# Personal Write-up

When implementing the project I took on 4 distinct features within the system:

* Internationalization
* Scheduling
* GoAber Web API
* GoAber Challenges Web Interoperability

Each of these features were implemented on the JavaEE and .NET versions of the software.

## Feature Contribution

### Internationalization

During the initial sprint I worked on the Internationalization feature. This turned out to be easier to implement than expected (a rare occurrence in programming), especially in JavaEE. JavaEE automatically placed most strings around the site into an automatically created bundle. By copying these strings into a second bundle called the same but with the addition of ‘\_cy’ and then adding the prefix of [WELSH] to each string, Internationalization in JavaEE was achieved.

.NET was more difficult. Firstly, strings were not automatically placed in a separate file and referenced, this had to be achieved manually. Although this was not difficult, it was time consuming. Secondly, extra code had to be added to use the internationalization files. This was added to a ‘base’ controller. The ‘base’ controller inherited the Controller class and the custom controllers that we created as a team inherited from the ‘base’ controller. The ‘base’ controller contained a method called during the execution of each controller which found the browser language and changed the internationalization file based on this language.

Although there was more work in getting internationalization setup in .NET it still wasn’t particularly difficult, especially as there was a clear tutorial online that could be followed [1]. During the following weeks of development team members updated the internationalization files for both projects as new work was added.

### Scheduling

In the second sprint (week 3) I implemented scheduling within the two projects. The systems required the ability to read user data from Fitbit and Jawbone on a recurring schedule and to also be able to send emails to users at scheduled times.

I first implemented the functionality in .NET. .NET does not contain scheduling functionality in its standard libraries. I therefore had to look at 3rd part scheduling libraries to achieve this. I found two popular libraries that would do the job: Quartz and Hangfire. As the documentation was particularly clear for Hangfire I chose this API.

Hangfire caused a couple of issues throughout the project. Firstly, Hangfire required database tables. This was so Hangfire could continue with tasks if the system was ever restarted. I had a choice whether to add these tables to our existing database or create a new database specifically for those tables. I chose the latter as it promoted maintainability and scalability within the system by decoupling our project tables from the Hangfire tables. If we ever wanted to change our DBMS we would not have to worry about the Hangfire system, and also if we ever wanted to change our Scheduling system we would not have to worry about accidently breaking our core data structures.

I wanted this database to be created automatically if it was missing. I used the entity framework for this and added a ‘dummy’ table to the Hangfire database. If the table was missing I added a record to the ‘dummy’ table. This caused the entity framework to create the database as the framework does not create databases until an attempt is made to use them.

There was a bug in Visual Studio, if a developer deleted the database file Visual Studio would keep a record of the database in its ‘Server Object Explorer’, however the reference within the ‘Server Explorer’ would be deleted. When the application was then started, Visual Studio would declare that it had a reference to the database, the database would therefore not be created. When a record was then added to the database, at this point the application would fail as although it had a reference to the database, the database itself could not be found. To solve this the reference in the ‘Server Object Explorer’ had to be manually deleted with the file.

The real issue with this was that I did not initially tell the rest of the team about this bug as I had assumed that they would not delete the database file after it had been created, at least not immediately. This proved to be naïve of me as almost the entire team deleted the database file on the same day that it appeared and then started having a meltdown that the system had broken. I informed them how to fix this and sternly told them to not select the Hangfire database and press the delete button, no matter how much they yearned to destroy this new project file. It did reinforce within me the knowledge that if there is a way to beak a system, no matter how small a way, almost every person who uses it will accidently find that way and break that system.

For JavaEE I had a choice between two standard library schedulers: Scheduler Beans and the ManagedSchedulerExecutorService. At the time we still planned to implement Emailing functionality. It therefore made more sense to allow admin users to be able to set their own schedule times via a web interface. This could not be done with Beans as the schedule time was set in the annotations within the source code. I therefore chose the more complicated option of the ManagedSchedulerExecutorService. One additional upside was that it would be more similar to the .NET implementation. Unfortunately we did not implement Emailing, therefore the Bean would have sufficed, although it is nice to allow admins to set their own schedules. However, time versus functionality trade off, the extra time to setup the ManagedSchedulerExecutorService was probably not worth it.

In both systems I created ‘Job’ classes. These classes contained a method which was executed by the schedulers as specified times, once or recurring. FitBit and Jawbone each had their own Job classes and Emailing would have as well if it had been implemented.

When it came to implementing the FitBit Job class in JavaEE I realized that the FitBit developers had added all the code to the WAR project. I had told the team from early on that I felt that logic should be contained within the EJB project whilst everything to do with the web interface should be within the WAR project. This would allow the WAR project to reference the EJB, but then extra components within the EJB to reference this logic as well. The EJB cannot reference the WAR so if logic is within the WAR the EJB cannot call it.

I created the scheduler in a separate project which was then referenced via the EJB. The Job classes were within the EJB and passed to this scheduler project. The FitBit Job class would call functions within the FitBit service class. This service class was within the WAR and therefore couldn’t be called by the EJB Job. It took hours to refractor this code into the EJB to allow it to be called by the scheduler. If it had been implemented within the EJB from the start time would have been saved.

### GoAber Web API

In Sprint 4 I undertook the creation of our Web API which would allow clients to send data to the GoAber systems. I implemented this using a SOAP WSDL web service. I created the endpoint methods and created an ‘ActivityData’ data transfer object (DTO) class. This class was created as the ‘ActivityData’ model could not be used due to the extra persistent functionality. Specifically the ICollection foreign key types, these caused a problem as interface types cannot be used with WSDL. I therefore created the DTO types for all types sent over the network, these were then converted to their model class equivalents.

I then decided that whilst being able to send data on behalf of a user was great, it did create a massive security loophole: any client could add data for any user. What I really wanted was for a user to only be able to add data for themselves and for an Admin to be able to add data for any user. I created a simplified OAuth-like scheme where a user could generate a key from the GoAber site and then add this key to the credentials of the SOAP message. The GoAber web service would then validate this key and determine whether or not it belonged to an admin.

For JavaEE and .NET I then created clients which implemented the WSDL web services and allowed a key to be entered. I then tested that I could send data to GoAber from these clients.

### GoAber Challenges Web Interoperability

The final feature I implemented was getting the different GoAber systems talking to each other. This proved to be by far the most challenging functionality that I implemented and in hindsight was probably a multiple person job.

I started off by creating a new web service in the .NET version of the system and a method to receive a Challenge DTO object. I then copied the .NET project to another directory and opened it as a second instance of Visual Studio, both with copies of the project open. I started one project and then added the web service as a client reference to the other instance. I now had the web service and a client referencing that web service within the same project. The client code could look up the endpoint address of another community from the database, this allowed the GoAber site to send the challenge to the correct address. Two instances of the project could now send challenges and receive them between each other.

Next I added another method to the web service which would allow communities to add each other. From the web interface an admin (from community A) could enter the details for another community (B) which would then be added to the database. A request was then automatically sent to B with the credentials of A. B would then generate a key and store the credentials of A and the key within the database and then send the key back to A. A could then store the key from B. This key was included in each subsequent request sent between the two communities which would allow each community to authenticate and authorize the sender.

The last web method was for receiving a results request. When a challenge was created between two communities the creator of the challenge would automatically schedule a job for that challenge. When the scheduled job fired after its specified amount of time the system would automatically generate results for that challenge by adding up all the data between the challenge dates and then store them and send them to the other community. That community would validate the key sent in the request and find which community was referenced by that key. It would then store the results sent by the community and generate its own results, store them and send them back. Both communities now had both sets of results. The challenge was therefore complete. At this point all the functionality was implemented for the .NET system.

For JavaEE I first imported the WSDL from .NET and created a web service from it. By importing the already existing WSDL it ensured that both web service endpoints were exactly the same which gave a higher likelihood of the interoperability between the two systems just working.

I converted the code implemented in the .NET system to the JavaEE system. This took time, but likely less time than if I had implemented it all from scratch again.

I then tested JavaEE with .NET and JavaEE with JavaEE. Getting two instances of Glassfish running on the same computer proved difficult due to port conflicts. I did manage it in the end but it turned out to be easier testing it on separate computers on the same network. The systems could now add each other, send challenges and receive the results between each other in all combinations of system implementation.

### Other Contributions

Throughout the project I always tried to make sure I was on hand to help other members if they were experiencing difficulties. One example was the FitBit implementation in .NET. Craig experienced difficulties getting the tokens to refresh using the OAuth2 library, so I wrote for him a method to manually create HTTP requests by building up an HTTP request using the HttpWebRequest .NET standard library. This was later adapted and used by Helen when implementing the Jawbone functionality due to further difficulties using the OAuth2 library.

## Group Evaluation

I felt that we worked well as a group. Our choice of development methodology (scrum) was successful. We had weekly sprint meetings where we decided what work we would each undertake and then retrospectives at the end of each sprint to discuss work that remained and to also test new parts of the system. We also had a couple of meetings throughout each week to work on features as a group and to help each other.

We often left the testing of the system until very late in the sprint. This was largely due to pieces of work overrunning or being held up by dependencies on other members. This meant that the retrospective was sometimes pushed aside while we tested pieces and merged them into our master branch. In hindsight we should have been more pro-active in getting our pieces of work merged in as soon as was possible to leave more time for the retrospective.

A lot of features within the system had dependencies on each other. Different members of the group would therefore end up altering the same files within the system which caused merge conflicts. Correcting these conflicts (using merge tools) wasted a lot of time every week. We could have potentially done a better job at dividing out the features. That being said we were somewhat restricted, an example being: It would have been easier in terms of code merging if a single person had worked on Fitbit and Jawbone as they used a lot of the same code. However, this would have likely been too much work for a single person in the timespan that we had.

I also believe that better code design would have led to fewer conflicts and a general higher level of maintainability, adaptability and scalability throughout the project implementation which would have saved time. For instance with the .NET project, almost all aspects of the system were contained within the single MVC project. I strongly believe that a similar approach to what JavaEE attempts to force developers into using should have been applied, i.e. using one project for the web interface and another (or multiple) for the business logic. This would have caused far fewer conflicts between members developing the interface and other members developing the core system logic. It would have also allowed for a more easily scalable system for any future development: Features could be more easily outsourced to other computers; message queues or a similar system could be later implemented for a more traffic heavy enterprise system.

Unfortunately, members of the team expressed an unwillingness to break the system up in this way. Likely because it meant that they would have to maintain references between the two projects which may have led to extra work in the short term. How the JavaEE WAR project was implemented (with a lot of business logic within it) gives evidence to this claim: members wanted to build the entire system within a single project like a small desktop application. The system was far more complex than this which sometimes, due to the application being constrained within a single project, caused spaghetti code to ensue.

# Bibliography

[1] - http://afana.me/post/aspnet-mvc-internationalization.aspx