CS221 Group Project
Final Report

	Aldridge, William	Atkins, Max
	Barnes, Xander	Dart, Nicholas
Project Team	Harizanov, Maksim	O'Hare, James
	Pocock, Michael	Tuff, Sebastian
	Wilcock, Daniel	Wilmot, Andrew
Version	0.1	
Status	Pre-Release	
Date Published	February 13, 2015	
Reference	SE_O2_FP_01	
Department	Computer Science	
Address	Aberystwyth University	
	Penglais Campas	
	Ceredigion	
	SY23 3DB	

Contents

1 Management Summary

foobar

2   Project History

foobar

3 End of Project Report

foobar

4 Team Performance

foobar

5 Test Report

6 Maintenance Manual

6.1 Android

6.1.1 Brief Description of Final Product

The Android app first required user details including name, phone and email. It then also requires a (pre-made on the web site) site name which is downloaded from the API on app start up. The data is then validated and on successful validation will proceed on to a specimen entry screen.

The specimen entry screen takes a Latin plant name and a DAFOR scale represented by a slider. There is also an option for a scene and a specimen photo, which can be retaken as needed. Multiple Specimens can be added and reviewed in the specimen upload screen.

In the specimen upload screen, specimens can be edited or deleted by long pressing on the list. They can then be uploaded to the server.

6.1.2 Program Structure

The Android app is structured into several key areas:

Activities
Activity classes are bundled together and contain event listeners, "Async Tasks" and location listeners. They also contain all the validation for input fields as well as adding lists to "AutoCompleteTextBoxs".

Utilities
The "Utils" package contains database utilities and classes for adding, searching, finding and deleting data from the databases. The following databases were implemented in the app:

1. Plants - Contains the Latin names

2. Specimens - Contains specimens the user has take for upload

3. Users - The information the user has entered for auto-completion

4. Sites - The sites that can be used, pulled from the API

There also exists a "OSGridReferHelper" class which is a third party class originally to be used to convert from GPS coordinates to a os grid reference for the specimen adder. It was decided that this feature was not required and so was not used.

Data Classes
These describe users, visits specimens and species and are used when reading/searching from and inserting into the database. The specimen class was not used as heavily as expected.

Layout
The layout of the Android activities are described in xml files, whose names follow the "activity_¡activityName¿" naming convention and are located in "src/main/res/layouts".

6.1.3   Significant Algorithms

The following details significant algorithms that may require alterations in the future.

When the app starts, there is an initial "Async Task" thread spun off to download the sites list and update the database and to download the specimen list. This thread is located in the "LauncherActivity" class. This thread will check, upon download of the sites list, will add non-existent sites to the database. If the site list cannot be found, no viable error will be shown to the user, however the user may continue to use sites that already exist. The thread then proceeds to check if the latin names database is present and populated. If it contains entries, it will not download the Latin names list (as it is very large and slow to process, and so should be avoided). If it is not present, it will be downloaded and processed in a batch insertion to the database.

There is a location listener in the "SpecimenAdder" activity which gets the users location via gps when adding specimens. It does not however use course (network based) location to get this. This listener implements the standard Android interface "LocationListener".

The databases are manipulated by Database Classes in the utilities package. These classes use the "DatabaseUtilities" class which inherit the standard "SQLiteOpenHelper" class. Tables are accessed via a class by their name eg "SiteDataSource" to access the sites table. These classes require a reference to the data class.

Upload of the final list of reserves is done in another "Async Task" in the "ReviewActivity" where a connection to the API is established and the data from the databases serialised to JSON and uploaded. At present the JSON is constructed manually. The list of specimens is then cleared (however the UI is not updated).

6.1.4   The Main Data Areas

The app makes extensive use of an "sqlite" database to store reserves, specimens, users and species. These are manipulated by classes for each table under the "Utils" package, which contain methods for selecting, inserting and deleting data from the databases, via the use of data classes related to the table's content.

The "DataClasses" package contains classes primarily used for storing data to be entered into or read from the database. These classes contain getters and setters for the data. They also contain an "id" property which is only set when the data is read from the database, this is to allow for deletion of data via it's unique id in the database.

Finally, we use "Adapters" in the android API to allow "AdapterViews" in the activity to access data. adapters

6.1.5   Files Accessed

The App accesses images located on the device which were taken in the specimen selection screen. These images are located on the root of the internal storage device.

The App also downloads the Latin names file from the remote location "http://nic-dart.co.uk/ nic/res/plantlist.json", these are then processed into the sqlite database. This is done only in the init thread.

6.1.6   Interfaces

The app required a gps device to acquire location during the specimen addition process, as well as a camera if an image is desired.

6.1.7   Suggestions for Improvements

- Specimen and Scene images could be silhouetted and represent a scene and an specimen. IE not the same image.

- Deleting of multiple specimens

- Upload completion dialogue/confirmation

- Users and sites are specific to specimen, not read at upload time

- Auto complete lists not overlapped by keyboard

- Settings/Preferences menu or activity

- Custom GPS coordinates for specimen

6.1.8   Future Improvement Concerns

6.1.9   Limitations of the Program

- Deleting images before upload will most likely cause crash

- The "current user" details at the time of upload will be used.

6.1.10   Building from source

The source contains an "Android Studio" project file, this contains all the links to the files needed. It will require JDK (or OpenJDK 8) and the Android API and may need to be told where they reside. Building requires either a physical Android device of version 4.0.3 (API 15) or above, or alternatively a virtual Android device, created in the AVD.

6.2   Server

6.2.1   Brief Description of Final Product

The server is a combination of a Database and an application program interface. It acts as the bridge between the Android device and the website to the Database, further protecting the database from direct access. This allows code, sensitive information and data to be stored in a non-accessible place. The server contains a relational database which holds all the data important for the system: specimens, reserves, records and users. To access this data, the website and the android application must use the API .php script commands which run validation, sanitisation and SQL queries using JSON data structures.

6.2.2 Program Structure

The server API is a list of commands that are to be run by the android application and the website to access and manipulate data in the Database. The Design Specification document contains all the information about the program structure.

For a list of the commands in use in the system and what they do, please see the document SE-O2-DS-00: Design Specification, section 1.5: Significant Server Components. For a detailed list of how the commands work including their required POST requests, their return status codes and required headers, please see the document SE-O2-DS-00: Design Specification, section 4.1.3: Server Interface. For an example sequence diagram of the most complex command (addRecord.php), please see document SE-02-DS-00: Design Specification, section 4.1.13. All other commands follow a simpified sequence to this. For the Component Diagram of the server, plaese see document SE-02-DS-00: Design Specification, section 4.2.1: Detailed Design: Diagrams.

6.2.3 Significant Algorithms

Validation and Sanitisation:

To validate data inputs we ensure strings are not empty and we ensure latitude and longitudes are valid as true values. Ensuring strings are not empty:

```
1 if (empty($locationName) || empty($locationOS))
  {
      die();
      return status code 400;
  }
6
```

Ensuring latitude and longitude values are true:

```
  if($Latitude > 90 || $Latitude < −90 || $Longitude > 180 || $Longitude < −180)
  {
      die();
      return status code 400;
5 }
```

To sanitise all the data inputs, the command real_escape_string() was used on all POST data received from the website:

```
$SpeciesName = $conn−>real_escape_string($specimen−>SpeciesName);
```

This prevents any malicious data from touching the database.
SQL Queries were used to retrieve/manipulate data in the database:
Add:

```
"INSERT INTO botany_users (user_name, user_phone, user_email) VALUES ('
    $UserName', '$UserPhone', '$UserEmail')"
```

Update:

```
"UPDATE botany_reserves SET location_name='$LocationName', location_os='
    $LocationOS', description='$Description' WHERE reserve_id = $ReserveID"
```

Get:

```
"SELECT * FROM botany_reserves WHERE reserve_id=$reserveID"
```

Remove:

```
"DELETE FROM botany_reserves WHERE reserve_id = $reserveID"
```

We manually manipulate the committing for the SQL queries to ensure no invalid data gets added. All checking is done, and then we tell it to commit. This allows to reverse any affected data if it proves to be wrong.

We provided the ability to authenticate a password input to ensure the user was authorised to use commands:

```
if (!strcmp($password, $CONFIG['adminPassword']))
{
    echo "true";
}
else
{
    echo "false";
}
```

We wrote SQL queries to create the database tables, e.g:

```
create table botany_records (
record_id INT AUTO_INCREMENT ,
user_id INT NOT NULL,
time_stamp INT,
reserve_id INT,
PRIMARY KEY (record_id));

alter table botany_records
add foreign key (user_id)
references botany_users(user_id)
on delete cascade on update cascade;
```

6.2.4   The Main Data Areas

To transfer data from the android application and the website to the database, the data had to be structured in a JSON file format. JSON was a nice, flexible and easy to use language, allowing us to quickly and easily create data structures to be decoded by the API. To do this, the commands json_encode() and json_decode() are used. For example, if a new specimen was being added by the website, the website will json_encode(specimen array), and the server will json_decode the POST data. If a specimen is being sent to the website, the server will json_encode(specimen array) and the website will json_decode(POST data). To see each specific JSON data structure, please see document SE-02-DS-00: Design Specification, section 4.2.2: Significant Data Structures. This section also details the config file for use when changing resource locations and passwords.

6.2.5   Files Accessed

The API accesses the resources directory defined in '/includes/config.php' heavily. One should refrain from manually accessing the contents of that directory, and altering the directory structure. The PHP scripts should be allowed read/write access to said directory. The API does not access files outside of the resources directory.

6.2.6   Interfaces

The server's PHP scripts that implement the web API functionality use the JSON standard to receive structured data from and transmit structured data to its clients. JSON is used in conjunction with the widely accepted HTTP protocol, the JSON being carried in either the POST headers (in client requests), or the HTTP response body (in server responses).

All web API commands need to be called via the HTTP protocol using the POST method, and need to be supplied the required parameters (either plain text, or JSON format, described in detail in the design specification).

A common settings interface is used for storing environmental variables such as database credentials, administrator password, and server root. Those are all stored in /includes/config.php inside the API directory.

6.2.7   Suggestions for Improvements

A number of improvements could be made, although the basic functionality of the API is complete up to the specification standard.

The choice of platform and programming language for the API was limited to the available technologies on Central. This does not represent a real-life situation, where the developers would likely have administrative control over the server the API is hosted on, allowing them a more flexible choice of technology and platform. Discarding PHP in favour of a more modern language:

- Would reduce future maintenance issues.

- Would have sped up the development of the product, and will speed up the development of any future improvements.

- Make quality assurance easier and more efficient.

- Could potentially reduce the hardware requirements of the API, depending on the selected replacement technology stack (Java EE/ASP.NET).

Therefore, a transition from PHP to Ruby on Rails, Python+Django, or even Node.js would bring long-term benefits to the project, with Java EE and ASP.NET bringing immediate savings from hardware and electricity costs.

Another, more specific improvement, is the inclusion of 'pretty URLs'. An example of that would be the current '/getResouce.php' web method, which could be rewritten as '/resource/id' followed by the desired resource ID. This requires relaxation of the requirement to use POST requests exclusively, as in many cases GET requests make more sense (a RESTful approach would suit the use-case perfectly, in fact). Another example: '/reserve/remove/id' replacing '/removeReserve.php'.

6.2.8 Future Improvement Concerns

Separate web methods in the web API are reasonably separated. A change in one method should not induce undesired changes in others. However, all methods share a single point of communication, and therefore – failure: the database. The database structure is rigidly defined, and as of now, the code base does not allow one to easily change table and column names. This means that any changes to the database schema, even trivial ones such as renames, need to be carefully accounted for in the API source code, and thoroughly tested before being released into production.

While separate web methods are independent of each other, the same can not be said of the separate routines that constitute the methods. The code was produced in a rush, and little to no time was left for refactoring. Many parts of the source code are therefore in the form of long, function-less scripts, with no modularity whatsoever. Therefore, extra caution should be practised when editing: a change in one variable, or a slight rearrangement of the statement order could generate unexpected behaviour in seemingly unrelated sections of code in the same file.

6.2.9 Limitations of the Program

As an interpreted language, PHP is fairly slow compared to compiled and JIT-compiled languages. For low-to-moderate size use cases, it's performance is perfectly reasonable, but it lacks the high scalability offered by some competing technologies.

The technical limitations are, therefore, modest: a mainstream 2006-era CPU such as a Core 2 Duo, and 1GB of RAM, would have little issue running the API provided it is not heavily used. As the incoming requests rise, though, hardware will begin to fail to meet the demand, requiring higher-performance hardware. Eventually, one would not be able to obtain a faster computer, and load balancing across several machines becomes necessary, which the API architecture does not allow at the moment (as it uses local file storage to save resources). At this point, resource storage would need to be moved to a separate server, and the API source code would require change.

Regarding the database: MySQL is a good database engine for all but the highest-performance scenarios. Switching the database engine to InnoDB will bring further performance benefits, but that would remove the support for referential integrity, currently built into the DB schema, and support for transactions, used by the API. Workarounds can be implemented, however. Considering the project lacks the ability to provide a revenue stream of

any sort, purchasing expensive licences for more capable databases such as Oracle Database or Microsoft SQL Server is inconceivable, and according to projections, unlikely to ever become necessary.

6.2.10   Building from source

The web API requires no rebuilding. Copying the API files inside the Apache 'public_html' directory, with PHP installed of course, should suffice.

The database can be set up by executing the table creation scripts (found at 'table_queries.txt) on a blank MySQL database.

Once the database is set up, one should edit the '/includes/config.php' file and configure it with the database address, name, username and password, the path to the server root, and the path to the resources storage. An admin password should also be picked at this point. Ensure all PHP scripts have write access to the resources directory.

7   Document History

Version	Edit	Date	Persons
0.1	Initial Version	Febuary 12 2015	nid21