Fall 2012
initial: 11/25/2012
last modified: 12/17/2012

The State of Attendance Project

Fall 2012
initial: 11/25/2012
last modified: 12/17/2012

Table of Contents

Fall 2012
initial: 11/25/2012
last modified: 12/17/2012

Executive Summary

The goal of this document is to draw up a reasonably comprehensive report on the current state of the FTTA Attendance Project. As the Training becomes increasingly reliant on the Attendance Server for its day-to-day operations, it becomes equally increasingly urgent that we carefully assess the current state of the project, especially with a view to its longevity[1].

In this initial iteration, we first present a brief objective overview of the project as context. Our primary burden is in the remaining two sections, where we provide our assessment of the project's current state and then propose several measures to address our concerns.

The attendance server provides great benefits to the training and has continued to grow in its scope due to advantages it provides over analog systems. Ostensibly, this upward trend should continue reliably to serve even more functions (e.g. announcements and room reservations). However, in actuality this is unsustainable. Several factors, principally poor design and lack of management, have brought the project to a point where its long term maintenance is impractical and even unfeasible. These difficulties may not yet be readily manifest, but they will certainly become apparent over the course of the next five to ten years.

We have proposed many solutions, primarily related to the development process, to address the shortcomings that led to the current state of affairs. However, to resolve the issue of the project's longevity, our conclusion is that the best recourse is to rewrite the project from the ground up. In fact, we strongly believe that this course of action will need to taken eventually, and thus it is expedient that we take it sooner rather than later.

— Jonathan Tien,
on behalf of the AP brothers

[1] It is the opinion of the author(s) that such an endeavor ought to be undertaken every term (to varying degrees of thoroughness and scope, dependent on need) so as to communicate not only obstacles and needs, but also progress and plans to both development team and the training administrators.

Current Project State and Scope

Technology Stack[2]

At its core, the Attendance Project is a typical **Linux**, **Apache**, **MySQL**, and **PHP** stack, otherwise known as a LAMP stack, a dated web application stack that was very popular several years ago but has since faded from prevalence due to the limitations of the PHP language.
In addition to these, **HTML**, **CSS** and **JavaScript** are used to construct the front-end of the application.

The key libraries we employ are **JQuery**, a powerful and widely used JavaScript framework with a large library of plugins, and **AdoDB**, an abstraction layer that eases PHP's interaction with the MySQL database. **Dojo**, another JavaScript library is also used in some places.

The entire codebase is managed under **SVN**, one of most common version control systems in use today.

As stated earlier, the technologies employed are now considered out of date, although they are still serving us passably for the time being. Newer technologies[3] may serve us better on multiple levels, although a more detailed study is required in order to adequately consider the costs and tradeoffs involved.

Codebase

The codebase currently consists of **~15K lines** of PHP code (does not include HTML, JS, and other file types), not including library files, and **2715 total files** (all types), not including library files. A quick build yields **40 errors** and **21351 warnings**[4]. On the bug tracker there are currently **97 standing errors**.

The codebase, thankfully, works, and is relatively stable, but is, by and large, "spaghetti code," a term used to refer to messy code that is seemingly "thrown together" and happens to work.

[2] In software terms, a 'stack' is a set of technologies that are put together to serve as a base upon which an application is built. In the case of the LAMP stack, Linux is the server operating system, Apache is the server software, MySQL is the database, or 'backend' and PHP is the implementation language.
[3] For example, using Python+Django or Ruby+Rails instead of PHP, Redis/Mongo instead of MySQL, and Git instead of SVN *(see also: Proposed Solutions and Suggested Measures—Rewrite Proposal)*.
[4] Via *Eclipse* PHP compiler. Many of these are trivial, belonging to library code (not ours), but others reflect some basic errors in our code that should have been caught by a compiler and/or code review.

The majority of the codebase is, by industrial standards[5], poorly designed, poorly organized, and poorly documented in many places, thus making it difficult to read, and in turn, difficult to maintain, and difficult to modify. As a result, the extensibility and longevity of the project are greatly jeopardized, and will be so increasingly as the codebase grows in its current state.

Database

The MySQL database consists of 217 tables containing 2,743,487 rows.

The database is also poorly designed, containing redundant and poorly named tables that are not only confusing, but necessitate long and hard-to-read queries for even the most basic of data retrievals.

Feature Scope

The project has, over time, grown to include much more functionality than it was originally intended and designed for. While the utility and scope of the project is impressive, this abnormal and unchecked growth reflects a serious problem in the development process and needs to be addressed urgently.

Below is a list (non-comprehensive) of functionalities provided by the project:

- Absent Trainee Roster
- Administration
- Announcements
- Attendance
- AV Requests
- Bible Reading Tracker
- Boston Application
- Gospel trips
- Class syllabi, seating charts, exams
- Class notes
- Grad
- Greek vocabulary
- House/Maintenance Request Forms
- House Inspections
- Life-studies
- Room Reservations
- Team Statistics
- Trainee Rosters
- Service Scheduler
- Web access requests

[5] While it may be easy to dismiss 'industrial' standards by claiming the project's non-commercial, non-industrial nature, it is (ironically) more imperative that we follow industrial practices due to the nature of our development team *(see Proposed Solutions and Suggested Measures—Project and Process Management)*.

Project Diagnosis and Analysis

Design

The fundamental problem with the project is its poor design. It is clear from the architecture of the project that it was not designed to support as much functionality as it currently does. Most modern web applications follow the Model-View-Controller (MVC) design pattern, which cleanly separates the application's data (the models), interfaces (the views), and application logic (the controllers) into modules, making it much easier to modify and extend various aspects of the application.

Powerful MVC frameworks exist for many languages, including PHP, but the attendance project does not make use of one, instead consisting merely of a loose association of PHP files with no consistent structure or conventions. Such a lack of organization makes the code very difficult to read and, in turn, to modify and extend. Even seemingly simple tasks can become nightmares due to the lack of modularity in the codebase.

Furthermore, poor design decisions impacting large sections of the codebase were made early on, which affects developers down the line. One severe example is the fact that leave slips are represented not as objects but as arrays[6]. In fact, there are hardly any standard classes in the project. Even without the presence of an MVC framework, there is no reason complex objects such as leave slips are still represented as arrays. This makes it very difficult, if not impossible, to modify the structure of a leave slip and its associated functions.

Code

Aside from high-level design decisions that affect the codebase, there are many problems that plague the codebase throughout.

The first is the lack of documentation[7]. As a standard practice, programmers should always document their code for their own sake and for the sake of any other programmers that need to read their code. While some functions are documented, there are just as many that are not and even more long blocks that are void of any comments, which oftentimes makes it very laborious to decipher what certain sections of the code are doing. In addition, commit logs[8] are often vague, sparse, or nonexistent, so that when bugs are found, it is difficult to trace back why code

[6] 'Objects' are specific instances of template 'classes,' which are structured definitions of data entities in a software application. 'Arrays' are basic, unstructured, general-purpose data structures that can contain any kind of data.
[7] Comments that are inline with the code, explaining what sections of the code perform.
[8] Comments that are written for each batch of changes committed to the code repository.

was changed.

A lack of standards and conventions across the project also contributes to the difficulty in being able to understand the codebase. Different portions of the code use completely different methods of implementing the same functionality, such as establishing a connection to the database and making a query. Some portions of the code use techniques or libraries (often without necessity) that are not used anywhere else in the code (such as AJAX techniques, or the dojo library).

Additionally, there are many places where bad practices have gone unchecked. The most serious example is the fact that we do not sanitize any of our database inputs, which makes the server vulnerable to any kind of SQL-injection attack[9] and produces very basic parsing errors[10]. This could have been avoided by simply using a slightly different format for database queries.

Database

The database contains many unncessary tables. Tables such as `accountType`, `traineeStatus`, `leaveSlipCategory`, etc. that rarely change should be represented in program files as global constants instead of in the database to reduce the number of database queries.

Other tables are unnecessarily redundant and can be combined, such as the ones associated with leave slips, which are 10 in total (`leaveSlip`, `leaveSlipApproval`, `leaveSlipMember`, and `leaveSlipTA`, for example, could probably be combined). This would reduce the complexity of our database queries.

There are also tables that are no longer used and should be cleaned up or exported and deleted, such as `logClassLogin`, `tmpLsImportOffense`, and `trainee_old`.

Process

The attendance project is in a very unusual predicament because of its unique circumstances. We have very little choice in who is available to work on the project—a team of trainees with varying background, amounts of programming experience, knowledge of software engineering principles. Each engineer spends a maximum of two years on the project and then leaves,

[9] A type of security vulnerability where an attacker can 'inject' a database query to be run on our database via an input form. In our case, because we do not sanitize, or double-check, our inputs, any user could theoretically delete our entire database. On a more practical side, this just means that some formatting will throw errors—this has happened multiple times, especially with quotation marks in titles.
[10] For example, since SQL queries make use of quotation marks, inserting quotation marks into a query improperly (without using escape characters) can cause database errors. This appears often with inputs like "Children's team" or "The Secret of God's Organic Salvation: 'The Spirit with our spirit'"

leaving behind code that may or may not be documented, readable, and bug-tested for the next engineer to pick up and decipher.

Furthermore, we have no formal development process. We do not manage our talent effectively, we also have no long-term view and budget of how the development is proceeding as a whole. We do not write specifications, which means we have no technical record of what we are implementing and why it is being implemented. We do not enforce any kind of coding or documentation standards, which, as stated above, leads to unreadable and unmaintainable ‘spaghetti’ code. We also do not have any code review or testing, meaning we often do not know whether our code is sensical, whether it works as intended (which, because we have no specs, is hard to tell) until it breaks in production.

Practically speaking, this means that each developer is operating more or less independently from the others, making the entire project a messy collaborative effort.

The reason we emphasize a strict process as employed in an industrial situation (despite the fact that we are not an industrial entity) is that these software engineering principles (techniques and processes developed over years of corporate software development) avoid the pitfalls that are encountered by any team of software engineers attempting to build a product. In industry, these principles are often not honored and the practices put forth not adhered to strictly. This is primarily because of the competitive nature of commercial software and the necessity of moving quickly and even breaking things[11].

However, our situation is quite different, as we are not a commercial entity, and we do not face any kind of competitive pressures. In addition, due to the short tenure of our developers, we require the superintendence of the guidelines in order to ensure a clean transfer of responsibility to the next crop of developers . As a result, we have every reason to and absolutely no excuse not to adhere to the standards that produce well-functioning software development teams. In doing so, we will not only produce high-quality software, to the benefit of all associated with the training, but also ensure that said software has longevity.

[11] Indeed, this is the now popular development principle, set forth by “hot” new startups such as Facebook, etc. that emphasize fast iteration to capitalize on a first-mover advantage and remain competitive and innovative in the face of ever-growing competition. There is no need for us to follow such a trend. If the modern way is to “move fast and break things,” then our way should be to “move slow and stay stable.”

Proposed Solutions and Suggested Measures

This section outlines many steps we can take to alleviate and address the problems noted in the previous section. Project and process management is the most important issue, as poor management is at the root of many of our problems—our current program is much too loose and unstructured. Developing and enforcing better developer practices will also be very beneficial (which is, of course, a matter of management). Certain developer tools can also empower developers to avoid pitfalls. The addition of a quality assurance team would also be a boon to the project, if it were possible. Lastly, our conclusion is that the entire project should be rewritten from the ground up, instead of continuing development of the existing codebase. The best way to implement the suggested measures is to start afresh and rebuild the project, applying these lessons.

Project and Process Management

Technical Oversight

The first need for a more solid technical oversight—trainees with a good understanding of basic software engineering principles and know how to manage a team of developers to build a project cohesively. This is generally a product/project manager.

Scope and Roadmaps

The function of this role would be to first evaluate the current and intended scope of the project and then establish priorities (through fellowship with TAs and other administrators). From there, a roadmap for the development process can be laid, and all feature development can be designed and carried out with a roadmap as a guide. This will ensure the project's long-term viability, as code written will be able to account for future development and be extensible enough even for unanticipated development.

Spec and Design

For each feature being written, even relatively minor changes, a specification (spec) and design document (however short) should be drawn up to document what is being changed, why it is being changed, and what the end result should look like.

Code Review and Testing

Using the spec, tests can be written and run to ensure that new functionality does indeed work as intended and, furthermore, does not break existing functionality. Code review can also be applied to ensure that developers' code is well-documented and readable for future developers' sake. Testing methodologies are discussed further in a later section.

Bug and Issue Tracker

The bug tracker in its current state is helpful, but does not come up to the standard. In actuality, it serves as an 'issue' tracker, as it is being used not only for bug reports, but also for SQL errors (which are not always necessarily bugs), and feature requests. There needs to be a way to differentiate between all of these, a way to open and close issues easily, a more accurate priority scale, an easier way to track which issues are assigned to which developers, and a better way for developers to report their progress on their tasks. Therefore, a new bug tracker should be written to help facilitate a more structured management process.

Interview Process and New Developer Onboarding

All developers have varying degrees of experience and expertise, and need to be applied in a way to maximize their value and minimize the damage they may do. A more thorough interview and onboarding process should be implemented so that overseers have more time to assess how to best use the new developers and also give the new developers time to familiarize themselves with the codebase and development practices.

Furthermore, the current onboarding assignment is insubstantial and should be expanded to evaluate the new developers more thoroughly. In addition, a few assignments should be given on a provisional basis with peer code review done afterwards with a view to mentorship.

Developer Practices

Coding Standards

Our Developer Guidelines page is outdated. This page should be updated at the start of every term to account for lessons learned over the course of the previous term. In addition, it should be reviewed with the developers at the start of every term to reinforce the guidelines therein.

Documentation

In addition, developers should specifically endeavor to document their code thoroughly so as to ease the reading of their code by future developers. Code should not be pushed into production without documentation meeting certain standards

Commit Logs

In the same vein, developers should write detailed commit logs so that every change is well-documented and can be traced back to a specific reason.

Developer Tools

Dev Server

A development or "dev" server should be provisioned as a testbed for new features or large changes that require more thorough testing. Not all changes can be tested locally, and this offers a good opportunity for other developers to attempt to break others' code, as any one developer often cannot anticipate all the edge cases that might break functionality. This would ensure that our features are tested and reviewed before being pushed out to the entire training.

Quality Assurance

Dedicated Team

If a dedicated group can be formed to take care solely of testing, it can be done effectively. We can either dedicate a certain number of brothers on the project to the testing effort, or alternatively, we can form a team of sisters with programming experience as a QA team.

Code Review

The QA team would be responsible for reviewing code before it is pushed into production to ensure it meets the standards set up beforehand in terms of readability, memory management, documentation, structure, etc.

Unit, Regression, and Integration Testing[12]

The QA team would also be responsible for writing various kinds of tests, which would be developed based off the of the specs written for each feature. Unit testing is the most crucial, followed by regression testing, followed by regression testing.

Refactoring[13]

A QA team could also pick up the task of refactoring existing code. This is would be a tremendous effort and may not be worth the investment. However, if the application is not rewritten, this will eventually become necessary for certain portions of the codebase.

Rewrite Proposal

Our best recourse is simply to start afresh, applying all the lessons we have learned thus far. It will still be possible to get by for some time, but eventually, continuing with the current project will be an untenable position. Ultimately, as the codebase becomes increasingly bloated, the project will have no other choice but to take this route, and as with the case with such

[12] Unit testing is a methodology for testing 'units' of source code, which can be thought of as the smallest testable part of an application. Regression testing seeks to uncover new bugs, or 'regressions' in functional code that has been modified. Integration testing involves combining multiple modules together to test if they work in aggregate.

[13] Code refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior, undertaken in order to improve some of the nonfunctional attributes of the software.

undertakings, sooner is always better than later. The longer this is postponed, the more painful the transition and rewrite effort will be.

Below is a rough proposal for a possible rewrite of the attendance project that we hope will spur more consideration in this direction.

Design[14] and Planning

A rewrite would require a carefully planned redesign. There would need to be an audit of all the current functionality provided by the attendance project and a thorough evaluation of how these functionalities are connected to each other and the database. Then we can begin to prioritize which functions to reproduce in the new version and how to ease the transition. Some administrative functions, for example, may have to remain live on the old server before they are ported over to the new server.

Next, future functionality should be considered and a development roadmap drawn up to guide the design effort. This entire process would necessitate much fellowship with the TAs and other serving ones.

Design can then begin in earnest, following the principle of MVC, separating data models from interface views from logic controllers. Data models should be designed to represent various entities the server will need to handle (users, trainees, teams, schedules, attendance records, leave slips, etc.) with a view to being extensible. Interfaces should be sketched, modularized, and then defined programmatically as templates. Controllers can then be spec'ed and designed, coded up according to spec and tested according to spec.

The entire development process should follow the measures suggested in the previous section. A rewrite would be a long-term effort and should be undertaken with caution and a measured pace so as to produce a high-quality end product. Engineers should be assigned specific responsibilities according to their abilities (some to design data models, others to sketch and test interface designs, etc). Deadlines and milestones should be set. The entire process should be thoroughly documented for the sake of future developers. The codebase should be tested before being pushed into production.

Technologies[15]

Language + Framework

We should choose a modern MVC web framework on which to rebuild the project. One very promising option would be to adopt the Django framework for the Python language. Python is

[14] By 'design' we are referring to software architecture design, not visual design or interface design.
[15]This section is intended for an audience with a technical background in computer science. It is intended to serve as an initial survey of what modern technologies could be adopted. Seek clarification and/or further explanation with the author(s) if needed.

a very powerful modern programming language that emphasizes code readability and is easy to learn. Django is the most popular Python web framework and is also extremely powerful. It adheres to the don't-repeat-yourself (DRY) principle and features an object-relational mapper that allows you to define data models entirely in Python and then generates a programmatic API for accessing database data (eliminating the need to write SQL queries in many cases), automatic admin interfaces for adding and modifying database data, a flexible templating system, and some other useful features. It is used widely by prominent websites such as Mozilla, PolitiFact, and Pinterest.

Other options include Ruby on Rails or Sinatra, NodeJS, and CodeIgnitier.

Database

In selecting a database, we can consider moving to a NoSQL solution (also known as a key-value store) that store data simply in key-value pairs rather than complex tables. This is often faster, does not bloat as easily, and easier to handle programmatically (not complex SQL statements). However, Django does not officially support any NoSQL databases and its object-relational mapper already alleviates many pains normally associated with maintaining a large SQL database. In addition, MySQL has the advantage of having a very nice front-end admin interface with phpMyAdmin. Examples of NoSQL database technologies include Redis, Cassandra, and MongoDB.

Front-end

We should strongly consider using a web UI framework such as Bootstrap or Zurb Foundation. These are powerful and extensive libraries that not only include scaffolding but also many common web interface elements that can be customized and reused with ease, such as dropdowns, tooltips, buttons, labels, etc. This will provide consistency throughout the UI and also speed up UI development.

Version Control

Our current version control system is SVN, which is undoubtedly the most widely-used VCS. However, Git is now considered by many to be the superior technology, although it is has a slightly greater learning curve associated with it. One advantage of Git is that it creates a local repo of the codebase on each user’s system, and it therefore allows developers to commit changes to a local repo before pushing those changes (whether one by one or in bulk) to the central repo. Since we can only access the repo in the TC, this would allow devs to commit changes even while not in the TC, and encourages finer granularity in commits (which in turn makes it easier to roll back changes and identify bugs). In addition, Git’s merging algorithm is generally considered superior to SVN’s (which reduces the need for manual merging).

In addition, it should be noted that because of an issue with our server certificate, we cannot connect to our SVN repo via command line SVN, and a number of other SVN clients. This is unacceptable.

Development Environment

We also need to consider how any change in technologies will affect the developers and their development environments. As much as possible we should ensure the developers will have a variety of ways to establish their own environment and workflow, and that testing can be done locally. Python, Ruby, and PHP run fine on all major operating systems, as does MySQL. NodeJS and Redis, however, do not. There are a variety of IDEs that can be used without problem (Netbeans, Eclipse, XCode). Git has a variety of (free) clients for each major OS as well.