Complimentary and Alternative Registry

Out By Five

Corey Batten Alex Howland Alexander Kozak Eric Thompson

Project Sponsor

Nathan Claes

Faculty Coach

Dr. Stephanie Ludi

Project Overview

The US Department of Veterans Affairs currently offers many forms of treatments and care for veterans. It has been found that there may be merit to other, less orthodox, treatments such as acupuncture, yoga, or meditation. These are known as Complementary or Alternative Medicines. Veteran's Affairs currently makes some of these treatments available to complement other medicines or in place of ineffective medicines. They, however, do not currently track these treatments, what they are used to treat, or their efficacy. As the next step into the use of these and similar treatments, Veterans Affairs is in need of a system with which these treatments and their outcomes may be easily tracked.

This project seeks to provide the next step. The goal of this project was to create a registry system that would allow for these treatments to be tracked quickly and easily by medical providers and allow for analysts to perform various analyses of the data. Medical providers may vary greatly in technical experience, from no experience to proficient. Analysts are expected to be much more experienced. As a result, the primary goal of this system is to ensure that Medical Providers are given the simplest and fastest forms of entry. The system will be determined successful if it provides a highly usable interface for medical providers, stores all necessary and appropriate information regarding treatments, patients, and outcomes, and meets the system constraints imposed by Veterans Affairs.

Basic Requirements

The CAM registry has a number of high-level functional requirements. First, users must authenticate to access the system, to preserve patient information confidentiality. To facilitate this, administrators must be able to create users in the system. Secondly, users designated as medical providers should be able to create patients in the system, and add, edit, and update their statuses as well as track patient progress. These statuses will often be tied to CAM treatments.

Thirdly, program analyst users must be able to retrieve data from the system to run statistical analyses to determine the effectiveness of specific CAM treatments for various disorders. Administrators and program analysts must be able to change the types of data collected about patients by medical providers, so that they can experiment with different data sets that may or may not be later determined to correlate with CAM treatment success.

Program analysts and system administrators can be assumed to have varying levels of technical expertise, leaning strongly towards the more technical. Medical providers, however, are expected to have little technical knowledge and little time to learn it. Therefore, the system must be quick and easy to use and simple to master.

Constraints

As specified by the project sponsor, the application must be web-based using ASP.NET and Microsoft SQL Server 2008. The system must also comply with a number of government regulations:

- 128-bit AES encryption of data
- HIPAA medical privacy
- Section 508 Government Accessability requirements

Additionally, our project must be completed within two academic quarters (approximately 6 months) at RIT, while all team members are likely attending 2-3 other classes and may have jobs.

Development Process

The team used an incremental waterfall process for the development. The system was broken into multiple increments of requirements to be implemented. We then estimated how long the increments would take and followed a waterfall development process for each increment. This process was accepted by the sponsor, but was not mandated. Our process worked well with communications with the sponsor, allowing regular progress updates through the process and a free flow of ideas between developers and the sponsor. The important roles were determined to be project manager, risk manager, planning and tracking lead, sponsor contact, requirements lead, and design lead. These were divided up amongst the team members based on willingness and expertise.

Project Schedule: Planned and Actual

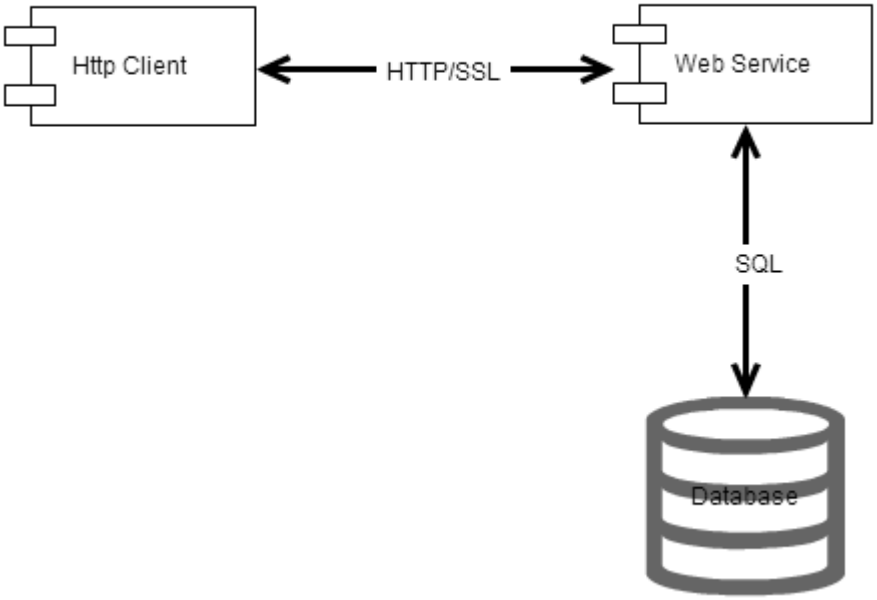
In order to develop the project schedule, we created a Work Breakdown Structure. We broke each task into the smallest logical chunks, and individually estimated the time necessary to complete each one. In order to aggregate and analyze this data, we utilized Microsoft Project. Our schedule was broken into a planning phase, spanning 25 days, and three iterations. The first iteration consisted of all the basic registry infrastructure, and was projected to take 12 days. The second iteration involved dynamic extension of the inputted data, and was also predicted to take 12 days. The third iteration consisted of the entire statistical analysis system, and was the longest, at 51 days.

The actual schedule ended up being very different. The first increment ended up taking a much greater amount of time than we had expected, primarily due to unfamiliarity with the technologies involved in development. Additionally, the requirements that the estimation was based on turned out to be far from the actual requirements of the sponsor. This discovery was not made until late in the process, when significant work had already been completed. More discussion of these problems can be found in the section titled “Project Reflection”.

Due to these unforeseen issues and our failure to properly plan for them, we only completed work that, with the change in requirements, can be compared to our first and second increments. We were forced to renegotiate the project scope with our sponsor, who luckily was very accommodating. Once this occurred, we were able to produce a finished product, albeit at the last minute.

System Design

Given our technologies, the basic application design structure was decided for us. At the absolute highest level, we have a client, a server, and a database. On the client end, we’re using HTML, CSS, and Javascript. HTML defines the page structure, CSS defines the styling of the page, and Javascript controls page interactions that don’t require a page refresh. On the server side, we’re using C# built on the ASP.NET MVC3 framework. Server side code interacts with the database to pull necessary information. It then takes that information, processes it, and embeds it into the HTML.

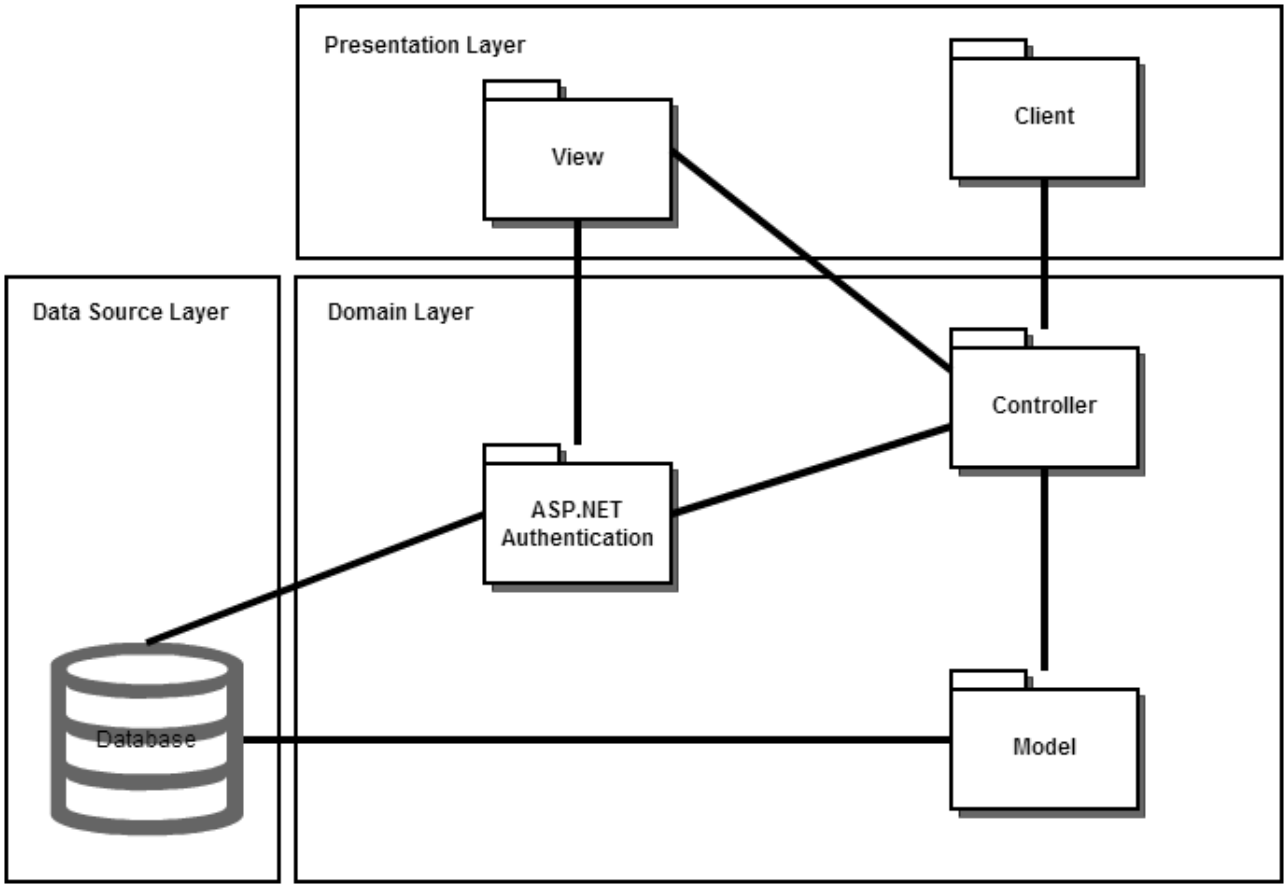


We chose the MVC3 framework for a number of reasons. First, the MVC framework provided a well-defined code structure. Since we were going to be handing this project off at the end of fall quarter, we needed a way to ensure that the code was clear, maintainable, and extensible. The MVC framework allowed for this. By separating the application into models, views, and controllers, it made separating the concerns a lot easier. Any alternative would potentially lead to sloppy, transaction script code.

The second reason we chose the ASP.NET MVC3 framework was for the sake of easing development. ASP.NET has a number of built in user functions that proved helpful in terms of development. Why build an authentication script when there's already one built in? By default, when designing an ASP.NET web application, ASP.NET gives code for user registration, login, logout, and password changes. Obviously, we had to tweak these to suit our design (both in terms of structural and visual design), but it was still a lot less work than it would be to code all of that from scratch. Additionally, ASP.NET had built-in functionality for handling user roles. With our administrator, medical provider, analyst user structure, having built-in role functionality was extremely useful and reduced the number of custom database queries made.

The final reason was for ease of deployment. ASP.NET MVC's Entity Framework allows Visual Studio to create the database from scratch. Using the connection string specified in the web configuration file and the model object structure, Visual Studio will populate the database tables for you. Additionally, any time a change is made to the model structure, the database will be updated to reflect that change in the model.

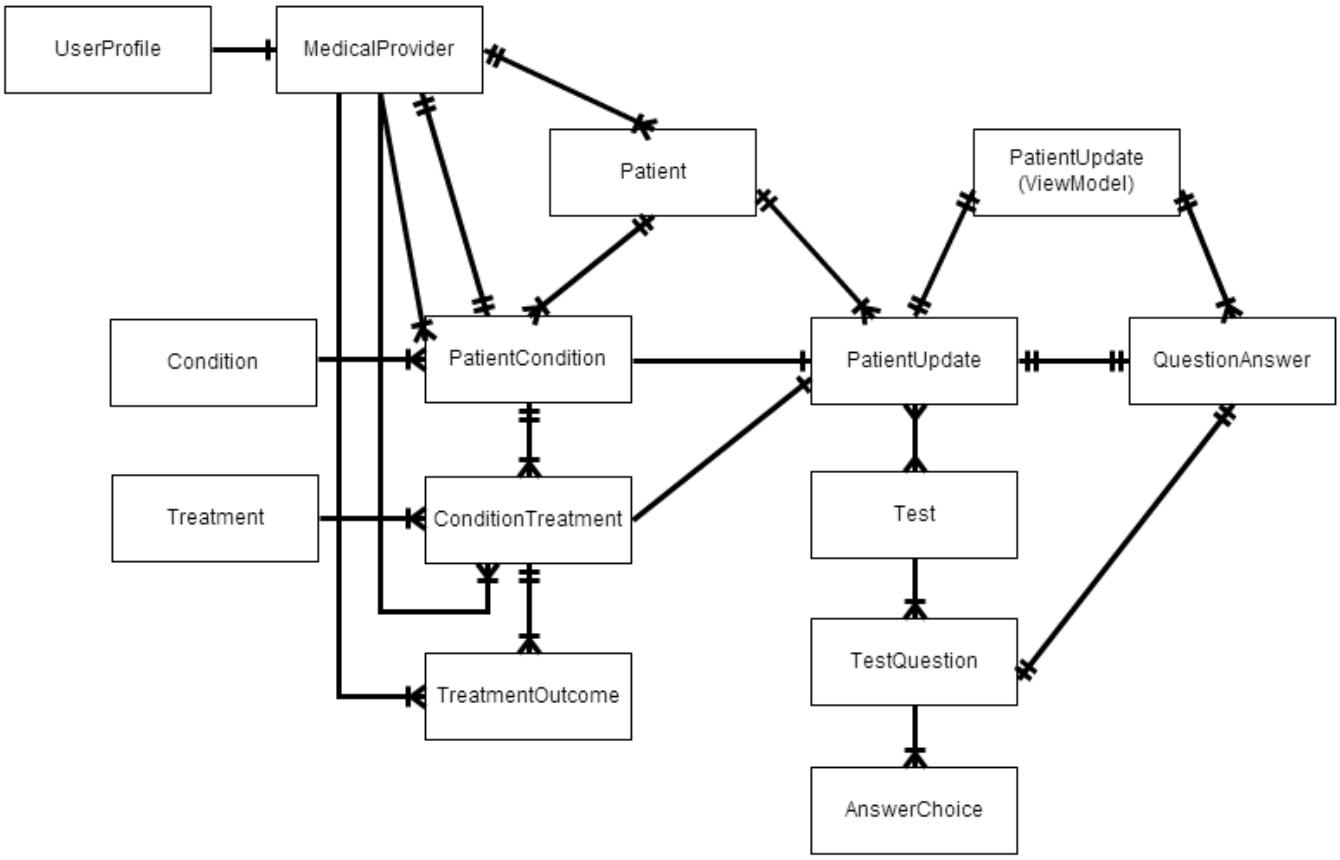
So, with ASP.NET MVC3, not only was our architecture decided for us at the very highest level, but the MVC design pattern was also selected for us. With this pattern, we essentially have three different layers. The first is the presentation layer, which handles how information is presented to the user. The second is the domain layer. This is where any domain logic or processing occur. The final layer is the data source layer. This layer is where persisted information is stored.



An HTTP request is made by the client. The server passes the request to the appropriate controller. The controller then decides which model it should be using. Once the model is requested, it accesses the necessary information from the database, which is passed back to the controller. The controller takes that model and passes it to the appropriate view. The view uses the model to inject information into an HTML template. The resulting HTML is returned to the client, where the HTML is rendered and the Javascript is executed. Additionally, the controller and view both have access to the ASP.NET user libraries, which pull user data from tables in the database constructed by Visual Studio.

Diving down to an even deeper level, we have our model object structure. This structure is directly related to the database thanks to MVC and Entity Framework. Our major entities are

UserProfile, MedicalProvider, Patient, Condition, PatientCondition, PatientUpdate, QuestionAnswer, Treatment, ConditionTreatment, Test, TreatmentOutcome, TestQuestion, and AnswerChoice.



At the very top is the UserProfile. A MedicalProvider can be connected with a User Profile. Alternatively, Analysts or Administrators can also be associated with UserProfiles, but because Analysts and Administrators don't currently store any unique data that the UserProfile doesn't, there isn't much of a point. On the other hand, MedicalProviders do store unique data regarding the Provider's facility and location. A Patient has a primary MedicalProvider. Patients also have PatientConditions and PatientUpdates. PatientConditions reflect a stored Condition in the system, put there by Administrators. Patient Conditions can have PatientUpdates and ConditionTreatments. ConditionTreatments reflect a stored Treatment, also put in the system by Administrators. ConditionTreatments can have PatientUpdates and TreatmentOutcomes associated with them.

PatientUpdates require a Patient to be associated with it. Optionally, it can be associated with PatientConditions or ConditionTreatments. These associations determine the update's position on the patient profile page. PatientUpdates can have any number of Tests associate with them. Like Conditions and Treatments, Tests are added to the system by Administrators. This

connection between Tests and PatientUpdates is a many-to-many connection. To accommodate this many-to-many relationship, an UpdateTest table is used, storing the PatientUpdateId and the TestId. PatientUpdates are packaged in a PatientUpdateViewModel with a list of QuestionAnswers. Each QuestionAnswers applies to a specific question in a specific PatientUpdate.

Process and Product Metrics

Our team was set to track hours spent in meetings, time spent on requirements vs estimated, hours spent per week, number of complete requirements, open defects, and lifetime of defects. Defect related metrics were not tracked due to a lack of definitive defect tracking and quality assurance. Hours spent in meetings became a somewhat less useful metric, as meeting times were generally very consistent, providing little insight into our success.

Time spent per week and on requirements faced a significant challenge. The metric tracking methods were not well streamlined in the process, and as a result became very inconsistently tracked. The team frequently failed to keep the document up to date, due to vague requirements recorded in the task list, a lack of streamlining of the process to ensure that metrics were easily and regularly recorded, and possibly other reasons. This may signify potential failings for our process or tracking methods, however provided little insight in the development of the project itself.

Product State at Time of Delivery

Initially, we broke the project up into two parts. The first part was the basic input functionality. The second part was the analytics portion. The input functionality would allow users to add things into the system, while the analytics portion would allow them to observe trends regarding the data stored. We planned to complete the input functionality by the end of summer quarter and the analytics functionality during the fall. However, due to a number of issues discussed in the next section, we fell behind during the summer quarter. We tried to catch back up in the fall, but after being so far behind and receiving a sudden, major requirements change, we lost all hope of completing the analysts portion in time.

The input potion of the application is complete (ignoring analytics input). Administrators have CRUD (Create, Read, Update, and Delete) operations for users, conditions, treatments, and tests. A medical provider is able to access their profile and make changes. They can also view a patient's profile. From a patient's profile, they can add conditions, treatments, treatment outcomes, and patient updates.

Administrators can currently add analysts to the system. Analysts are also able to log in, view, and edit their profile. Unfortunately, without any of the analytics functionality, that's about all they can do. There were also a number of user interface changes that we considered making that didn't get completed before the code freeze including the idea of replacing a number of the select boxes with autocompleting text fields. There was no extra functionality. We lacked the time to complete the original requirements, let alone extra, undiscussed functionality.

Project Reflection

As a learning experience, this project was an enormous success. Every member of our team has expressed the feeling that we have a much greater understanding of what working on an actual project will be like.

At the beginning of the project, we did a great deal of process and planning. As we had been taught in many SE classes before, we completed a formal requirements document, architecture document, and project plan. However, this is where it started to go wrong. In all of our prior experiences, we either had fairly complete requirements defined, or we were able to elicit them easily. In this case, however, the requirements were significantly more nuanced, and the initial document we received from the sponsor did not detail them enough. In this case, the burden falls upon us to meet with the sponsor and determine exactly what the software needs to do. Unfortunately, we were all fairly new to actual proper requirements elicitation, and we did not do a good enough job. What happened was we ran with some assumptions about how the system needed to operate, which caused some significant redesigns and time delays later on in the project.

Additional problems surfaced once we made the transition to coding. We quickly found out that our collective lack of experience with the technologies involved (ASP.net, Visual Studio, Git, C#) would be a greater problem than we had anticipated. We ended up having to take a great deal of additional time to familiarize ourselves and learn how to use the required tools. This negatively impacted our schedule yet again.

Throughout these problems, our sponsor was very understanding and helpful. We had to renegotiate the project scope more than once, and Nathan helped us find a compromise that hopefully works for both parties. Our team leaves this project with a number of new skills, and a greater understanding of the software engineering process, and we will be hopefully better equipped to deal with these problems when we encounter them again.