CE301 Final Report

'System Status Monitor' and 'Heimdall'

Written By Sean Fussell

2/6/2016

Table of Contents

The Project....1
- Company Background....1
- Project Goal....1
- Project Overview....2
- Hierarchy and Accountability....3
  - Key People....3
  - Project Team Members....3

Progress Made On Project....4
- Features....4
- Future Development....5
- Project Proposal vs Finished Project....5
- Project Timeline....7
- Things I Learned....7
  - Code Preface....7
  - Build Manager....7
  - Reflection....8
  - Transfer Objects, Data Delivery & Serialisation Witchcraft....8
  - Event Listeners....8
  - Timezones....9
  - Asynchronous Programming....9

Quality Assurance....10
- Project Methodology....10
- Coding Standards....11
- Unit Testing....11
- Problems Encountered....12

Risks....13

Previous Courses....13
- Programming....13

IS.....13
Improvements or Recommendations.....14
Essay Conclusion.....15
Closing Remarks and Conclusion.....15
Appendix A: Sprint Plans & Reviews.....16
Appendix B: Reflection.....19
Appendix C: Mock Data Testing.....21
Appendix D: Timezones Are A Pain.....23
Appendix E: Bite Sized Apps.....26
Appendix F: Serialisation.....27

The Project

McKesson has a system in place called McKesson Capacity Planner, this is an application designed to monitor hospitals or health care facilities resources and improve patient throughput. The project is building a companion or monitoring system for that application called System Status Monitor (SSM).

Company Background

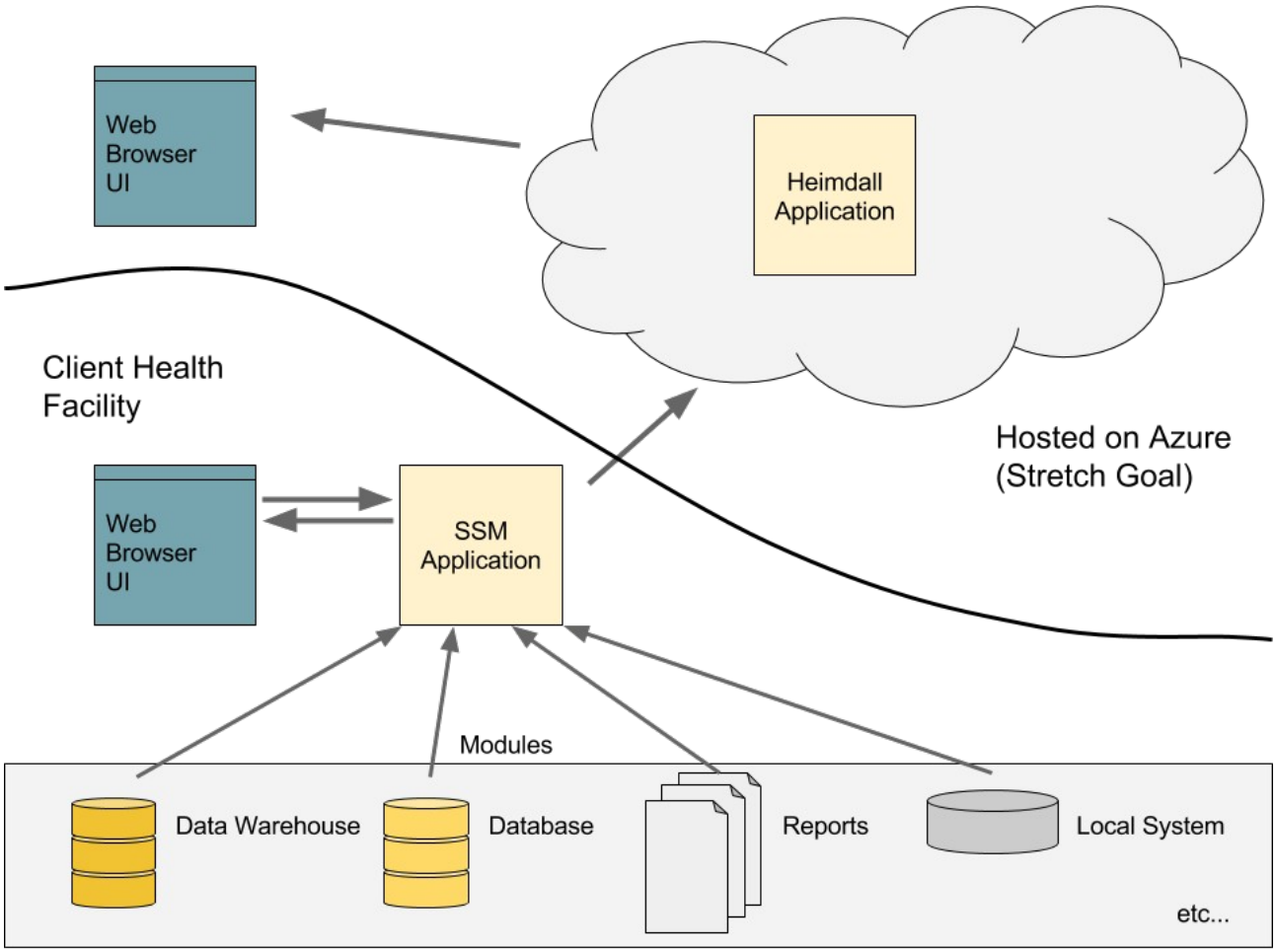
"McKesson ANZ is a fully owned subsidiary of McKesson Corporation, a Fortune 500 global healthcare services firm with 37,000 employees and a $US137 billion turnover. We leverage this world class expertise for the benefit of our customers in Australia and New Zealand.

In Australia and New Zealand we have a specific focus – providing software solutions to help hospitals optimise their operations. The global development centre for our McKesson Capacity Planner software is situated in our region, and we have 40 people across the two countries."

Project Goal

The goal of this project is to deliver a system that monitors and provides two main aspects. Firstly an 'at a glance' health status of the other running systems, such as patient importing, database jobs and log files. Secondly a web based configuration screen to enable or disable jobs, see last run status and set warning thresholds to provide alerts if a job or process is not performing as expected.

Project Overview



Architecture Diagram

The ultimate idea is that SSM will be running in multiple healthcare facilities and reporting back the status of the monitored and configured modules (Data Warehouse, database etc.) so that users can see at a glance the status of all connected facilities.

Hierarchy and Accountability

Key People

Role	Name	Contact
Industry Supervisor	David Taylor	David.Taylor2@Mckesson.com +64 3 379 6662
Course Supervisor	David Weir	WeirD@cpit.ac.nz 940 8324
Academic Supervisor	Mike Lance	Michael.Lance@cpit.ac.nz 0800 242 476

Project Team Members

Name	Contact
Sean Fussell	sef0097@student.cpit.ac.nz 027 7580011
Michael Smith	michaeljaredsmith@hotmail.com 021 0360270
Jessie Velano	coyvelano@gmail.com 020 40506096

Progress Made On Project

Features

Feature	Feature Requirements	Expected Completion	Actual Completion
Web Interface Links with self hosted windows application	Application is a c# program that runs a web server	Sprint 2	Sprint2
Monitors Event logs of the Data Warehouse		Sprint 5	Sprint 5
Provide Event Details		Sprint 5	Sprint 5
Queue Manager Integration		Sprint 5	Sprint 5
SQL Server Reporting Services Integration		Sprint 5	Sprint 6
Web UI provides contextual help for errors	When an error is encountered, steps to further identify or troubleshoot are presented from the McKesson support process.	Sprint 7	Sprint 7
Attractive Web Interface	See 'at a glance' status of SQL Agent Jobs, Rhapsody Information, CIA Define SQL Agent Jobs with name, frequency and warning thresholds Define database and data warehouse credentials Built with AngularJS	Sprint 8	Sprint 8, but further refinements until sprint 11
Dashboard for clients to 'call home'	Cloud based, AngularJS app Stores statuses in non volatile storage	Sprint 9	Sprint 9 Stretch goal scoped
Notifies User when alert threshold triggered	An email is sent to notify user / admin of detected error.	Sprint 8	Sprint 9 Re-scoped due to some clients not allowing outside web access.
Implements SignalR for real time notification to web clients		Sprint 8	Not implemented - De-scoped

Email feature was re-scoped as a useful alternative for continued functionality should clients prevent outbound web access from the application server where the program will run.

SignalR was de-scoped and not implemented as the refresh interval on the AngularJS side of the applications worked sufficiently well.

The stretch goal of having a cloud hosted site that client instances could report to was scoped as a desired feature, as it was the logical 'next step' and would really bump the usefulness of the product, and help sell it to the product owner.

Future Development

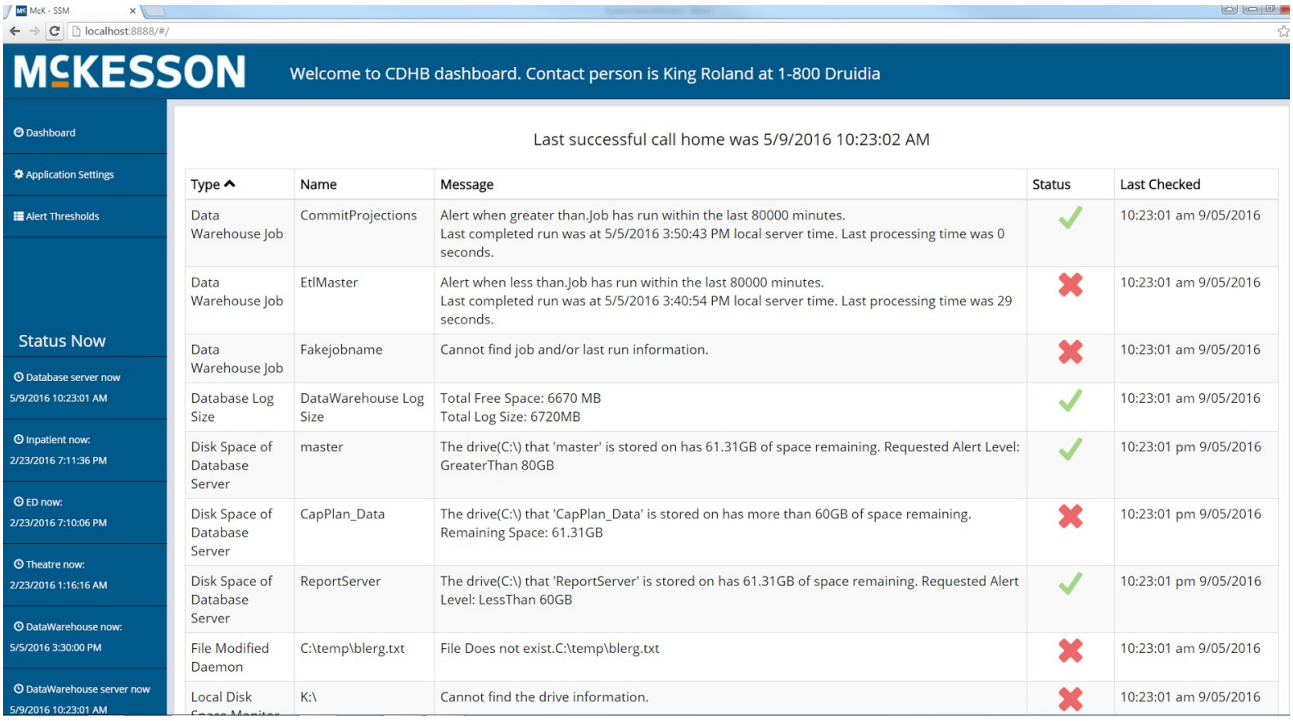
Developed as an internal support tool, clients should not be aware of its existence. It runs in the background to help support and show clients that McKesson is proactive about fixing problems, hopefully fixing the problem before the client is even aware of them.

However, Sales talked about developing it further, which would mean the application would have to go through the established QA programme for client facing applications. A much more rigorous set of tests compared to what an 'internal support tool' would go through. But this did prove that as a working concept, it would be useful to the company and with executive support resources should be more easily justified in assignment.

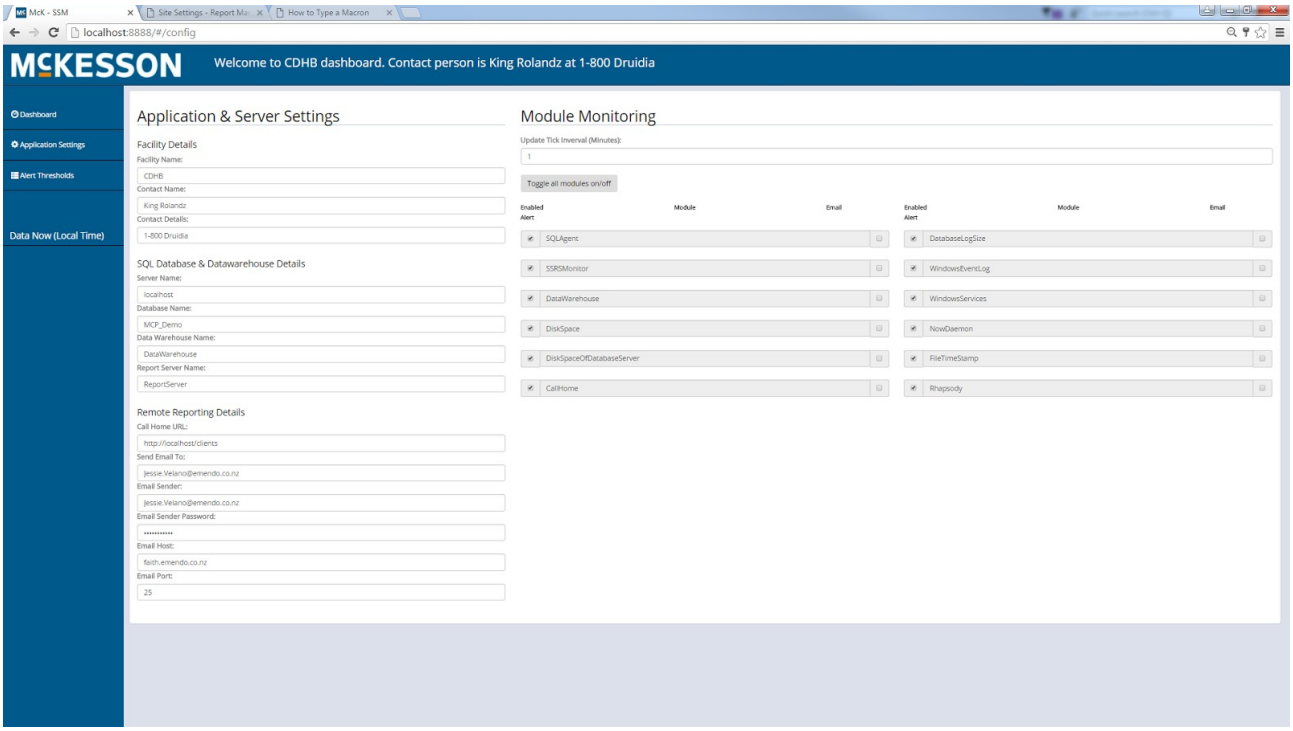
Project Proposal vs Finished Project

The project proposal defined 9 major features with a stretch goal of a web hosted service for at a glance reporting for all (enabled) client instances. The final project completed 8 of those features, and the stretch goal.

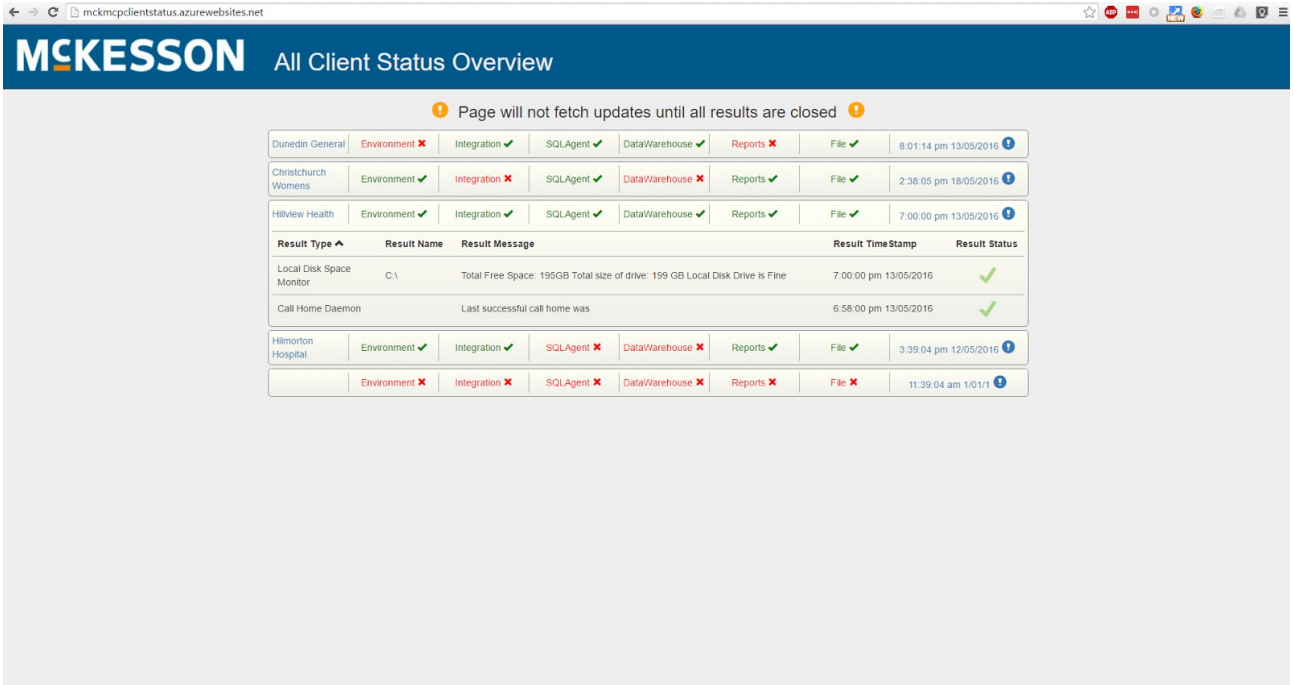
The project was presented to the company on the 12th of May where the product owner, industry supervisor and members from marketing, business intelligence and every other department were in attendance. The completed product was met with interest from everyone and the sales team seemed very keen to use it in sales materials to show how active support can be.



Screen showing the main page of the application, where the status of all defines jobs are reported



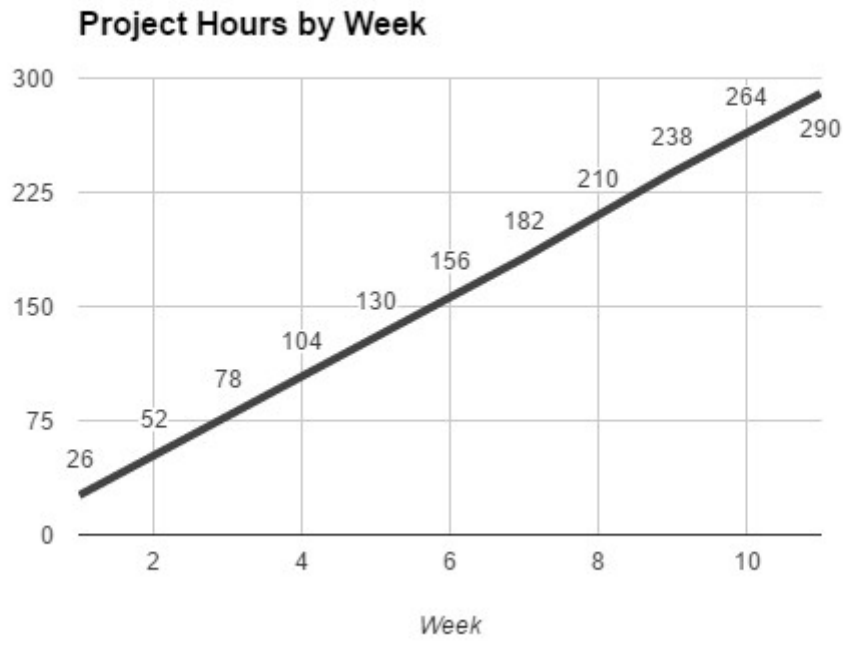
Settings page of the application, client details are set here as well as enabling modules and authentication settings



Azure application, where all client instances results are shown, results expandable to show all information.

For that reason I would say that the project was a success, and in fact exceeding the brief due to the buy-in from other departments. The project realistically, was also a bit of a failure in the sense that proper quality control was not followed, and that I have reservations about its robustness and accuracy given the bugs found in the last sprint.

Project Timeline



Initial project estimates had 10 one week sprints, but at the halfway phase the projected hours would not reach the 288 minimum so another week long sprint was added, bringing the total hours to 290. Initial estimates were off due to lunch time not being counted as billable so over the course of 10 weeks the 4 hours per week added up. This extra week was useful for testing and finishing up the project, so was time well worth spent.

Things I Learned

Code Preface

Daemons AKA Modules: An object that generates a result corresponding to what the daemon is looking for, eg SQL Agent Daemon looks at the SQL Agent jobs and determines run status etc.

Result: A generic object designed to hold arbitrary information, status flag, result name and corresponding daemon type. Eg “Type: SQL Agent, Name: Daily Job 01, Message: Daily job 01 is enabled on SQL server. The job last ran at 2pm Tuesday 22 May with a status code of 1, Status: true”.

Tick: The SSM application operates on a given time interval in which the daemons will do their jobs, in usual use cases this time would be every 10-15 minutes for a semi ‘real time’ set of data.

Build Manager

Using build managers such as gulp and package managers such as Node Package Manager (NPM) the visual studio workspace can be cleaned up by not having client-side files listed in the project. Our use case was limited to the package manager side of things as the web page did require active development, but this tool would automagically copy Angular, bootstrap, jQuery etc from various CDN’s into the build folder, preventing the need to manually do so and preventing the need for the

html itself to externally link to a CDN to get the files which may or may not be available at a future date.

Reflection

Reflection is very cool. This is the first time I have used it and it allowed for a low(er) amount of code to quickly iterate over an object for properties of a given type and add them to a transfer object dynamically, instead of manually having to copy each property to the transfer object, easing maintenance and reducing workload when new features are added, in this case results from data scraping daemons. Further refinement with LINQ is even more powerful, allowing a 'blacklist' of result types that can be excluded if for whatever reason a particular result set doesn't need to be copied to the transfer object. Appendix B has more detailed explanations / examples.

Another planned use of reflection had there been more time to refactor the project was to iterate over the 'static object' looking for an attribute name that a controller can then assign the new result to, as opposed to the big switch case that currently exists for each result type.

Transfer Objects, Data Delivery & Serialisation Witchcraft

The SSM and Heimdall applications were designed to heavily rely upon JSON to transfer data between application and view. This data was also serialised to an XML file for persistent storage in the event of application failure or closing.

Inside the SSM application existed a static object that contained all settings, results and jobs. Upon an HTTP request this object was serialised into a JSON string for usage on the client side, allowing for the view and model to stay consistent along with the content of results.

Some transfer objects used tags to tell the JSON serialiser to ignore a particular attribute, most notably in the case of a enum value being used on a job object. The purpose of this was to prevent an application crash from occurring if the settings.xml file has been altered to have bad values in it. The field was unnecessary for the client side application so was ignored by the serialiser. This control provided some tidying and compacting of the JSON string.

Serialisation of the xml files followed a similar pattern, all public fields would attempt to be serialised unless explicitly told not to. Not setting any rules for the serialiser created problems, especially for arrays, and the end result was that all fields had an appropriate xml tag. Appropriate tags could be [XmlIgnore] or specifying the name of the xml tag [XmlElement("JobName")] or [XmlArray(ElementName = "Daemons")] with an [XmlArrayItem(ElementName = "Daemon")] for children. Appendix F contains example code.

During development a Chrome extension/application called ARC (or Advanced REST Client) was widely used to debug and test the HTTP response and result of the applications. Also previously overlooked during learning was the in built network analysis tools inside web browsers which allowed debugging of routing issues that were not immediately apparent (eg a call 404'd on the server).

Event Listeners

The controller that kept track and controlled the daemon's tick subscribed to an event fired by the daemon when it had finished. This prevent the daemon's needing to have an explicit knowledge of the controller. The interface defining the daemon has the event attached to it, forcing all implementations to use this method.

This was easier than using the observer pattern explicitly, even though .Net does effectively implement it in the background.

Timezones

Timezones, or more specifically storage of times, I learned is a complicated mess and requires serious thinking and planning up front. Mid-way through the project I came to the realisation that the application may be deployed internationally which would mean users would have differing time display formats (dd-mm-yyyy vs mm-dd-yyyy). This turned out to be the least of my concerns.

Client side this is not a big deal to convert, provided the time is stored as a JavaScript Date object, the web browser itself uses the system time zone to determine the offset when using utc or a ms value. The javascript 'to locale' functions This approach however then requires that the time string or value the object is generated from is timezone agnostic, or contains the offset at the time of creation. The timestamps used and generated by our application are UTC based and for the lifetime of useful lifetime of the timestamp this is sufficient as the timestamps are within the last few days under normal use cases. Timestamps are formatted using ISO 8601 format, which contains the offset.

ISO 8601 was chosen to avoid the potential issues of using a millisecond value as .Net and JavaScript use differing epoch bases. Theoretically the JSON serialiser used in .Net made it compatible, and casual observation ruled it so, but I felt that being explicit was a safer option.

The real (and unsolved) problem comes from times stored in the database. These times do not contain offsets at the time of storing, meaning that assumptions cannot be made about the data at all. For example, a time is recorded during May and this time is then examined in December. To try and resolve issue, the user or view could use the current time offset to get the appropriate time, but this result could be out by an hour due to daylight savings being observed in the region. Using the December offset may not be sufficient as the May offset is what is needed to get an accurate historical time.

Appendix D is an excellent article on the subject.

Asynchronous Programming

The SSM application required writing thread safe for some areas. The most notable was when an updated settings object was accepted by the application in the middle of a tick. The application would crash as job parameters for the daemons had been updated mid-iteration. The solution to this was using lock object and prevent a new settings object being applied until the tick is completed. This same approach was also used on any file writes, to prevent multiple inbound settings being written to the disk before the previous had completed.

The one drawback to this approach, at least in the current implementation, is that during a ‘bad’ tick in the sense that there are a lot of SQL timeouts, the web UI will not receive a HTTP response until the settings are applied which could take a long time depending on configuration. I lack the knowledge of HTTP requests and RESTful interfaces to have the application send multiple responses for a proper solution. An alternative would be change the SSM response so that it has been “received but not yet acted upon”.

Further development of the application would have the daemons do their work asynchronously, the current application as it stands uses Hangfire to asynchronously make the controller ‘tick’ but the controllers tick function is synchronous in nature, somewhat defeating the point.

Quality Assurance

Project Methodology

The project will be managed with the agile methodology, with strong pressure to pair program when appropriate. Sprints are weekly and are defined by a sprint goal. User stories are used to generate tasks and the outcome(s) for that sprint. The daily stand-up is scheduled at 9.15am and each week a reflection or retrospective is conducted for the sprint, along with a presentation to the product owner.

The people present in these meetings are the development team, Dave Taylor and guest appearances from other developers or Hayden the Product Owner.

The project team was structured with no defined roles or areas of specialisation, the reasoning behind the decision was that way everyone can experience and get to work on each part of the application or that a team member could specialise for a short period of time such as a sprint or 2 working on server side logic then client side the next sprint.

In practise what happened was I stayed on server side for 5 sprints as everyone was comfortable where they were. At the halfway point an effort was made to shuffle the 'roles' and I worked mostly client side, letting Jessie and Michael take on server side maintenance of the SSM app and primary development of the azure app.

The result of this is that I gained considerable knowledge of how the SSM server application worked as I had spent the most time and developed the majority of the application myself, and when the role switch occurred I learned AngularJS in more detail and was the main developer of the azure client side page.

Tasks are derived from the user story and sprint goal(s) and placed on the whiteboard as sticky notes. By sprint 5 we found a flaw in the implementation, in 2 of the iterations the sprint goal was missed, in both cases due to not focusing on priority tasks or not focusing on tasks that further the sprint goal. This was first encountered in sprint 2 but no action was taken. During sprint 5 we ordered tasks on the scrum board that were most relevant to the sprint goal to make sure the target was hit.

The effectiveness of ranking tasks was effective, despite 1 incomplete sprint occurring after implementation. It did clarify and help me have a clearer understanding of what needs to be done, but sometimes less interesting priority tasks are skipped for lower priority, but more interesting tasks which reduced the effectiveness and purpose of ranking tasks. The ranking system stayed in place until the end and no further sprint goals were missed.

During sprint 4's sprint review we decided to put reminders on the whiteboard of things to improve, or lessons to keep in mind going forward. This included things such as "frequent commits', task breakdown, defined 'done' metrics. The purpose of this was to take the bad/improve from sprint reviews and put them in an easily seen place to act as a reminder.

The most effective of these reminders was to break tasks down further, initially tasks may have been whole features, even though it required development in multiple applications/areas. It also helped morale somewhat by having more sticky notes in the done column by the end of the week. The less effective reminders included 'frequent commits' and 'complete commits' as I often still fail to check in all relevant files for the change, only to find it breaks the build on another workstation hours later, the lack of immediate feedback or effect limited its effectiveness, as does working on multiple tasks simultaneously which added difficulty in checking in the relevant files.

At the end of each sprint a retrospective was held in which the weeks practices and problem were examined to find solutions. This revealed problems such as lack of 'done' metric for whiteboard sticky notes and lack of prioritisation of tasks. These were effective in improving the sprint and

related work-flows as appropriate, helping myself understand product development better.

Coding Standards

Business work-flow is that comments are created for the class and methods, unit tests are created for every method, or at least every class, and that before source control check in are submitted the following conditions are true: the code has undergone peer review, all units tests pass, no compilation errors.

I did comment at least the classes I developed, and all but the most self explanatory methods were commented to their purpose and return values (if any). A gut feeling of 90% of the code is commented appropriately, some of the daemons are not. The usefulness of this was not apparent at first, but it became very useful when looking at other people's code as everyone codes slightly differently and may or may not refactor long code into separate functions.

In hindsight, there are classes I wrote that I would like to go back and refactor, for both testing compatibility and for readabilities sake as some methods are very long.

Peer review of the code before check in, at least the way we did it, I feel could have been improved. What happened is that a person would finish and another would be called over to have a look, the coder would then explain what they did. An improvement, I feel, is that the coder should not explain what they did and see if the other person can figure it out using the comments that are in place to determine readability and validity. The other drawback is we all have the same level of education so there may have been better ways of doing things but we are unaware of them, as a result there was only a couple of instances where improvements were obvious. During a sprint review this was brought up and in some instances more seasoned developers were asked to have a look at code but this was not a common occurrence.

Unit Testing

Testing was not done to any satisfactory standard. In the end there were about 9 unit tests, 2 of which always failed due to lack of knowledge in how to compare objects (if that even was the correct way to do it). The industry supervisor was very hands off this project, in contrast to other ones so without enforcement we as a team fell back into old bad habits and we justified it away for reasons such as ‘we don't know how to test it properly due to the external nature of the data’ and ‘we don't all have the same data on our local machines’. This really came back to bite us at the end, where a few large flaws were revealed by chance observation.

3 particular bugs of importance were noticed in the last sprint, the first being that a SQL query was not returning the correct result (but to a casual observer it looked like a proper result), the second was that a daemon was reporting incorrect results due to the comparator not working properly, finally the daemon that monitors file timestamp was not looking at the date part of the timestamp, only the time. Both of these could have been caught early with proper and comprehensive unit tests. The timestamp one in particular slipped by as the test files created were not set so far in the past to allow for proper test, it looked like it worked as the files were very recent and a false assumption was made.

The correct way to handle this would have been to create mock data providers that act as stand ins for the external data sources, which may or may not be consistent across work stations. By the time this was realised it was far too late to introduce such a large change due to weeks of code that was not created to work with such a system. For example, the structure of the daemons makes testing with mock data providers difficult, different coders had different styles of doing things, and the interface was designed to be simple so it doesn't micromanage the daemons. Fixing this would be a

large undertaking, and would be at least 1 sprint worth of work. To do this effectively the interface should be developed further to force implementations to be more testing friendly. See Appendix C for details.

It sounds like I'm justifying the (poor) choices made, and I would like to point out that that is not the case, testing really would have saved the project some grief and stress at the end. Bad habits were created during my time at polytech and it was only when I encountered a ‘real’ application that it became apparent how important testing is, as up until now I have gotten by without it. I have certainly learned my lesson.

Problems Encountered

In 2 of 7 iterations the sprint goal was missed, in both cases due to not focusing on priority tasks or not focusing on tasks that further the sprint goal. This was first encountered in sprint 2 but no action was taken. During sprint 5 we ordered tasks on the scrum board that were most relevant to the sprint goal to make sure the target was hit. Sprint 6 was completed, but sprint 7 was not. This occurred due to starting late on the relevant sprint tasks and a belief that it would not take long to implement the remaining relevant features.

Unit tests, or lack thereof have been a problem during development, and has come up in multiple sprint reviews. Part of the problem stems from being unsure how to test or what tests are appropriate, as all classes and methods have the lowest visibility level required, leading to only being able to test the end result and inputs, which does not help with debugging to see why the unit tests failed.

During sprint 4’s sprint review we decided to put reminders on the whiteboard of things to improve, or lessons to keep in mind going forward. This included things such as “frequent commits’, task breakdown, defined ‘done’ metrics. The purpose of this was to take the bad/improve from sprint reviews and put them in an easily seen place to act as a reminder.

The effectiveness of ranking tasks still requires measurement to see if it is effective, with 1 complete and 1 incomplete sprint occurring after implementation. It does however clarify and help me have a clearer understanding of what needs to be done, but sometimes less interesting priority tasks are skipped for lower priority, but more interesting tasks which reduces the effectiveness and purpose of ranking tasks.

The reminders placed on the whiteboard are effective in getting myself to follow some of them, which resulted in more frequent but smaller commits for example, but I often still fail to check in all relevant files for the change, only to find it breaks the build on another workstation. I don't look at that section of the board as often as I should.

At this stage of the project (halfway), the lack of unit tests I don’t feel is an issue. I justify this by believing that the only things that can be tested are the inputs and outputs of whole classes, as the internal methods and properties are at a lower visibility. This could be resolved by using ‘internal’ visibility levels, but am yet to be sold on whether it would help much. The other factor is the data sources for the tests are external programs that may or may not be present, with a multitude of possible values, making it feel more effort than it is worth. The solution here is have mock data providers that emulate the various external data sources. Its likely I don’t see value or justification in this partially out of not knowing how to implement it properly for this program, the remainder simply not having seen the benefit to developing tests.

The primary issue encountered in the second half of the project was the impact of the lack of testing, this allows several bugs and flaws to go unnoticed until the end. These were fixed but at the end of the project there is no ‘proof’ that it works in the sense of passing unit tests with mock data. The application ‘looks’ right but certain configurations may cause problems as testing in the sense of

widely different scenarios was not explored as far as it should have been. Prime example being the file timestamp bug where only part of the timestamp was compared and would result in an incorrect result.

Risks

The initial risk evaluation for the project proposal was carried out in an IS301 style framework; where risks are ranked on probability and impact. During this time a lot of the risks were very academic due to myself not being exposed a 'real' project with any kind of scale. The real risks identified by this process were already covered by company practises such as source control and using fake or test data for development. These risk evaluations were carried out in sprints 1, 2 and 5. In effect, this approach was not useful and was not used as a guidance or control method, the risk management inherent in Scrum became the primary tool to manage and control risk.

The daily stand up and weekly retrospectives made risks or issues highly visible and allowed the team to mitigate or remove the risk as it cropped up on a daily frequency. Tasks where on a whiteboard in a highly visible fashion where team members could see at a glance how the project is progressing and what is left to do, allowing for opportunity to refocus efforts if required. Microsoft has a Solutions Framework (MSF) which follows the iterative and highly visible Agile approach to project management where it is about "empowering team members, delivering incremental value, staying agile and learning from experiences".

By following this process I felt that risk was managed adequately.

Previous Courses

Programming

I feel that the first course that was 'real' OO programming was PR282 Java, as previous to that with PR280 JavaScript, it was written procedurally, plus at that stage I still didn't really know what I was doing. Some of the SSM app uses plain JavaScript (as opposed to the AngularJS specific flavour) so familiarity with the language was useful. At the PR282 stage I knew enough to not feel completely out of my depth and was ready to deal with a real IDE and concepts such as MVC. Alongside PR282 I did PR294 Server Side Programming where I was introduced to the client server model which helped for designing the SSM application. PR283 taught me C# and during it I spent more time trying to figure the 'right' or 'nice' way to build applications as opposed to trying to figure out how to get them to work at all. It also taught me some language features such as LINQ which the SSM application uses for finer control over iterations. PR301 continued to improve my architecture designs through use of patterns and common pitfalls or 'bad smells' to avoid in code.

PR203 gave me enough SQL skills that I could complete this project, albeit relying on T-SQL documentation heavily.

IS

IS201 covered differing methodologies, software life cycle phases such as requirements gathering and analysis, integration and maintenance. The requirements were already drawn up by the industry supervisor and his team and the management methodology already decided. The course, previous with past industry experience, furthered my agile understanding.

In IS301 I learned some risk management tools and processes, none of which were used in the project, but realistically should have been used. The course also further refined requirements

gathering through strategic analysis, but this wasn't used as the feature requirements were already set.

Improvements or Recommendations

I think the biggest drawback to the programming courses in general, is that at the end of the course the student only has 1 application, which I think is appropriate as a final result piece, but the learning that goes into it varies incredibly and I feel I would have learned more by having multiple, smaller applications each week (or two) as practise, in addition to the 'main' app the class develops. PR280 JavaScript did this with the iteration approach with the 'PSP'. Personally I feel repetition is what commits things to memory best, and during SE101 and SE102 I made a point of manually writing / typing code rather than copy pasting what Amit provided so that it would stick better in my mind.

The drawback to the one application approach is that the student can and will stumble through it, and by the end learn enough to say 'I know better ways of doing this now' but then lacks the opportunity to fix it. Smaller bite sized applications would help syntax/language stick to memory better and could provide a variety of differing requirements. During PR282, PR283 and PR294 we had syntax exercises, which could be interpreted as 'small applications' but what I had in mind was bigger. See appendix E for examples.

Perhaps the ultimate goal would be requiring the student develop an assistant application that is needed to complete the final assignment for the class. If the final assignment was some some of data processor, a helper app may be needed to retrieve the data and/or put the data into some kind of intermediary format that the final app is to use.

The client/server model is a much larger topic than is covered in PR294 or more generically I feel there is room to expand upon inter program connectivity, such as databases, web server or not covered: communication between processes. Basically connecting things to things, how could a developed program fit into the wider view. Only students who took PR294 got a taste of that kind of work, which I personally really liked, having my PHP application pull from a small but comparatively large database and serve to a web page really showed how things can work together.

Unit testing was taught in PR282 and PR283, but it was only during this project did it click how useful it really is, especially in regard to dealing with external data sources. Again, how can it fit into the larger picture as testing public methods feels limited.

Finally, in regard to IS, I struggled with why I should have to learn it, as in my mind that subject matter is the realm of project managers etc. I did not expect when enrolling that 'non technical' things were such a big part of the curriculum, I had to go to a lot of effort to avoid them. This is my fault in a lot of ways I admit, I certainly could have/should have been more proactive in finding out what the degree was all about. I also may not be the typical student, came in from industry and more or less knew I wanted to switch career and become a programmer, or network admin. This mis-match of expectations did cause some stress.

One final thing I feel would be a great improvement is the creation of courses focusing on the hardware side of programming. With micro-controllers and internet of things aspects. The idea being that the wider view or bigger picture of how things can fit together translates into the physical world as well.

Essay Conclusion

It may be a bit of a cliché, but what I have learned from these experiences is that Scrum only works if you do it properly and enforce the rules. If that fails then what is left is effectively a free for all with no real structure. However, when it works, it works well in my opinion. My experience at McKesson showed me the potential of a well run team as on the whole, the work was owned by all of us, the immediate feedback provided a better product and clarity of tasks. They do Scrum properly.

Closing Remarks and Conclusion

This project taught me a lot about working a real software project and certainly made some of the lessons tutors were trying to teach very clear. I feel I gained significant experience in varying technologies or frameworks such as AngularJS, client/server architecture, reflection and generally getting an idea where an application may sit in the larger picture.

Going into these courses at polytechnic I already had an idea that I wanted to write “boring business software” as opposed to “trendy” video game application or more consumer facing products. My distaste for the latter came from previous industry experience, and my liking of the former is about 'getting things done' or making something that’s useful, ideally that helps a bigger process run.

Throughout the courses I formed the opinion that testing was academic and not necessarily worth the time, this project severely challenged that notion, and it finally came back to bite me in the end. What really pushed it home was the inconsistency of the external data sources I used in the application, and compounding this was when I realised it was a problem and I couldn’t do much about it without serious re-engineering effort. Up until that point I justified it away with *‘we don't know how to test it properly due to the external nature of the data’* and *‘we don't all have the same data on our local machines'.* The uncertainty of how to deal with it prevented action and old habits were fallen back into.

Another thing revealed to me was that my usual efforts to over engineer or create the most robust/versatile solution was not needed and took time more effectively spent elsewhere. This experience has taught me to not try and create the best solution and that when a second use case comes up, use an 'if', a third use case is when engineering effort is justified for a better solution.

If I were to redo this project, I would certainly enforce the company process on the team, unit test and don’t check in until everything passes and ask for help more often when I need it. I feel I did OK at asking for help, but in hindsight I may not have been direct or clear enough that it was a roadblock and I couldn't solve it on my own.

Despite those draw backs I feel my efforts on the project were significant and that I learned a great deal. Having the JSON data seamlessly jump from client to server to file system the first time was something I remember fondly.

Comparing this experience to the classes I have taken, the biggest difference is the sense that the program was a part of something larger. It was linking up with an already existing system and adding value. Course applications were created form the ground up with no already existing rules or datasets, I had full control over the program.

I certainly learned more through experience, challenges that are outside of my control become apparent and I cannot get away with some of the things I did in an academic environment. Some of these lessons can only be taught the hard way as unless the problem actually occurs then I'm less likely to think its a real issue.

In closing I really enjoyed challenging myself and would like to thank Dave Taylor and all the other kind folks at McKesson for taking me on for the project. It was fun and I learned a lot.

Appendix A: Sprint Plans & Reviews

Sprint 1 - 22 - 26 February

Sprint Goal	"learn stuff"
Deliverables / Requirements	SignalR Angular SQL Server Already existing solutions IIS & ASP.NET

Sprint goal achieved

Reflections :

Good	**Bad**
• Had working examples • Support was very useful • Visualised design/ architecture • Set up environment	• Lack of "done" metric for tasks
Improve	**Escalate**
• Sprint goal • Task definition	

Sprint 2 - 29 February - 4 March

Sprint Goal	Angular web page that we can save configuration data with
Deliverables / Requirements	Web API Read settings from file Web API Update settings file Angular Model Configure Page Status Page (Mock data)

Sprint goal not achieved

Reflections :

Good	**Bad**
• Task definitions improved • Get and Set XML files pairing	• Broken UI • Failed to focus on functional UI
Improve	**Escalate**
• Focus on delivering functionality • Team communication, who is doing what, who needs help	

Sprint 3 - 7 - 11 March

Sprint Goal	I can see the status of a SQL Agent job in the web UI
Deliverables / Requirements	Link to SQL Agent Display real data All basic web pages

Sprint goal achieved

Reflections :

Good	**Bad**
• Sprint Goal Achieved • Class Diagram • Implemented comments • Used scrum board	• Left Door Unlocked • Comments for HTML & CSS lacking • Missing Unit Tests
Improve	**Escalate**
• Clarify names of attributes / methods • Check builds (some items were missing) • Keep upto date with source control	

Sprint 4 - 14 - 18 March

Sprint Goal	fully working web UI (behaves properly does not need all data sources or massaging)
Deliverables / Requirements	No non working or broken elements on UI All developed functionality or configurability is exposed to UI

Sprint goal achieved

Reflections :

Good	**Bad**
• Sprint Goal Achieved • Team member role shuffle (good to see other parts of system) • Enlisting Roy's help at the start • Learnt from last week - function before form	• Unit test coverage sparse • Some tasks were not required for sprint goal • Changes were not committed properly
Improve	**Escalate**
• Ensure tasks are clear • Be vigilant with checking in (smaller tasks could help) • Be careful with over engineering	

(SSMStatus) & consider other implementations • Set weekly 'lessons' on board to keep in mind for coming weeks	

Sprint 5 - 21 - 25 March

Sprint Goal	Refactor app & UI and working UI
Deliverables / Requirements	

Sprint goal achieved

Reflections :

Good	**Bad**
• Sprint Goal Achieved • Broke tasks down • Cool looking UI • Learned new technology • Commenting improved	• Some excess code - keep it simple • Michael Sick
Improve	**Escalate**
• TDD (Test Driven Development) • Focus on sprint goal (KISS)	

Sprint 6 - 4 - 8 April

Sprint Goal	Fully working UI with multiple jobs per daemon & SSRS integration
Deliverables / Requirements	Can add an arbitrary number of jobs to all daemons

Sprint goal achieved

Reflections :

Good	**Bad**
• Sprint Goal Achieved • Small commits • Prioritised items	
Improve	**Escalate**
• Incomplete commits • Pair programming • Outside group code reviews	

Appendix B: Reflection

The best example of reflection use in the project was the call home daemon which iterated over all fields with the type of DaemonResult and added those into a transfer object to be sent to the Azure application.

Code is as follows:

```
foreach (PropertyInfo property in
SSMStatus.Instance.GetType().GetProperties().Where(x => x.PropertyType ==
reflectionCompare.GetType()))
            {
                if (property.GetValue(SSMStatus.Instance) == null)
                {
                    continue;
                }
                DaemonResult ReflectedResult =
(DaemonResult)property.GetValue(SSMStatus.Instance);

                if( daemonblacklist.Contains(ReflectedResult.Type))
                {
                    continue;
                }
                if (ReflectedResult.ChildResults == null)
                {
                    result.AllResults.Add(ReflectedResult);
                    continue;
                }
                else
                {
                    foreach (DaemonResult childResult in
ReflectedResult.ChildResults)
                    {
                        result.AllResults.Add(childResult);
                    }
                }//end if else
            }// end foreach
```

In effect, this allowed the daemon to continue operating without changes when new result fields are added to the static object. It also saves a significant amount of space when compared to using a chain of ifs or switch cases, which would require updating every time a new daemon is created. The master controller in the project uses this approach and it takes up significant space. The planned refactor is that the master controller will look for a field with the same name as the daemon and update the result from there, instead of the large switch case as shown below.

```
DaemonResult res = e as DaemonResult;
switch (res.Type){
    case EnumDaemons.SQLAgent:
        SSMStatus.Instance.SQLAgentResults = res;
        break;
    case EnumDaemons.WindowsEventLog:
        SSMStatus.Instance.WindowsEventLogResult = res;
        break;
    case EnumDaemons.DataWarehouse:
        SSMStatus.Instance.DataWarehouseLogResults = res;
        break;
    case EnumDaemons.WindowsServices:
        SSMStatus.Instance.WindowsServiceResults = res;
        break;
    case EnumDaemons.SSRSMonitor:
        SSMStatus.Instance.SSRSResults = res;
        break;
    case EnumDaemons.DataNow:
        SSMStatus.Instance.NowResults = res;
        break;
    case EnumDaemons.DiskSpace:
        SSMStatus.Instance.DiskSpaceResults = res;
        break;
       ***|
   ***  4 more switches in here
       ***|
    case EnumDaemons.DiskSpaceOfDatabaseServer:
        SSMStatus.Instance.DiskSpaceOfDatabaseServerResults = res;
        break;
    case EnumDaemons.Rhapsody:
        SSMStatus.Instance.RhapsodyResults = res;
        break;
    case EnumDaemons.GroupStatus:
        SSMStatus.Instance.GroupResults = res;
        break;
    default:
        Logger.Log("Unknown daemon type result");
        break;
}// end switch
```

Appendix C: Mock Data Testing

The interface that all daemons implemented is as follows:

```
public interface IDaemon
    {
        /// <summary>
        /// A string to store the help text that is collected from the SettingsXML
when the daemon runs
        /// </summary>
        string HelpText
        {
            get;
            set;
        }

        /// <summary>
        /// No return, tells daemon to start processing
        /// </summary>
        void LoadSettings(SSMStatus options);

        void Execute();

        /// <summary>
        /// All daemons use handler to notify of changed results
        /// </summary>
        event EventHandler PropertyChanged;
    }
```

There is not a lot of rules to it and as a result coding the daemons in such a way to be friendly to mock data sources. Some daemons are coded in such a way that all processing happens in the execute function eg

```
Function execute(){
    var rawResults = functionToGetRawResults();

    //process things here to create result object

    OnPropertyChanged(result);
}
```

Whereas other daemons will work differently:

```
Function execute(){
    var rawResults = functionToGetRawResults();

    processRsults(rawResults);
}

Function processResults(results){
    //process things here to create result object
    OnPropertyChanged(result);
}
```

This subtle difference would make all testing easier, provided the processResults was publicly exposed on the interface as a mock data provider could substitute in for the sql database and provide the same type of results without actually needing a sql instance. There are some problem this could raise however, the internal classes that are used only by the daemon would have to be moved into a more visible scope and it takes away from the 'go do it and I don’t care how' approach that we wanted to take for the daemons.

Appendix D: Timezones Are A Pain

https://blogs.msdn.microsoft.com/bartd/2009/03/31/the-death-of-datetime/

The Death of DateTime?

March 31, 2009 By bartduncan

SQL Server 2008 added a new data type named "datetimeoffset". This is similar to the old datetime data type, with the following significant differences:

- Internally, the time is stored in unambiguous UTC format
- The local time zone offset is stored along with the UTC time, which allows the time to be displayed as a local time value (or converted to any another time zone offset)
- The data type is capable of storing more precise times than datetime

So, when should you use datetimeoffset, and when should you use datetime? The first answer is that you have no choice at all if you aren't using SQL 2008, since datetimeoffset was first added in this SQL version. If you are working with SQL 2008, let's address the question by examining some problems with the older datetime type:

Problem 1: DateTime is inherently ambiguous

Suppose you are a consultant looking at an existing table with a datetime column. A row in this table tells you that some critical event occurred at 4:35pm. Is this the server's local time? Local time for the end user's machine? GMT, or Coordinated Universal Time (UTC)? Local time for a time zone selected by convention that may or may not match the server's local time zone? Here in Microsoft, for example, some systems were designed to store Pacific time by convention, even though in some of these cases the SQL Server may reside in a data center that isn't on the west coast. Other databases store UTC times, again by convention adopted by whoever designed those systems. So this is the first problem: datetime is ambiguous. A datetime value by itself actually does not identify a particular moment in time; it takes on an clear meaning only when you interpret it in the context of some assumed, and usually unenforced, time zone.

Problem 2: It is impractical to convert historical DateTime values to/from time zones

If you've ever built a data warehouse that consolidated data from several data sources, you may have struggled to convert various ambiguous local time representations into a consistent form. Some data sources store their times as UTC, some store local server time, others use a local time based on some end user's time zone. You could just cram all of those values into a datetime column in your warehouse, but it would be impossible to interpret the times in a meaningful way. No matter what time zone you selected as a lens through which to interpret the data, it would be wrong for some values.

So you must expend some extra effort to figure out the implicit time zone for every data source in your warehouse, and then you have to write code that converts all of these various times to some

consistent time zone, likely UTC time, in the central data warehouse. If that was the end of it, it would be bad enough. But Daylight Savings Time makes it next to impossible to do this conversion in a generic way. The problem is that different locales have different Daylight Savings Time rules. Many parts of the world don't honor DST at all, some places do honor it but use half-hour offsets instead of full hour offsets, and various places begin and end DST on different dates. Here's an concrete example: suppose you have the local datetime value 2008-03-05 08:30:00 in the database, and you need to convert this to UTC time. By interviewing the right DBA or by examining source code, you have determined that this database stores datetime values using local server time. The local server is in the Pacific time zone, and you've found that you can use a T-SQL expression like this to determine the current time zone offset for the local server:

```
DATEDIFF (minute, GETUTCDATE(), GETDATE())
```

This tells you that the local server time is 7 hours behind UTC right now, so you should be able to add 7 hours to the local time to get the equivalent UTC time, right? That would be wrong; Daylight Savings Time is in effect in the U.S. today (April), but when this datetime value was collected back in March of last year, DST was not in effect. The correct time zone offset to use is UTC minus 8 hours. So you have to have knowledge of what the local time zone offset would have been on arbitrary past or future dates, and bake this knowledge into your conversion routine. If your data comes from a variety of locales, you have to have correct information about the time zone rules in every place your data comes from.

To make matters worse, the rules for DST may change from year to year in the same locale due to legislative changes, so you have to capture different sets of rules for different ranges of dates within each region. Have you taken into account the fact that (most of) the state of Arizona doesn't use DST? Or that Indiana didn't use DST at all prior to 2006, but you do need to adjust for DST for any data that was captured on a server from Indiana after 2006? Does your conversion routine account for the fact that Daylight Savings Time in the U.S. was lengthened by about a month starting in 2007? This problem isn't unique to the United States, and the situation can be even more grim if your data comes from more than one country. DBAs and developers in China, India, and Japan get off a little bit easier because DST is not observed in those countries, but they still have the problem to some degree if they ever need to consume data that originated in other places or push their data to a consumer in a different country.

Finally, there is small time window each year that will defeat even the world's most intelligent time zone conversion routine. In 2009, DST in the United States will end at 2:00 am on November 1st. At that time the clocks will roll back an hour, to 1:00 am. In other words, each year there is a one-hour window during which times like 2009-11-01 01:35 am will actually occur twice. That ambiguity is completely intractable.

Problem 3: If you do manage to convert DateTime values collected in various places to a single time zone, you lose important information

Let's suppose that a database you are working with stores UTC times. Good for you: all of your times are unambiguous. But unless your users all live in London or Lisbon (and DST is not in effect), UTC is generally not very meaningful to a user. You could theoretically convert the times to the end user's local time zone (if it weren't for the inconvenient fact that this is impractical, as we just discovered). But what if you wanted to present the time in local time relative to the place where the timestamp was captured? For example, suppose you wanted to show records from a

consolidated server health log as local times for the server where they were captured. You can't. The information about the current local time zone offset at the moment the timestamp was collected was lost when you converted the difficult-to-work-with local time into that nice, pure UTC time.

These three problems can combine to create a real mess. Frankly, I think I might consider a career change if I was tasked with solving all of these problems in a large-scale project that consolidated historical data from many different places. Of course, you might be thinking, *My datetime values are all local server time; their meaning is perfectly clear to me*. Well, one day your company may expand and your little homegrown system might need to handle data from more than one region. Or you might need to import the data into a new system when your solution is thrown out for being too provincial :). Or the data in your local database might turn out to be needed in some central data warehouse that consolidates data from a variety of sources. You can save yourself and your successors some grief by using the more robust datetimeoffset data type from the start.

When to Use DateTimeOffset?

Because the datetimeoffset data type stores a UTC date internally, it's free of the ambiguity that causes problems #1 and #2. And because it also stores the time zone offset that was current at the time the timestamp was generated, it doesn't suffer from the data loss problem that you face if you store times as UTC datetime values (problem #3). In other words, with the same datetimeoffset value you can represent the value as a local time or easily convert it to a UTC time. SQL will do the right thing if you compare two datetimeoffset values, even if the values were captured from systems with very different time zone offsets.

So, let's return to the original question: When should you use datetimeoffset instead of datetime? The answer is: *you should almost always use datetimeoffset*. I'll make the claim that there is only a single case where datetime is clearly the best data type for the job, and that's when you actually require an ambiguous time. For example, if you wanted a column to record the fact that all stores in a chain should open at 8:00am local time (whatever the local time zone may be), you should use datetime. But any time you want to store a value that represents an *absolute moment in time*, you would be better off using datetimeoffset. For most applications, that means that just about everywhere you currently use datetime would be a good candidate for datetimeoffset.

Please don't beat me up over the fact that SQL 2008's DMVs still use datetime :). This is a known problem, and it's on the books to look at for future versions. We're facing the same problems that you'll face in your existing systems: it's hard to make a system-wide datatype change that doesn't break someone somewhere. For any brand new SQL development work you do, though, I encourage you to pause and consider your choice carefully before using datetime. It may be an appropriate choice in some cases, but most of the time you'd probably be better off with datetimeoffset.

Finally, be aware that .Net 2.0 SP1 added support for the datetimeoffset data type, so you can round-trip this nice new data type between SQL and a client app without any fuss.

Appendix E: Bite Sized Apps

I think the biggest drawback to the programming courses in general, is that at the end of the course the student only has 1 application, which I think is appropriate as a final result piece, but the learning that goes into it varies incredibly and I feel I would have learned more by having multiple, smaller applications each week (or two) as practise, in addition to the 'main' app the class develops. PR280 Javascript did this with the iteration approach with the 'PSP'. Personally I feel repetition is what commits things to memory best, and during SE101 and SE102 I made a point of manually writing / typing code rather than copy pasting what Amit provided so that it would stick better in my mind. Smaller bite sized applications would help syntax/language stick to memory better and could provide a variety of differing requirements.

Such apps could be

- Client / Server web page that displays the content of a text file on a web page as a text area box, with send /receive functionality
- Application that generates 'fake data' for sql testing purposes
  - for example created 2000 entries with a random name, email address etc from defined lists
- Creating a small card game, card games have already defined rules so recreating a small feature should not be overly difficult
  - The pulp alley character creator from PR301 was excellent for this, it was perhaps too big but a subset could work well
  - Other things could include a small round of combat from table top games
- A clock application that reminds the user to get up and exercise every X period of time or has todo items associated with times

Perhaps the ultimate goal would be requiring the student develop an assistant application that is needed to complete the final assignment for the class. If the final assignment was some some of data processor, a helper app may be needed to retrieve the data and/or put the data into some kind of intermediary format that the final app is to use.

Appendix F: Serialisation

Class showing XML serialisation property definition.

```
    [XmlRoot("Settings")]
    public class SettingsTO
    {
        [XmlElement("ClientName")]
        public string ClientName { get; set; }

        [XmlElement("ContactName")]
        public string ContactName { get; set; }

        [XmlElement("ContactNumber")]
        public string ContactNumber { get; set; }

        [XmlElement("ServerName")]
        public string ServerName { get; set; }


        [XmlElement("DatabaseName")]
        public string DatabaseName { get; set; }


        [XmlElement("DataWarehouseName")]
        public string DataWarehouseName { get; set; }

        [XmlElement("ReportServerName")]
        public string ReportServerName { get; set; }

        /// ********************
        /// DAEMONS Enabled Area
        /// ********************
        [XmlArray(ElementName = "Daemons")]
        [XmlArrayItem(ElementName = "Daemon")]
        public List<DaemonEnabledTO> Daemons { get; set;}
    }
}
```

COURSE MANAGEMENT

Unsure what mark I am aiming for

- ❒ established
- ❒ actively maintained
- ❒ extensive
- ❒ exceptionally effective
- ❒ displaying excellent control
- ❒ initiating communication throughout its execution.

THE PROJECT

Aiming for 5

- ❒ ✓ completed the project
- ❒ ✓ to the industry supervisor's satisfaction
- ❒ ✓ demonstrating an exceptional grasp of the subject.

CONTENT OF THE LEVEL 200 AND 300 COURSES

4 or 5

- ❒ ✓ correctly identified
- ❒ ✓ evaluated content
- ❒ ✓ shows material has been applied in a relevant and innovative manner.
- ❒ ✓ perceptive content recommendations

QUALITY ASSURANCE PROGRAMME

4?

- ❒ ✓ created
- ❒ ✓ maintained
- ❒ applied
- ❒ comprehensive
- ❒ ✓ in-depth understanding
- ❒ ✓ critically analysed
- ❒ ✓ insightful conclusions

RISK MANAGEMENT PROGRAMME

3?

- ❒ ✓ created
- ❒ maintained
- ❒ ✓ applied
- ❒ comprehensive
- ❒ ✓ in-depth understanding
- ❒ ✓ critically analysed
- ❒ insightful conclusions

METHODOLOGIES ESSAY/REPORT

Happy with 1, maybe 2

- ❒ extensively referenced accepted theory
- ❒ ✓ industrial practice
- ❒ ✓ related
- ❒ exceptional standard.

REPORT

4 or 5

- ❒ ✓ polished
- ❒ ✓ imaginative
- ❒ ✓ clearly and fluent
- ❒ ✓ insightful
- ❒ ✓ accurate grammar and spelling.
- ❒ ✓ very full analysis of performance

PANEL

- ❒ confident
- ❒ skilled communicator
- ❒ presented clearly and logically
- ❒ responded clearly and logically
- ❒ perception in appropriately responding to supervisors' reports and questions.

POSTER

4 or 5

- ❒ ✓ Imaginatively
- ❒ ✓ professionally
- ❒ ✓ displays project's outcomes
- ❒ conveys learning achieved.

The mark I feel I achieved is 86%