Final Report

The Project Overall

We believe the project has been a success overall. We have nearly finished writing the code, and we plan to put the finishing touches on it before the code is handed in next Monday. Major components such as the website, data gathering, and SMS and Voice integration are all complete and working together. We have made a product that would be useful and informative for subsistence farmers, and it is accessible in many different languages. Relevant information such as weather forecasts and crop prices are provided to users, and we have made a system for government workers or NGOs to monitor the health of users' crops, and broadcast advice where required.

Successes and failures

- **Success**: good integration with SMS and Voice systems for both sending and receiving messages
  - We managed to successfully work with a third party API (Twilio) to manage to send SMS messages to all numbers and receive them as well. We have also managed to create an IVR that picks up and deals with all calls made to the system and allows all users to move to whatever is the required thing to do, whether this be hearing a new update or reporting on crop health.
- **Success**: flexible multi-lingual implementation based on Google Translate
  - By entirely making use of Google's translation systems, in text and text-to-voice, we have made a highly extensible system that would work with a large variety of languages and therefore this would allow the system to be easily adapted to work in any country. The system was made from the ground up with this assumption, and therefore the additional coding that would be required to get the system to work in a new country would be minimal. Additionally, the usage of Google's systems means that we would be utilising the fact that their translation and voice systems are likely to improve the fastest due to the amount of data they are getting. This is relevant in especially slightly less used languages, including the ones we would be likely to deal with.
- **Success**: navigable configuration process, with persistent storage of user information
  - We chose to make the configuration process entirely done over text to allow information such as location to be easily sent. In this configuration process, the user can choose everything from which language is to be used to the crop they farm and their location. All this information is stored in a persistent database that is polled to get this information to provide localised and personalised updates.
- **Success**: we can provide accurate weather data, and somewhat up to date crop data
  - Accurate weather data is acquired using Dark Sky and Food Security Portal is the source of crop data. A new (up-to date) message is generated every day and sent to every registered user, however, they can also send a message to the server to get a new update or call in and receive a completely up-to date update including both weather and crop data.
- **Success**: secure website for sending and receiving messages
  - The website to send and receive messages, largely intended from crop health works only for those who are given an account, which then allows them to access the sending system. In this, they can provide a message, a relevant location and radius for whom the message should be delivered. In return, the users can call in and provide a crop

    - health report which is delivered through the map interface and allows the administrators to hear what the farmers had to say.
- **Success**: website for hearing and viewing crop health reports works, with a map based interface
    - This allows one to easily find any trends of location of reports and to hear specific reports from specific locations. This might also help to advise where to target advice for messages to be spent as well as finding the correct radius.

- **Failure**: finding crop data sources of decent quality
    - From the beginning, the biggest issue we had was finding accurate, up-to date and accessible data. In general data sources were at most two of these with the best sources being highly expensive. Therefore, we eventually settled for the best data source that we could find, the Food Security Portal which offered reasonable accuracy and high accessibility, with the prices being updated every month. While therefore not very up-to date it clearly still added a large amount of value to farmers to offer some idea of what the prices should be.
- **Failure**: secure communication between our servers
    - By introducing two seperate systems that need to communicate between one another, we added a number of complexities of security in ensuring this channel is secure. However, largely due to time constraints, we have been unable to spend too much time considering this communication and / or making it entirely secure.
- **Failure**: we have not had time to implement our 2-day long final testing plan, but we will have time to do this and perform modifications before the code submission deadline
    - In our first report, a major way that we would evaluate the system would be through a long, two day test, where all aspects of the system would be tested and made sure to be working. Due to the time constraints, we have yet to entirely finish and cannot start on this test until that has occurred. Thus, we hope to begin the test on Friday and end on Sunday.
- **Failure**: Registration system not complete by the time this report was written.
    - Due to time constraints the configuration system has yet to be finished before writing this report. We aim to have this working by the client meeting tomorrow, and will be having a long group meeting before the client meeting to fix this, amongst other things.

Lessons learnt

- **Unnecessary Complexity**
    - We implemented the project in two parts, the part which communicates over voice and SMS and the part which serves the crop health data and receives messages to be broadcast to the users. We decided to keep these separate to start with to make it easier to produce. However one subgroup decided in implement their system in PHP, and the other used Java Spring.
    - However, this added a whole lot of unnecessary complexity which was communicating between our two systems, and by the time we realised this we couldn’t move over to one system without throwing significant amounts of code away.
    - Instead we decided to continue with this split-system approach, and have simple communications between the two systems (using HTTP requests with two ports of the

server open). This worked, but it added new security problems which we have been unable to deal with due to time constraints.
    - We realized that in real-world circumstances, it is important to have a framework the whole team agrees to adopt. It would be far easier to deliver all functionality under one framework (i.e. one of Spring and PHP). We realized this is also why it is necessary to have prior knowledge of popular development tools (version control, frameworks, testing harnesses, etc.) before joining a project or becoming an employer for a corporation.
- **Time Management**
    - **(1) Dealing with unforeseen issues**
        - Through the course of the project, three of the five team members fell ill and were unable to do work for a period of time. This affected the plans of integration since specific components were not ready while others were. In future, it would be better to be more proactive, and make sure that people are able to hand off work to others if necessary, if they are ill. Additionally, it would be good for individuals to have a priority list for their work, to ensure they do the most important elements first and the elements which can be left to later, can be dealt with later. This is especially important when they could be struck down with illness at any time.
    - **(2) Better splitting work earlier in project**
        - In a particular example, after two weeks one of the large tasks, the configuration messages and communications with the user was moved from one person to another. In some ways, this is an assurance of the success of communication, as one person realised they had too much work and was able to shift this over. However, in other ways, it is a failure of planning at the start of the project, where we should have been able to foresee this issue and assigned the work better to begin with.

Individual Work and Technical Contributions

Ashwin - Translation, SMS and Voice Communication

I worked on using a number of different API's to take messages from the form that they could be sent to the user, and to take messages from the user and parse them. To this end, I utilised a number of Web APIs to do this:

- Twilio (https://www.twilio.com/) to send and receive messages and to receive calls.
- Google Cloud Translate API (https://cloud.google.com/translate/docs/) as the method for translating messages. While this is a paid API in general, one can receive $500 of free credit to test systems, so we have chosen to use this.
- Sound of Text (https://soundoftext.com/docs) to get MP3s of the output from a Google Translate text-to-voice system. If you want to do this directly via Google, it is rather expensive and so using a third party who in general go to the online (free) version of Google Translate and get the audio from there was a better idea. We were originally going to use Twilio's text-to-voice system, however, this would have ended up being mildly expensive as well as not very good quality.
- What3Words (https://what3words.com) to get a longitude and latitude from three words for any location on earth. We thought this would be a good way of getting location, taking the assumption that someone would be going around getting people to sign up, so they could

simply give them their three words and then without needing to know the name of the village (and have to deal with whether Google knew the existence of the village), we had a reasonably accurate location.

Working with APIs
For each of the APIs, they either worked with HTTP POST or HTTP GET requests. Therefore, I used the Apache HTTP library in order to send the requests and get the response back from the system. In order to send data in JSON format, the GSON (Google JSON) library was used to create the JSON from a custom object. In order to parse the responses, the Jackson ObjectMapper was used to create an object of the correct kind from the response from the server, in order to more easily parse the response. In particular, communications with each API were as follows:

- What3Words: used a simple GET with custom parameters and returned a large response with a lot of unnecessary information as well as the location. Therefore, everything else was ignored, other than the lat long and then this was returned.
- SoundOfText: used two HTTP requests to eventually get everything. Firstly, the required text was POSTed to the service and the response contained an ID. This ID could be used in a GET request to then get a URL at which the MP3 was contained.
- Google Cloud Translate: Utilise the Google Cloud Translate Java library to create the required objects as well as to send requests and parse the response. I created an interface by which any of the other parts could translate a message simply given the input and output language and the text.
- Twilio API: Use the Java library of Twilio to send messages and to create structures in XML for the IVR system. As part of the former system, I created an interface for the other parts of the code to be able to send any message, either translated or untranslated, given a source language, output language, text and phone number. For more information on the IVR, see below. For receiving messages, I simply pointed the POST requests on a phone call or message received to /callIn and /messageIn on the server through the Twilio console.

IVR: The IVR handles all the incoming calls, and I created it to work as follows:

1. If the user is not registered (phone number is not recognised) - enter registration process and tell user to respond to SMS messages
2. If user is registered
    a. Offer the follow menu:
        i. Hear new update
            1. Get new update and play to user
        ii. Offer crop health report
            1. Tell user to start talking, record this and pass this onto the submitCropHealthReport.php
        iii. Re-enter configuration
            1. Re-enter registration process and tell user to respond to SMS messages
        iv. Hear menu again

IVR Technical Challenges:

1. Text-to-voice Packet Issues
    a. Turned out that the format and header values that the text-to-voice API provided for the MP3 wasn't being recognised by Twilio.

    b. Therefore, I created a system to automatically download the files on our server and then serve this file to the Twilio system when it went to look for the file. This was all created using Input and Output Streams in Java
    c. Each file has a unique (random) file ID to allow it to be accessed later.
    d. In the future, should create a garbage system that deletes these files every so often.
2. Text-to-voice character limit
    a. Turns out there was a character limit on text-to-voice inputs from SoundOfText. The only time this was tested was when we provided update messages to the user - therefore I just split these into composite parts and do each part separately and play consecutively.
3. Latency
    a. When the system had to split these into parts, this made everything rather slow, especially leading to the user having to wait several seconds for an update (which was okay, but preferred to be avoided). Therefore, I implemented a form of caching, where the URLs of an update (as an MP3) was saved in the information about the user, as well as the date at which the URLs were created. Therefore, if an update is requested, we first check if there has already been MP3s created today, and if so, it simply uses these without creating a new set of them, which would take a long time unnecessarily.

Benji - Data Gathering and Message Generation

I worked on collecting data from individual data sources and parsing them into messages that could be sent to users. The Web APIs I used were:

- Dark Sky (darksky.net/dev) for weather information. Dark Sky are a popular weather forecasting company.
- Food Security Portal (foodsecurityportal.org/api/) for crop prices, which gives monthly updates for local Rice and Maize prices for developing countries. It is facilitated by the International Food Policy Research Institute.
- Currency Converter API (currencyconverterapi.com) for localisation of currencies. I'm not sure where the data comes from exactly, but it seemed to be very accurate and it offers a vast range of currencies for conversion for free.
- OpenStreetMap's Nominatim (nominatim.openstreetmap.org) for converting geographical coordinates into country names, because Food Security Portal and the Currency Converter API both index their data on country names, but we are storing our users' locations as exact coordinates (which is a level of precision that can be used by the weather forecasting API).

I did research into trying to find a better crop data source, and couldn't find any other free websites with more types of crops and also an API to read the data automatically. Food Security Portal get their data from the UN Food and Agricultural Office's (FAO) Global Information and Early Warning System Food Price Monitoring and Analysis Tool. This didn't have an API that I could find, and it would challenging to try to navigate this website automatically and scrape data out of it. This tool gets data from various sources, but for the region we have been considering all of the data comes from the East African Grain Council Regional Agricultural Trade Intelligence Network (RATIN). To access this data starts at $400 per year, up front, which are costs well beyond the means of a group project. Therefore I decided to make do with the small amount of freely accessible data from Food Security Portal. If this was being developed into a product to actually be used, a licence could be bought and it wouldn't be very difficult to write a few more data gathering classes to use this data source.

Dark Sky, Currency Converter API and OSM Nominatim all use JSON based REST APIs, so it was a matter of crafting HTTP GET requests based on the user data, and parsing the responses. I used Google's GSON library to deserialize the responses and wrote classes to represent the expected request results for GSON to deserialize into. Food Security Portal returned results as CSV files, so I used Apache Commons CSV to parse the responses.

I wrote the code to turn the data into simple English, to give the translator the best chance of producing something sensible. This mainly involved putting the data in simple sentences, and choosing the most relevant data for farmers to include as the weather update (in the end I chose the rainfall amount, maximum and minimum temperatures of the day, and their times, as well as Dark Sky's textual summary of the weather). This was a balance between giving useful information for the farmers, without making the messages overwhelmingly long.

I added a table to the User database to store the custom broadcast messages submitted on the website, and the code to accept these messages as HTTP POST requests from the PHP server. I wrote the code to decide which users to send these broadcast messages to, based on the users' location. I also configured the server application to send update messages to each user on a daily basis.

Jason - User Database and Registration Protocol

My main contribution was working through the documentation for JPA (Java Persistence API) as well as Spring AutoConfiguration in order to build the persistence layer for our web application in the most hassle-free way possible.

Persistence was eventually achieved by using Spring's AutoConfiguration to wire up Hibernate (an object-relational mapping tool), an H2 Embedded database (rather than having to work with a separate MySQL instance running elsewhere), and to pick up JPA annotations across our code space so Hibernate knows which objects we need to store in the Embedded database.

The main object we need to store is the User object, which stores information associated with a particular phone number our platform delivers information to. I created the User class, added the necessary JPA mapping annotations, and wrote a demo Controller that used @Autowired to acquire an instance of the "UserService" -- the Service-layer object that behaves like a Collection of Users by talking to a "UserRepository".

My contribution to the group extended beyond the User class I wrote -- the User, UserRepository and UserService classes served as examples that Benji could directly copy from in order to add a table to the database to store custom broadcast messages coming from Kyra's PHP server. This effectively removed the need to work with SQL schemas on the Java server. However the PHP server running the website used a MySQL instance to store Crop Health Reports and information about users of the website, and this could have been avoided if we all used one system. This is described in the "Unnecessary Complexity" section of this report.

During this process, I realized that most members of the group were writing code that is completely unaware of the Spring framework (for none of us have any substantial experience developing using Spring -- or web applications in general). Benji and Ashwin's API & text-message-generation classes were all (or nearly all) static utility classes that made zero use of Spring dependency injection. In fact, Ashwin's life would be made much easier if he used Spring's RestTemplate for sending HTTP requests from our application to the various API providers. I wrote extensive documentation to make the rest of the group aware of this issue. We eventually established that it is more useful to have everyone work in the way they are most comfortable with, and deal with style / use of framework at the integration phase rather than implementation phase.

At the second phase of the project, I was also responsible for the Controller that handles incoming text messages to our platform. This involves the registration protocol and storing state for each User to track where they are in the registration process. We agreed to make use of the database we already have and store such state in the database, as opposed to in a Map in the Controller (which would mean we have to handle concurrency ourselves).

A minor contribution of mine was the occasional clean-up of our git repository, removing accidentally-tracked binary files, IDE preferences, editing the gitignore, etc.

JP - Crop Report Database and Web Interface

I worked on implementing the interface for the website which displays crop health reports which have been submitted by users. To do this, I built a MySQL database which stores a url link to the mp3 recording of the report, in addition to the telephone number of the user that submitted the report (in case they must be contacted regarding the report), a geographical coordinate associated with the user, and a timestamp. Building the system which allows new reports to be submitted was simple, and involved basic PHP, however extra details were needed to ensure security.

To display the reports, I made use of the Google Maps JavaScript API. This allows a Google Maps view to be embedded in the page, and markers added in the locations in which reports were submitted. To implement it, the map itself mostly used JavaScript, however a PHP script was needed to get the reports from the database, and XML was used to interface between the map and the PHP script. The API has a fee to be used, however upon signing up free credit is granted by Google for a demo period, which more than covers what we need for our group project.

Kyra - Web Interface Authentication and Message Broadcast Interface

- Built the basic web interface layout using HTML / CSS
- Using PHP / MySQL for the login system
- Tested it against common web attacks
- Built the message interface, and send the information on using a HTTP POST request
- Failures: building a secure way of authenticating users, rather than just giving out premade logins, time management