DEVELOPMENT OF A MULTI TENANT WEB APPLICATION INTO A SOFTWARE PRODUCT LINE FOR TRANSIT SYSTEM SCHEDULE MANAGEMENT

Michael Carey, Derek Riley
Computer Science
University of Wisconsin - Parkside
Kenosha, Wisconsin 53144
carey012@rangers.uwp.edu
rileyd@uwp.edu

Abstract

Developing large, complex web applications that are easily re-deployable for new clients is important to decrease the overall cost of re-deploying software. Further, easy deployment increases maintainability of a complex system, so updates and additions to the original code are often less difficult to manage if the code is easy to re-deploy. Since maintenance is often the most costly phase of software development, it is important to increase maintainability of code to reduce long-term cost and increase the longevity of software. Server-side application deployment can be especially difficult and tedious because often many configurations and adaptations are required to customize a software deployment for a new client. Software Product Line (SPL) Engineering can provide a valuable solution to managing software complexity through sharing common characteristics. This allows for rapid deployment for new clients, more straightforward updates, reduced memory utilization, and straightforward backup increasing maintainability. In this paper, we present development of a SPL for a server-side application to manage a transit system database for use with a mobile application to help transit riders find out when busses are arriving at stops. The server-side system we present is a web application that allows users of various privileges to access, modify, and push updates of transit data to the mobile applications to increase transit user knowledge. The server-side application contains many complex, interrelated components and includes an automated deployment system developed for new clients through the use of hierarchical user privileges, and database manipulation. Our application utilizes open source software, builds upon the foundation of privilege restriction, and adopts SPL patterns emphasizing the need for a customizable, manageable product that can easily add new clients.

1. INTRODUCTION

Developing large, complex web applications that are easily re-deployable has significant potential to reduce the cost of re-deploying software, therefore increasing reuse and maintainability. Increased maintainability allows for simpler updates and additions to the original code especially if the code is easy to re-deploy. Since maintenance is often the most costly phase of software development, it is important to increase maintainability of code to reduce long-term cost and increase the longevity of software.

Server-side application deployment can be especially difficult and tedious because often many configurations and adaptations are required to customize a software deployment for a new client. This challenge is further exacerbated in mobile and web applications where multiple platforms must be supported with frequent updates for new standards, libraries, and operating systems. Software Product Line (SPL) Engineering can provide a valuable solution to managing software complexity through sharing common characteristics.

Multi-tenancy refers to the ability to use the same software or interface among many users, thus isolating data and traffic [4]. Multi tenant applications share the same codebase, thus providing reliable use of server resources while also allowing ease of updates cascading onto all tenants. Since tenants also share a common database schema, it is possible to limit database usage as well. This use of shared resources allows a single deployment of the system to serve numerous users over several applications while also permitting the ability to add other sub-applications with ease.

In this paper, we present development of a SPL for a server-side application to manage a transit system database for use with a mobile application to help transit riders find out when busses are arriving at stops. The server-side system we present is a web application that allows users of various privileges to access, modify, and push updates of transit data to the mobile applications to increase transit user knowledge. The server-side application contains many complex, interrelated components and includes an automated deployment system developed for new clients through the use of hierarchical user privileges, and database manipulation. Our application utilizes open source software, builds upon the foundation of privilege restriction, and adopts SPL patterns emphasizing the need for a customizable, manageable product that can easily add new clients.

2. RELATED WORKS

SPLs have been developed for industrial mobile applications previously. A product line architecture was developed for a role-playing game on an early smartphone that allowed developers to improve performance and development speed through the incorporation of SPL development methods [8]. Effective planning for SPL development requires familiarity with relevant SPL methods and the core product being developed.

The reuse of software is an important SPL topic since the reuse of software is much more than just reusing the code [8,6]. Even just the structure of an SPL may be of use in another application. Though mobile applications tend to get the most attention to resource restrictive,

server applications can also be of restricted resources if large applications utilize space. Server applications usually incorporate more intense backup which may result in error if applications take up too many resources.

Numerous software programs aid in the development of site maintenance while permitting other users to access resources separating the administrator from authorized users differentiating from the anonymous user. Particularly, Content Management Systems (CMS), such as Wordpress or Drupal, have built-in functionality for management of users. Though this type of system works well in some application environments, many applications require a higher control over user abilities and maintaining administrations. Atop the usual user hierarchy, many systems also desire a super-user profile who can manage all users underneath, cascading privileges down the hierarchy tree.

SPL development requires planning and has been analyzed from different perspectives including: asset developer [3], requirements engineer, and product developer [2]. However, the application of method engineering SPL approaches have encompassed a more comprehensive view of the software to improve consistency and alignment with business goals for the software [2].

3. DOMAIN AND CHALLENGES

Bus systems transport over 50% of the public transportation audience and the number of people using these facilities has been increasing drastically [1]. A majority of these systems are funded by municipal governments funded by local government support, federal grants, and fares [7]. Much of the core features are common or standardized among bus systems, therefore benefitting from SPL focus. Identification of the common characteristics creates the potential to automate software deployments easing the workload that is put on the transit system employees and improving the information available to riders.

Transit systems rely heavily on paper schedules to relay route information to pedestrians. The predetermined routes are printed in brochure or poster format and cost the transit system large expenses to reprint and redistribute when changes happen. Digital technology can decrease the costs to change the schedule and with mobile technology allow real-time changes, such as delays, to be addressed to users.

Transit systems tend to maintain their own website where schedules and transit information can be obtained. Though this is a valid solution to portray information, systems without the resources may have out-of-date methods for portraying this information, such as hard-coded schedules that require manual changes and static cascading style sheets that prevent smaller screens from reading schedules. Using a website as a means of collecting transit information may be troublesome for mobile users if the schedule information is not within the “Three Tap” rule - any content should be accessible within three taps of menus/links - causing distress if the user does not know where to look. This results in numerous amounts of phone calls to the transit system every day from riders desiring time or routing information.

Some transit systems implement mobile applications or texting services that can aide the dispatchers’ valuable time. *Google Transit* is a service provided by Google that provides users

with information on how to get from one destination to another via public transit, walking, and bike paths. Of course, this material must be provided to Google via General Transit Feed Specification (GTFS) [5].

An existing mobile application and web management tool for managing transit information was developed as part of a class at UW-Parkside. This solution provides the means of editing route information and exporting to GTFS that can be sent to Google to update Google Transit. This implementation has been built with the intentions of running one tenant with multiple users. The manager is used as a platform for updating the mobile application that feeds information to users about transit schedules. More deployments have been implemented since the original software was written, necessitating the need to improve the deployment process when the software is redeployed for new transit systems [7].

The original application benefits further from a multi tenant solution because of the automated deployment incorporated into the manager system. When a new mobile application is to be deployed, a server deployment can be setup with minimal effort. This leaves less time required for actually deploying a new tenant and thus more time for adding features to the application.

4. MULTI TENANT SOLUTION

A multi tenant application must have a structured way of managing users and ownership of data within the application. Without such, any user may perform restricted processes such as deploying a new tenant. Similar to the UNIX user system, we implement multi-tenancy privileges in a hierarchy where users are associated in groups limiting their access to data and functions.

At the top of the hierarchical privileges is the super user – in this case, the owner of the manager system. The super user has the top ruling over the system allowing administrators within tenants who can then administer their own environment. Through this distribution of privileges, the super user is relieved of maintaining every environment but rather the system as a whole and creating new environments while leaving the administrators to their own realms. That is, the super user is responsible for building new deployments whereas the tenants themselves are responsible for only their deployment. Administrators of a deployment may create other users equal to or beneath their privileges and are responsible for editing information on their own segment of the system. The super user should only need to modify a deployment if a problem were to arise, such as passwords being forgotten, again much like the UNIX user system.

Placing the super user at the top of the privilege hierarchy gives the deployer the highest functionality on the site while restricting the administrators to their domain for their agency. These elevated privileges allow the deployer to add another agency easily without redeployment of code by simply adding configurations for an agency. With administrator privileges, each agency may manage other administrators, authorized users, and basic users within their own agency allowing each agency to maintain their own realm and the super user to maintain the system as a whole. With each new deployment comes a new database both for restricting privileges distributed over agencies and for easier backup. Hierarchy of database user privileges

and database tools further restricts agencies to their own realm. Separate databases allow the application to protect their own assets and avoid accidental overwrites of another agency's data.

4.1 Product Architecture

Our SPL design uses an already existing OSS framework revolved around Object-Oriented Programming (OOP) allowing us to extend core functionality, thus permitting the manipulation of an existing application with minor modifications to the code. This method of indirect manipulation of core functionality allows us to continue with a modular design and opens opportunities for adding tenants to pre-existing applications with little code modifications. We also can build a base incorporating multi tenancy for future applications, which can then be built off this architecture. By developing our solution modularly, we can simply place the code we have developed into the packages and enable them properly.

Rather than modifying framework core objects directly, we instead extend these core objects to build upon our own functionality that can be used as core. This allows us to overwrite - changing the function's definition - without having to change how the framework calls upon items. This allows us to place the code in a pre-existing application without changing every call made to framework defined functions. Moving to a newer version of the framework results in simply replacing current framework code with the newest version with minimal errors from our current applications configurations.

This case study uses a Model-View-Controller (MVC) pattern for development. This would then include a way to distinguish models from views and controllers. In an OOP MVC, a base object would be extended from to express the differentiation. We call the base controller object Base_Controller, and any controller built will then inherit from this object. Instead of having the custom controller inherit from the Base_Controller, a custom controller, Common_Controller, extends from Base_Controller and thus every model used now inherits from Common_Controller. This allows the functionality of the Base_Controller to be extended into even more common functionality related to the application. The Common_Controller can then autoload the User_Model, which contains the functionality used for logging in, creating new users, etc. This allows any controller to use the User_Model functionality when desired. Figure 1 demonstrates the flow of hierarchy used by the Common_Controller.

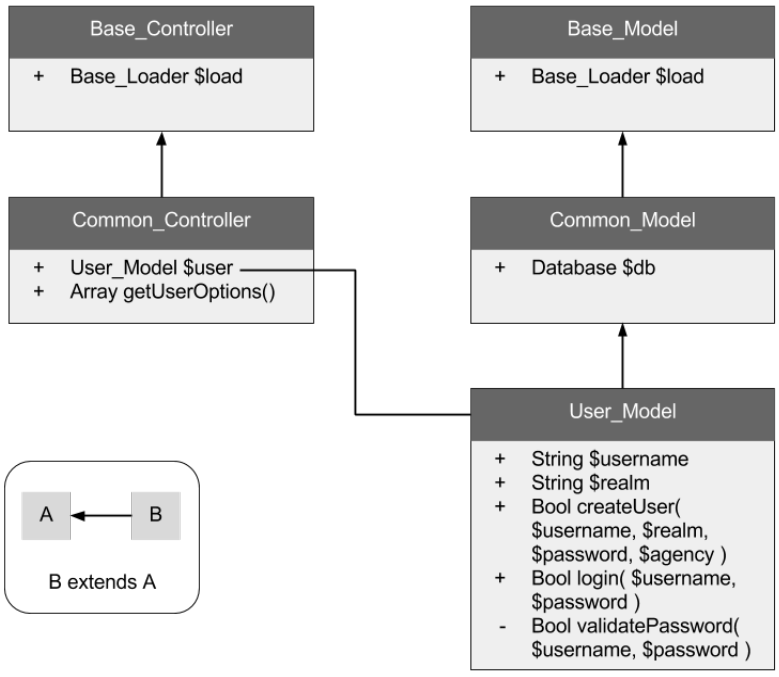

Figure 1: Common_Controller UML Diagram

4.2 File System

Frameworks usually consist of a file directory structure that must be maintained for use with the application. When multi tenancy is incorporated into this structure, we take into consideration the resources that should be kept consistent among tenants and which resources should vary by tenant. With this in mind, we also manage the file system differently by requiring a dynamically built structure.

In our implementation, a tenant is given their own base directory that can hold individual resources and data. We use the concept of skeleton directories to initialize a new tenant's directory while also using the idea of templating to configure these files accordingly. As well as influencing consistency of the file structure, the production of a tenant's file structure is also very flexible. We have the ability to manage this file structure differently according to system-wide configurations, which dynamically decide where files fit.

Taking principles from the UNIX user system, we set environmental variables to aid in tracking which user is currently being used but still maintain a layer of security by confirming this data within the database. To track which tenant is being used, we identify a unique identifier that is known by all PHP scripts that can be used in requests for files or URIs. Since uniqueness is the key component in this configuration, it would seem the best solution would be to identify by the tenants directory, which by the rules of the file system, must be unique. Without this session management, users may simply change their URL to enter a different tenant's environment causing a authorization bypass.

4.3 Database Structure

Database resources need to be highly secured sectors of web development since databases are some of the only efficient means of storing "permanent" information. Within these important resources include confidential information on users such as their passwords. Therefore, we must consider the security implications of sharing database information. Every database program manages access in different ways but to conform different database schemas, it is best to maintain for a generic database format.

The database usage is easiest maintained by creating a user for each tenant and restricting user privileges in the database. This ensures that if a user were to find a way out of their own realm to run their queries against the database, they are still restricted so they cannot get user credentials for example.

Due to configuration issues, particularly the problem when the application owner does not have proper permissions to allocate certain details, the database structure has two potential forms. The implementations involve running a single database restricting users access based on stored procedures or running multiple databases sharing a common database which tracks users and other databases.

4.3.1 Single Database Using Stored Procedures

Typically an application receives a database and a user to access this database. The desired database management would include one database with base tables shared amongst users defined by one key. By maintaining one database and shared base tables for all users, the database can be managed as a whole, cascading updates to the database schema throughout. Then, to restrict users access to their own realm of the database, stored procedures would perform the necessary insertion, update, deletion, and selection of database columns.

Though this implementation is desirable, it requires the higher, administrative permissions to grant users' permissions on specific stored procedures. This database variation allows easy backup of the entire system but requires a more sophisticated backup if each agency desires a separate database backup. This variation also limits the amount of database files on the system; however, permissions become scattered and harder to replicate.

This method of implementation seems fitting for a server maintaining multiple applications at once where manual operations are typically run by administrators. Granting permissions to specific users to specific stored procedures aides in constraining the abilities of a database attack since raw SQL cannot be injected into a stored procedure. This implementation leads to more restricted permissions scattered amongst many application database users, but leads to highly constrained application users.

4.3.2 Multiple Databases Sharing a Common Database

Another database option is running a single shared database with multiple databases cross-referencing the shared database. This option requires more attention for creation and updates. Besides requiring a new user to be created prior, a database must also be created with all needed

permissions granted to the user. This approach helps limit the amount of grant statements run on a database while also restricting the user information to a separate database and user. With a fresh database, tables can be created with a schema provided matching the desired format. Triggers may be maintained on these tables to permit default values or rather force values on a table. By using database triggers, we can eliminate the need for pesky cross-database foreign keys by enforcing the desired key onto the database instead. This approach allows the easy backup of individual tenants but harder for the system as whole.

Though this approach restricts the user to a particular table, it can lead to numerous tables running identical schemas. This approach emphasizes the update process, forcing all databases to run the same update process on them. This work would include generalizing the database updates and be forced to run on all databases likely by one user that would push to all. Thus suggesting a single database user with the ability to run commands on all databases. A catalog of databases would need to be maintained to push the updates on all versions.

4.4 Shared Resources

Some resources such as Javascripts and Cascading Style Sheets (CSS) must be made public to allow the Web browser to render appropriately. These resources are typically kept in a different area to help differentiate between the public and private. Due to the hierarchy of the framework used, a directory on the project root serves as the place for shared resources. This resource pool originally included resources such as uploaded files and exported files. These resources were to be requestable by mobile applications specific to the tenant. These resources were more suited for the individual directories created for the tenants.

To accommodate the drastic change in file structure, a resource helper was incorporated to aide in the management of resources. This resource helper takes a configuration that dynamically builds functions thus allowing resource path and urls to be determined at runtime rather than static allocation. This allows the file structure to change dynamically without causing the application to crash. This dynamic allocation of resources allows the application to also distinguish which resources will be consistent amongst tenants thus providing individual resources to also be included with this functionality.

5. EASE OF DEPLOYMENT

Deployment of an application can be a particularly painful process: loading proper configurations, manipulating files and directories, and testing for consistency and stability. This process can be tedious and troublesome when re-deploying the same application with different configurations. Ideally, a deployment involves placing a codebase and it just runs; however, this is typically not the case. The deployment process is best left to automation which can improve efficiency and spare an administrator large sums of time attempting to deploy by hand. Of course even with automation the administrator should assure the deployment was successful and may need to fix some configuration problems by hand.

5.1 Initial Deployment

The application we use as a case study in this paper, had an existing codebase that first ran static variables that were converted to configuration files on the second tenant deployment. Configurations made the application easier to deploy fresh installs of the application on a server; however, the installation involved re-deployment of all code with minor changes to configuration files. When two tenants were to run on the same server, the code is side-by-side without sharing any resources making further deployments even larger storage wastes.

When an update needs to be made to the code, an update has to be made on each deployment of the application essentially by re-deploying the code and tossing in the desired configuration files. This exhaustive method requires each deployment to be placed, tested, and fixed until working. The same changes made on the first deployment must be made on the next and so on.

When a new deployment is to be placed, certain files must be placed in the correct location for the application to work correctly. Special directories are to be built upon deployment for placement of resources. If these directories are missing, the server will not allocate the resources and logs would fill until properly fixed.

5.2 Automated Deployments

To ease the deployment process an automated deployment system was created to mimic the manual deployment process. Not only does automating the process significantly reduce the amount of time spent to complete, but also gives the control of creating tenants over to a user who may be unfamiliar with the manual deployment process. The automation process allows a user to identify the core assets that need to be used in the process for the configurations and the program takes care of the rest. This process will build files and tables consistently with little error and no experience required.

Figure 2 displays the form that is displayed only to the super user when attempting to deploy a new tenant. The form uses this information to build the foundation of a new deployment: building a database for the tenant, adding the tenant to the list of tenants, and building a directory and structure for the tenant. Figure 3 demonstrates the tenant being added to the list of deployments.

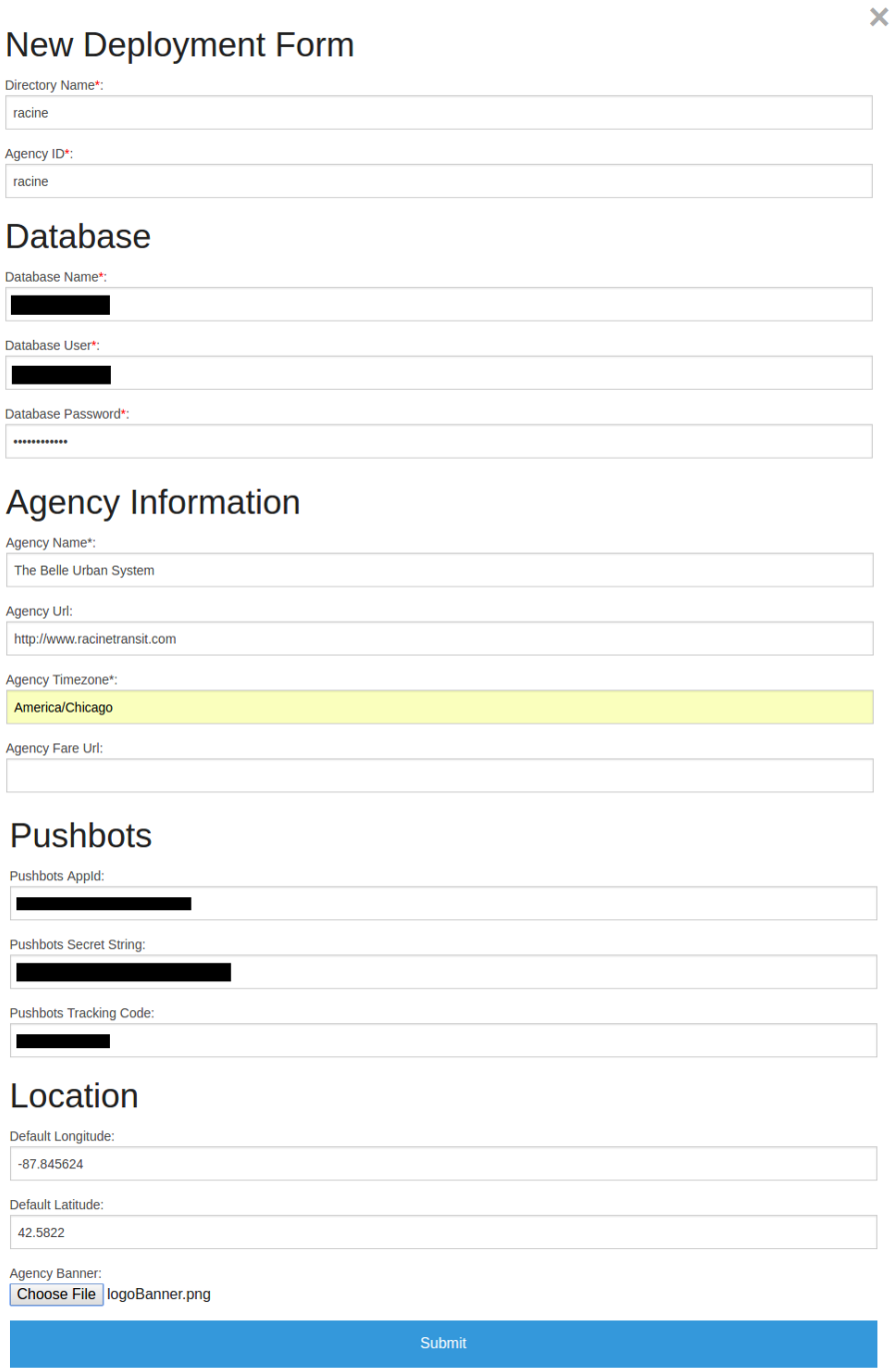

Figure 2: Automated Deployment Form

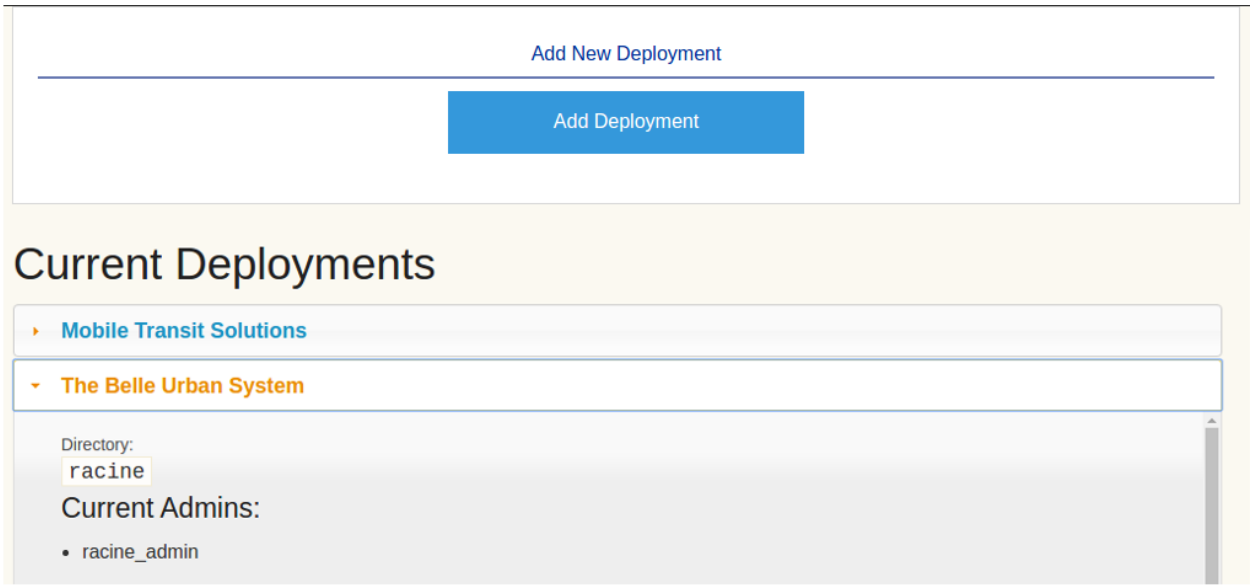

Figure 3: Automated Deployment Complete

The automated deployment system was built with the ability for configurations. The build of a tenant takes configuration files to identify how it should build accordingly. The three major portions of the configuration involve defining what fields to collect and what fields are to be required, what and how the database should build given some parameters, and what the file structure should look like for a tenant's directory. In fact, these three simple steps can be used to build a variety of application suites that desire tenancy. The process itself could possibly be integrated in another environment requiring similar circumstances.

7. LESSONS LEARNED

Security was considered one the highest priorities for this project. Though the data was not confidential or sensitive, it was desirable to prevent tenants from accidentally causing other tenants issues such as overwriting their data. It was determined the best way to prevent a misfit user was to treat them like an attacker who attempts to break outside their own realm. Some information security tactics taken include least privileges, session tracking, and segregation of duties.

At the lowest end of the system for this application is the database and operating system. The database had to be restricted so that users could only access their own data. It was decided each tenant was to receive a single database user and restrict their access accordingly. If a single database is to be used, stored procedures would constrain what SQL commands are run, preventing SQL injection and dynamically-built queries.

If multiple databases are to be used, stored procedures would be a nice addition to the restrictions, but at least if granting access on only one database, problems are contained within a single database. By restricting how a database can be used, it prevents poorly written code and missed bugs from allowing a user undesired access. Users would also be restricted on their access to the operating system though this is less restrictive on the programmer. A separation of

private and public resources were designated and htaccess files were written to prevent certain requests from being executed.

Within the application itself, privileges had to be determined to avoid users from running processes outside their abilities such as deploying a new tenant. User management was already implemented in the given application; however, it seemed to be constructed in a rush. Cookies maintained easily modifiable fields that determined user and realm of which neither were logged or stored in the database. Session tracking has been implemented to prevent a user from modifying their cookies and all logins have been logged to prevent session hijacking.

Through the implementation of security features and restrictions, one can see how difficult it can actually be to prevent misfit users and attackers from running rampant while also not restricting legitimate users that need tasks to be completed. One may also notice through the implementation of multi tenancy, how the necessity for ease of use can shift quickly on programmers for the necessity of security and configurability. Multi tenancy is better built upon the foundation rather than atop a pre-existing application. The development of multi tenancy from the start allows better-implemented security features, configurations, and software design.

8. CONCLUSION

The application of multi tenancy can provide better use of resources and provide easier means of updates to software deployers. Multi tenancy can include common characteristics provided to individual tenants that allow the efficient use of configuration and schematic updates throughout. This efficiency can lead to automation of deployments and limit the amount of errors that occur in a deployment. Limiting the amount of time that is spent on deployment and updates leaves more time for improving code and adding features.

We believe this SPL design could be further adapted to provide multi tenancy onto other applications. The already configurable design could be further generalized to work with any application with minimal changes. A modular version of this design may prove useful for designing new applications that require these capabilities.

CITATIONS

[1] American Public Transit Association 2014. 2014 Public Transportation Fact Book. Washington D.C.

[2] Chastek, G., Donohoe, P., McGregor, J. D., & Muthig, D. (2011, August). Engineering a production method for a software product line. In Software Product Line Conference (SPLC), 2011 15th International (pp. 277-286). IEEE.

[3] Clements, P., & Northrop, L. (2002). Software product lines: practices and patterns.

[4] Definition of Multitenancy - Apprenda. (n.d.). Retrieved February 17, 2016, from https://apprenda.com/library/glossary/definition-multitenant/

[5] General Transit Feed Specification Reference. (n.d.). Retrieved March 21, 2016, from https://developers.google.com/transit/gtfs/reference

[6] Martini, A., Pareto, L., & Bosch, J. (2012). Enablers and inhibitors for speed with reuse. Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1.

[7] Riley, D., Zygowicz, M., & Carey, M. (n.d.). A Case Study in Developing an Agile Software Product Line for a Mobile Application. Submitted to FSE 2016.

[8] Zhang, W., & Jarzabek, S. (2005). Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices. Software Product Lines Lecture Notes in Computer Science, 57-69.