familiarity with the language. Java, being a compiled language, is considerably faster than Python, which is interpreted. Jersey was chosen as a RESTful service because it seemed to be the most widely-used, and as such, had the greatest amount of documentation available to us. Grizzly was used as the server due to its simple and lightweight nature. Since it is developed by Oracle, and is contained within the GlassFish project, documentation was also abundant. An additional bonus for each of these technologies is that they are all contained within the Central Maven Repository, enabling rapid deployment with minimal hassle. Using a Jersey/Grizzly Maven archetype, we were able to get a simple REST service up fairly quickly. Maven also resolves most of Java's class path headaches when compiling and running. In short, each component's compatibility with Maven created a very pleasant development environment.

The photo service, as of this writing, has two main actions. The first is event code validation, in which a user's Android app will send a REST GET call of the form http://toss.myphotos.cc:18081/validationService/<code>. Plans for the next iteration of design include implementing a caching system to avoid interacting with the database too frequently. The second function is to receive photo and event information from the Android application via Multiparts and pass the relevant data to Josh, who will be managing the API abstraction layer required to transmit the photos to Dropbox and Facebook.

The communication between the Android application and the Jersey photo service takes place through a REST framework. The REST call for the photo uploading is a POST method using the Multipart media type to the URL http://toss.myphotos.cc:18081/photoUpload/.The communication between the photo service and the API abstraction program is implemented over sockets.

The interaction between the Android and the Jersey photo service uses JSON for the code validation function. The photo uploading function uses Multipart data, consisting of an InputStream to contain the photo data, a FormDataContentDisposition object containing the file details, and a String with the EventID. JSON was chosen due to its simple nature and widespread usage. Multipart is used largely because nothing else we tried was successful. Multipart was the most commonly-used method of transmitting files, but it was not trivial to implement. We had been attempting to convert the image to a byte array and send it via JSON, but the connection timed out. Fortunately, Greg discovered how to use Multiparts.

Maven dependencies included in the Jersey photo service:

Artifact ID	Group ID
jersey-container-grizzly2-http	(org.glassfish.jersey.containers)
jersey-media-multipart	(org.glassfish.jersey.media)
commons-io	(org.apache.commons)
json	(org.json)
mysql-connector-java	(mysql)
org.apache.commons.codec	(org.apache.directory.studio)
junit	(junit)

API Abstraction Layer

In its final version, Toss will allow users to store photos on a variety of services, including, but not limited to, Dropbox, Facebook, and an FTP server. Java was chosen for implementing this functionality for two reasons. First, and perhaps most importantly, it is well understood by Josh, who is responsible for the API abstraction. Second, Java remains in prolific use. Most, if not all of the APIs with which we will be working support the language.

Originally the API abstraction was to be written using the Jersey/Java photo service. However, as we did more research into what specific API's required to function properly, we found the Jersey/Java service to be wholly unnecessary.

We decided on implementing Dropbox first, primarily for practical reasons. Because the APIs we intend on using change so quickly, the documentation we would need to properly integrate these programs into TOSS fail to keep up. As a result, figuring which puzzle pieces goes where can be a Sisyphean effort. Dropbox, however, was the exception which proved the rule. We wanted to get a basic version of Toss working as quickly as possible, and Dropbox provided well written and easily comprehensible documentation for our purposes.

The photo which will be uploaded to Dropbox will be passed to me through a socket. I have created what is called a TossObject, a class shared by both Patrick and I which is suitable for this purpose. A TossObject contains a variety of pertinent information. It contains the device Id of the device which is attempting to upload the file; the event ID, which will determine, among other things, where the photo is to be stored (dropbox, facebook, ftp server, etc.); a photo comment to be attached to the photo; a photo's tags; and, lastly, an input stream which can be passed to the Dropbox API upon upload.

Patrick will pass me a TossObject through a server socket I create. I will then extract the information from the TossObject, interpret it, and send the photo where it needs to go.

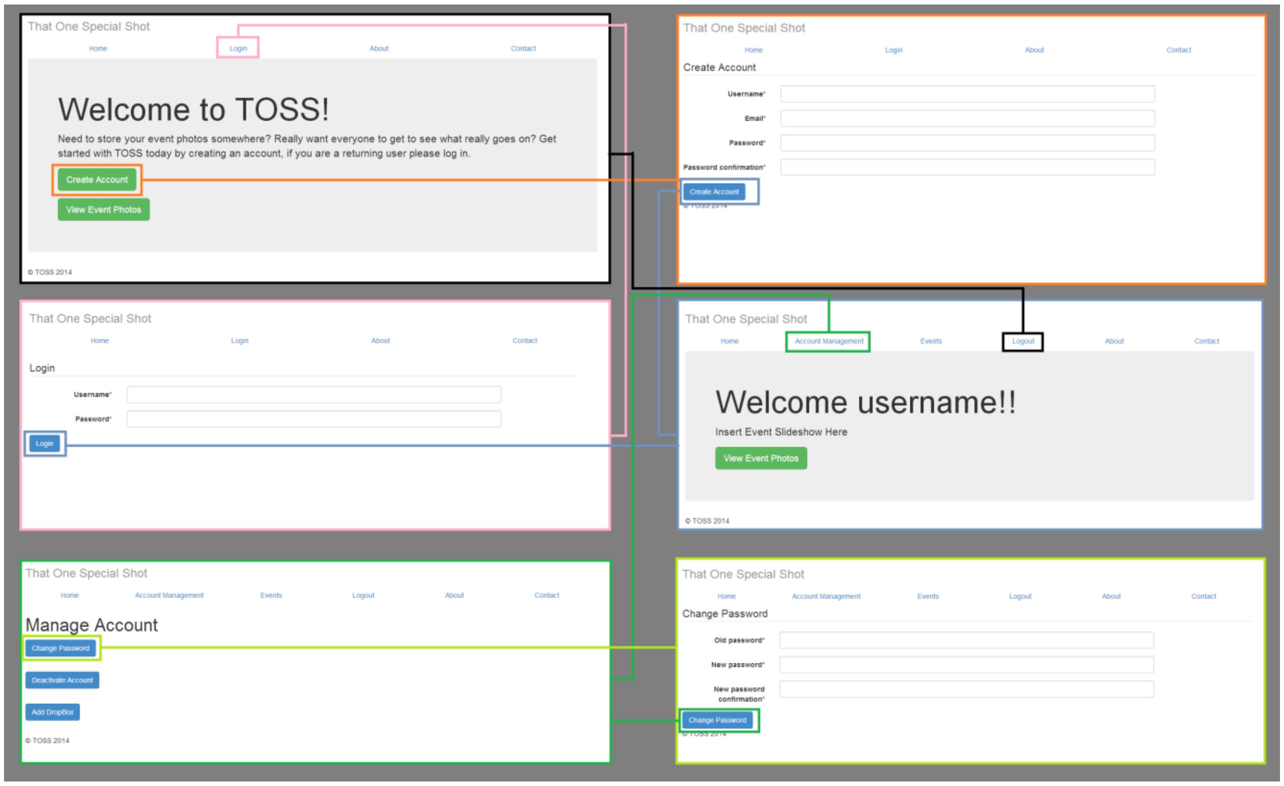
Web Service

The web service is implemented with Django, Nginx, and Gunicorn.

- Django is a formidable web display engine which has been proven to be effective in very large implementations which scale from single person websites to the largest of newsrooms which have constant updating documents and display them in real time to their massive userbases.
    - This engine was chosen for its ability to scale and its simplicity of implementation. Additionally, having a moderate expert in Django makes it something that was an obvious choice for our team. There is no reason for us to choose less secure and more complicated environments when we have something that works well and is well known

and documented.

- Nginx is a premier content management system. It handles the presentation of information, images, content, etc to the user. Nginx very carefully handles the control of url access and flow redirecting the user as needed and making the web service as secure as possible from actual file access.
    - Easy implementation with Django and built in security makes it the obvious choice.
- Gunicorn (Green Unicorn) was chosen as a scalable web engine which has good interactions with Django and large precedence.
    - Apache tends to not work as well with scalable Django applications which made us lean more towards Gunicorn for doing the implementation we desired.

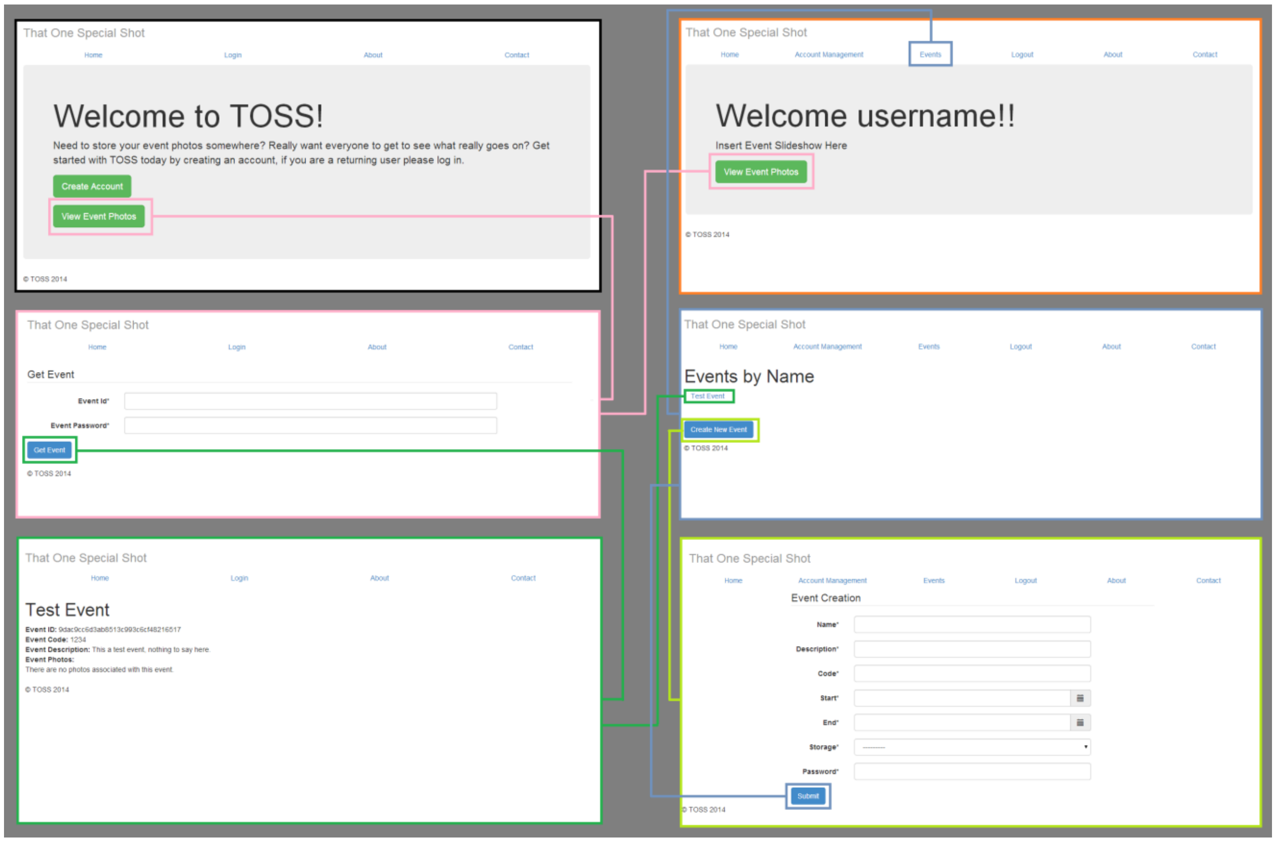


Need to store your event photos somewhere? Really want everyone to get to see what really goes on? Get started with TOSS today by creating an account, if you are a returning user please log in.

Web Service: Account System Flow

User Accounts

The web service will require users who want to create events to create a user account with our service. They will be directed to a form prompting them for a username, which must be unique in our database, an email, which must follow valid email format, a password, and a password confirmation, to avoid typos. A new user entry in the Users table in the database will be created. The user's unique username will serve as a key. The username will be tied to all events they have created as well as any APIs they have added to their account. An existing user will be able to login and logout of our web service. They will also be able to do basic account tasks like add APIs, change passwords, and deactivate their account. A logged in user will have access to all of the events they've created and all their corresponding photos without being prompted for the event id and event password unlike users who are not logged in.



Welcome to TOSS!

Need to store your event photos somewhere? Really want everyone to get to see what really goes on? Get started with TOSS today by creating an account, if you are a returning user please log in.

© TOSS 2014

Welcome username!!

Insert Event Slideshow Here

Get Event

Event Id*

Event Password*

Events by Name

Test Event

Event ID: 9dac9cc6d3ab8513c993c6cf48216517
Event Code: 1234
Event Description: This a test event, nothing to say here.
Event Photos:
There are no photos associated with this event.

Event Creation

Name*

Description*

Code*

Start*

End*

Storage*

Password*

Web Service: Event System Flow

Events and Event Codes

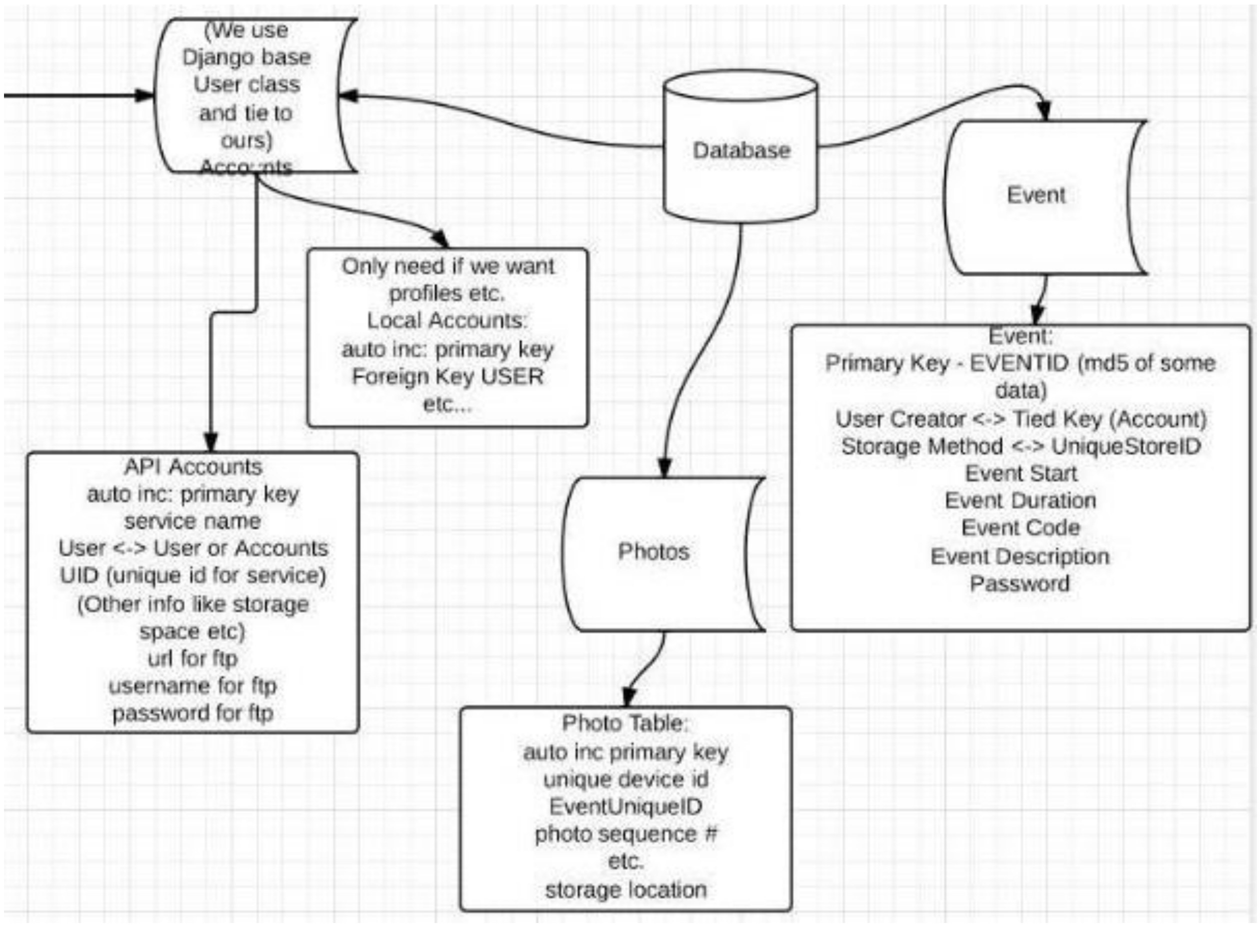
The web interface allows for users, who are registered within our site, to add storage locations for their photos and then create events which allow other users to upload photos during a time frame to those locations. In the photo above, the last image is the event creation form. In this form you are automatically given a list of all added API points to select from, the ability to choose your own event code, and set a password for the event. Event codes are first come, first serve, and expiring codes. This process was chosen as a simple interface for hosts of all demographics to be able to create in a simple manner which follows an intuitive flow.

Photo Retrieval

The web service will provide photo retrieval capabilities to both users who have accounts as well as those who do not. Users who have accounts and are logged in to the service will be able to access all photos of their own events. Those who do not have user accounts or are not the host of that event, guests, will be able to look up an event and corresponding photos using the event id and event password provided to them by the event host. With valid ids and passwords they will have access to that event's information like name, description, and photos. The web service will interact with the API abstraction layer to retrieve the photos from their corresponding storages to allow users to both view and download photos.

Website Front-end

One of the most important issues for our project is the website design and framework. We will use bootstrap as a tool to create our websites because it contains HTML and CSS-based design templates for typography, forms, buttons, and interface component as well as java extensions. The advantage of using bootstrap is that it is compatible with the latest version of all major browsers. In our project, we downloaded the bootstrap package from the official website and put it in our project file for instant customization. Bootstrap is modular and consists essentially of a series of LESS stylesheets that implement the various components of the toolkit. A stylesheet called bootstrap.less includes the components stylesheets. Developers can adapt the Bootstrap file itself, selecting the components they wish to use in their project. There are a wide variety of features such as fonts, backgrounds, tables, headings and so on. Bootstrap can provide an easy way to design the look of our website. In addition to the regular HTML elements, Bootstrap contains other commonly used interface elements. These include buttons with advanced features like grouping of buttons or buttons with drop-down options, make and navigation lists, horizontal and vertical tabs, navigation, breadcrumb navigation, pagination, etc., labels, advanced typographic capabilities, thumbnails, warning messages and progress bars. Since we are using Django to create the web application for our photo service, we will mainly divide the website into three parts which include user account settings, home page, and extended web service (gallery, help, contact and so on). We use the bootstrap to set up the basic theme and framework for our website interface in all these three parts. Also, it will be easy to maintain or change the appearance if we want to.


(We use Django base User class and tie to ours) Accounts
Database
Event
Only need if we want profiles etc. Local Accounts: auto inc: primary key Foreign Key USER etc...
Event: Primary Key - EVENTID (md5 of some data) User Creator <-> Tied Key (Account) Storage Method <-> UniqueStoreID Event Start Event Duration Event Code Event Description Password
API Accounts auto inc: primary key service name User <-> User or Accounts UID (unique id for service) (Other info like storage space etc) url for ftp username for ftp password for ftp
Photos
Photo Table: auto inc primary key unique device id EventUniqueID photo sequence # etc. storage location

Database Schema

Database

Mysql was chosen as our Database Backend for its well documented and well implemented history. We chose to proceed with this DB as a result of its inclusion in all versions of Python and Java as well as its security and scalability. Various other DBs were examined such as CouchDB but our lack of knowledge in them led us to choose a sql which is well known.

Our schema is a simple 4-5 table implementation which, while possible to expand, is quite complete in its current form. The image above shows what we currently have with the event table, the API table, and the photo table. Accounts table is still being considered but is currently seen as not necessary. There are various other tables included by Django, like the user table, which are not necessary to explain here. They are default and auto implemented.

3 Project management

Gregory

Responsibilities

- Project lead
- Develop the Android app

Qualifications

- Over a year of experience of working at a software consulting company
- Android app Experience
- Experience building multiple web applications

Patrick

Responsibilities

- Handle the interfacing with the Android app from the web service side
- Handle the interfacing with the website front end web service
- Live viewing of photos

Qualifications

- Interest in graphics, which will be needed in creating a custom live preview of the photos

Joshua

Responsibilities

- Handle saving photo saving to all storage options
- Handle Retrieving photo using APIs
- Social media integration
- Creating the schema for what information from the photo will be saved and how it will be saved

Qualifications

- Interest in working with the various APIs needed to save the photos to various storage options such as Dropbox, Facebook, Photobucket and others

Hao

Responsibilities

- Branding the website
- Creating the website excluding the parts specifically tied to a backend process.

Qualifications

- Experience with website design

Zachary

Responsibilities

- Database design
- Schema for event code generation and maintenance
- Purchasing the web server hosting

Qualifications

- Full time employee at Texas A&M - CIS - Software Developer
- Created and maintained web applications based on Django
- Experience in Database maintenance
- Already had a hosting account and has experience with web server maintenance

Kira

Responsibilities

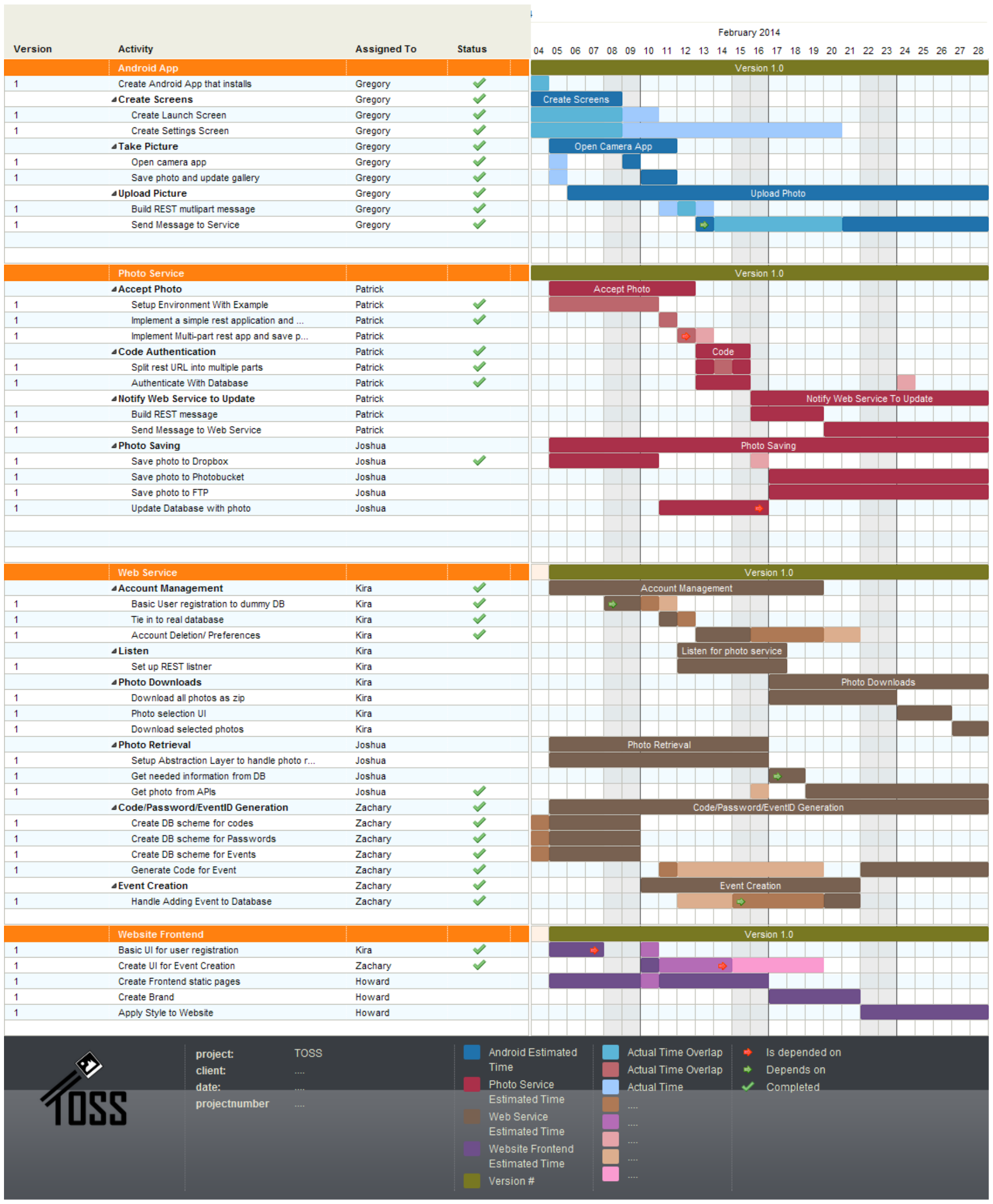
- Handle the interfacing with the photo upload web service.
- Creating the schema and implementing user accounts
- Handle both the frontend and backend needed to retrieve and download stored photos

Qualifications

- Experience developing a Django web application

3.1 Updated implementation schedule

Every Monday we do a progress check which is used to update the Gantt chart in the next section. Our approach to task management is working very well and we have been ahead of schedule on most tasks. We have a few tasks that are starting to take longer than expected, but our task management strategy allowed us to adapt and break the task down further. This allowed us to put more people on the task and speed up the development.



Version	Activity	Assigned To	Status
	Android App		
1	Create Android App that installs	Gregory	✓
	◢**Create Screens**	Gregory	✓
1	Create Launch Screen	Gregory	✓
1	Create Settings Screen	Gregory	✓
	◢**Take Picture**	Gregory	✓
1	Open camera app	Gregory	✓
1	Save photo and update gallery	Gregory	✓
	◢**Upload Picture**	Gregory	✓
1	Build REST mutlipart message	Gregory	✓
1	Send Message to Service	Gregory	✓
	Photo Service		
	◢**Accept Photo**	Patrick	
1	Setup Environment With Example	Patrick	✓
1	Implement a simple rest application and ...	Patrick	✓
1	Implement Multi-part rest app and save p...	Patrick	
	◢**Code Authentication**	Patrick	✓
1	Split rest URL into multiple parts	Patrick	✓
1	Authenticate With Database	Patrick	✓
	◢**Notify Web Service to Update**	Patrick	
1	Build REST message	Patrick	
1	Send Message to Web Service	Patrick	
	◢**Photo Saving**	Joshua	
1	Save photo to Dropbox	Joshua	✓
1	Save photo to Photobucket	Joshua	
1	Save photo to FTP	Joshua	
1	Update Database with photo	Joshua	
	Web Service		
	◢**Account Management**	Kira	✓
1	Basic User registration to dummy DB	Kira	✓
1	Tie in to real database	Kira	✓
1	Account Deletion/ Preferences	Kira	✓
	◢**Listen**	Kira	
1	Set up REST listner	Kira	
	◢**Photo Downloads**	Kira	
1	Download all photos as zip	Kira	
1	Photo selection UI	Kira	
1	Download selected photos	Kira	
	◢**Photo Retrieval**	Joshua	
1	Setup Abstraction Layer to handle photo r...	Joshua	
1	Get needed information from DB	Joshua	
1	Get photo from APIs	Joshua	✓
	◢**Code/Password/EventID Generation**	Zachary	✓
1	Create DB scheme for codes	Zachary	✓
1	Create DB scheme for Passwords	Zachary	✓
1	Create DB scheme for Events	Zachary	✓
1	Generate Code for Event	Zachary	✓
	◢**Event Creation**	Zachary	✓
1	Handle Adding Event to Database	Zachary	✓
	Website Frontend		
1	Basic UI for user registration	Kira	✓
1	Create UI for Event Creation	Zachary	✓
1	Create Frontend static pages	Howard	
1	Create Brand	Howard	
1	Apply Style to Website	Howard	

project:	TOSS
client:	....
date:	....
projectnumber	....

🟦	Android Estimated Time
🟥	Photo Service Estimated Time
🟫	Web Service Estimated Time
🟪	Website Frontend Estimated Time
🟩	Version #
🟦	Actual Time Overlap
🟥	Actual Time Overlap
🟦	Actual Time
🟫	....
🟪	....
🟥	....
🟧	....
🟪	....
➡	Is depended on
➡	Depends on
✓	Completed

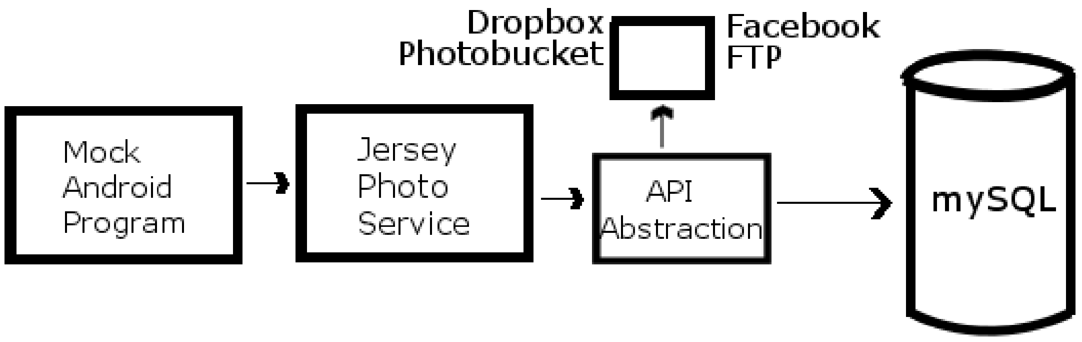
3.2 Updated validation and testing procedures

The validation and testing procedures will focus more on component integration for this stage of project. Stress testing the connections between modules is critical. Notable areas of the project are:

- Android application and Jersey photo service (REST)
- Jersey photo service and Java API Abstraction (sockets)
- Jersey photo service and mySQL database
- Java API Abstraction and mySQL database
- Java API Abstraction and the various APIs (Dropbox, Facebook, FTP server)
- Web service frontend (Django) and mySQL database

Each of these areas is important to the overall functioning of the system. The Android application needs to be able to communicate with the Jersey service quickly even under high loads. The Jersey service needs to be able to send the photo information quickly and without error to the API layer. The API layer needs to be able to upload the photos to the various storage mediums without delay. A large buildup of photos stored in memory while waiting for Dropbox to handle our request could create difficulties for all programs running on the server. The API layer also needs to be able to update the information (photo location) stored in the database. This information will be being retrieved by the Django web service at regular time intervals to update the frontend gallery and slideshow. The Django web service should be less critical than the more internal components of the system, since it can be assumed that the majority of users will not be excitedly hitting the F5 key to refresh the event's photos. However, almost all users will be using the internal parts of the system (Android app, Jersey service, API, and database).

There are two main ways to test the integrity of the overall system. Either we could have a chain of tests running, where the data from the beginning (mock Android program) is the same data that is being sent from the Jersey service to the API, from the API to the storage mediums (Dropbox), and all database interactions along the way. Since many of these component share resources on the same system, it may be beneficial to test them concurrently.



Stress Test Flow

This illustration shows a comprehensive stress-test of the system. Dummy images are sent from the mock Android program to the Jersey service, which are then relayed to the API, which is sent to Dropbox. The resulting location is stored in the database. Not shown are the connections between the Jersey service and the mySQL database or the Django frontend.

Alternatively (or perhaps additionally), each connection between components could be individually stress-tested. The connection between the image-spewing mock-up and the Jersey service could be tested and evaluated before moving on to the extreme socket testing between the Jersey photo service and the API program, and so on.

Stress testing for the REST framework between the Android and Jersey service could be simulated using a mock program to rapidly toss large numbers of images to the Jersey service through REST and Multipart. This test program could play the part of the Android phone, mimicking the actions of a thousand simultaneous TOSS users.

The Android application will need to be stress-tested by itself. Is it capable of submitting a hundred queued photos to the Jersey service? Will the app crash or slow down? Will the photos reach their destination? Will we receive a broken pipe error or timeout?

The sockets between the Jersey service and the API abstraction could be tested individually by having the Jersey service spam large numbers of InputStreams very rapidly through the socket. Ensuring that the socket does not break under the load is important. Perhaps even more important would be to implement a ‘self-healing’ socket implementation. If the connection between Jersey and the API Abstraction is lost, the programs need to be able to detect that and automatically set the socket back up. Human intervention each time the service breaks would take far too long.

The limits of the various APIs need to be explored a little more carefully. Meticulous research would probably be more appropriate than spamming Dropbox with a thousand photos a minute before knowing their policies. In order to determine whether or not an online storage medium will meet our requirements, we should see if limits exist on data sent, calls made, or frequency of calls. Once we are sure of the official policies of these services, we can begin stress-testing the relationship between the API Abstraction and the various storage mediums.

Testing the database interactions could be done by having the API program quickly populating a dummy table of photo information. The efficiency of this interaction could be measured by accuracy (was everything stored properly), memory usage (both peak and average) and cpu usage (peak and average).

It would probably be best to test each area individually, making sure we are within limits for each connection between components. Once we are confident in the paired performance of our different modules, we can test the system as a whole, using the same mock data for each stage.

3.3 Updated division of labor and responsibilities

Gregory LaFlash is the project lead and will oversee the progress of the entire team both on their individual assignments as well as the entire project overall. Gregory will also be in charge of the overall Android application including design and implementation.

Gregory’s Deadlines

Basic Android Application Completion - March 10
Extended Android Application Completion - April 14
Final Android Application Test - April 25
System Overview - April 27
Project Documentation Consolidation - April 29

Patrick O’Loughlin is responsible for designing and developing the jersey photo service which will handle the interfacing between the Android application and the API abstraction layer. The photo service will receive event and photo information from the android application and perform validation checks before forwarding the data to the API abstraction layer.

Patrick’s Deadlines

Basic Photo Service Completion - March 10
Extended Photo Service Completion - April 14
Final Photo Service Test - April 25

Joshua Howell is in charge of designing and implementing the API abstraction layer. The API abstraction layer receives photo and event information from both the photo service and web service. It is in charge of photo storage and retrieval to the various storage APIs using the information passed to it from both services, as well as information found in the database. Joshua will also be our presenter during presentations.

Joshua's Deadlines

Basic API Abstraction Layer Completion - March 10
Extended API Abstraction Layer Completion - April 14
Final API Abstraction Layer Test - April 25

Zachary Snell is handling the overall database including the schema design. Zachary is also in charge of the web service event system which includes event creation, code generation, and photo retrieval using the API abstraction layer.

Zachary's Deadlines

Basic Web Service Event System Completion - March 10
Extended Web Service Event System Completion - April 14
Final Web Service Event System Test - April 25

Kira Jones is responsible for the web service user system which includes user creation, login, logout, and account management. She will also help Zach with photo retrieval using the API abstraction layer. Kira is also in charge of consolidating document material as well as putting together presentations.

Kira's Deadlines

Basic Web Service User System Completion - March 10
Extended Web Service User System Completion - April 14
Final Web Service User System Test - April 25

Hao Sun is responsible for branding the website. Hao is in charge of the overall look and feel of the website including the HTML, CSS, and bootstrap excluding the parts that specifically tie to a backend processes like forms.

Hao's Deadlines

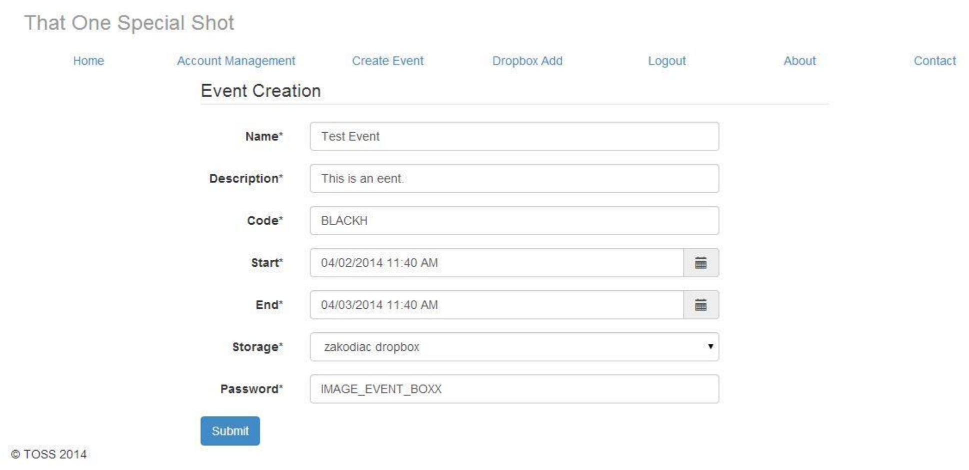
Basic Website Branding Completion - March 10
Extended Website Branding Completion - April 14
Website Appearance Overview - April 25

4 Preliminary results

Development of the project continues on schedule as expected and with no real delays. There are always various problems which are more difficult but all of these have been resolved in portions of time excess to various other areas of the project. The results we have available so far include many attributes necessary to allowing the actual use of our project.

We have been able to:

- Enable our full server and run it efficiently without too many issues. Various memory problems have been seen but seem to be stabilizing.
- The basic website is up and running allowing for user registration, host registration of API access point to Dropbox, and creation of events.
    - The figure below shows event creation and API storage selection.


That One Special Shot

Event Creation

Name*	Test Event
Description*	This is an eent.
Code*	BLACKH
Start*	04/02/2014 11:40 AM
End*	04/03/2014 11:40 AM
Storage*	zakodiac dropbox
Password*	IMAGE_EVENT_BOXX

© TOSS 2014

- The acceptance of photos from the android application has been completed as well as the entry of events via code or QRCode.
- Our REST entry point is online and taking photos as well as doing validation. The uploading process is being implemented now to put the photos on the storage server.
- Various features are being designed and implemented currently including email submission and additional storage locations.
- All progress and testing has demonstrated ability to scale as well as decent idle speeds