2016

Mobile Test Viewer: Web Application for Interactive Exploration of Product Test Plans

Jason K. Vernon
Grand Valley State University

Mobile Test Viewer: Web Application for Interactive Exploration of Product Test Plans

By
Jason K. Vernon
April, 2016

Mobile Test Viewer: Web Application for Interactive Exploration of Product Test Plans

By
Jason K. Vernon

A project submitted in partial fulfillment of the requirements for the degree of

Master of Science in

Computer Information Systems

at

Grand Valley State University

April, 2016

Advisor: Dr. Robert Adams **04/28/2016**

Table of Contents

Abstract

Testing is an important part of the manufacturing process and in order to make sure proper test coverage exists, a test plan model is created in a database and used by production testers to execute tests on the plant floor. The need exists for a fast and simple way to view and modify these test plans that will complement the existing tools used for creating and executing tests. The solution offered here is to develop a mobile first web application that can present the test plan in a user friendly way and allow editing of test parameters as required. This results in an application that can be used on any device with a browser and does not require any special software or other dependencies. This application provides a resource for management and non-technical users to visualize a test plan as well as giving technical users the ability to troubleshoot and modify test plans on the manufacturing floor without the use of large desktop applications.

Introduction

The goal of this project is to create a mobile first web application that allows users to quickly view and edit product test plans. Product test plans are essentially the recipes that are used by automated testers on the production floor to test subassemblies and completed products based on customer and internal requirements. There is a need for the ability to easily access the test plan data which is stored in a database and created using a desktop application. This new application will allow users to visualize and interact with the testing process without the overhead present with the desktop editing tool.

This new application works side by side with existing tools including the desktop editor application and the tester libraries used to query and process the test plans from the database. It allows any device with a browser the ability to view production testing specifications and procedures as well as modify testing parameters to aid with troubleshooting and new test setup. It is not meant to replace existing tools but provide an alternative to the existing tools that do not fit every use case.

The current desktop editing application relies on a large set of tools to present users with a rich editing environment where test engineers can create test plans. While it definitely fits this need, there are many users that do not want or need to install the tool set necessary to run the desktop software. This new application is meant to fill the needs of those users. It allows on-site technicians to see what testers are actually doing right on their mobile devices. It allows managers and QA users the ability to sit in meeting rooms and easily display test plans as they determine the proper test coverage needed on a new part. It allows test engineers the ability to easily change test parameters when developing new test plans during the initial setup of a new part. In short, its main purpose is to give users access to test plan data in places where access is currently limited and on devices where it is currently not available.

Background and Related Work

What is a Test Plan?

The concept of a test plan is a way to visualize various aspects of test information relevant to test engineers. As new products are developed and new features are introduced, it is the test engineers who are responsible for assuring that the products do not make it out of manufacturing without the proper testing being performed. The intent of the test plan is to define four root elements in the testing process.

1. Product specification and failure modes (Requiresments)
2. Measurement and evaluations (Tests)
3. Test station harness and wiring documentation (Fixturing)
4. Manufacturing assembly flow (Flow)

Each category can be thought of as a tree that branches out in the format described in Figure 1. As the tree is traversed, each parent element has a one-to-many relationship with its child element. For example, under the tests root element, each part number element has one or more associated groups and each group contains one or more Evaluations and/or Binding Calls. Test plans can exist for both parts and subassemblies.

The main focus of this application is to allow the user to interactively traverse the tests element and offer insight into the actual tests and evaluations that occur on a part. In order to get a better feel for the testing process, the following sections will review the other parts of a test plan and show how they relate to the tests element that this application will focus on.

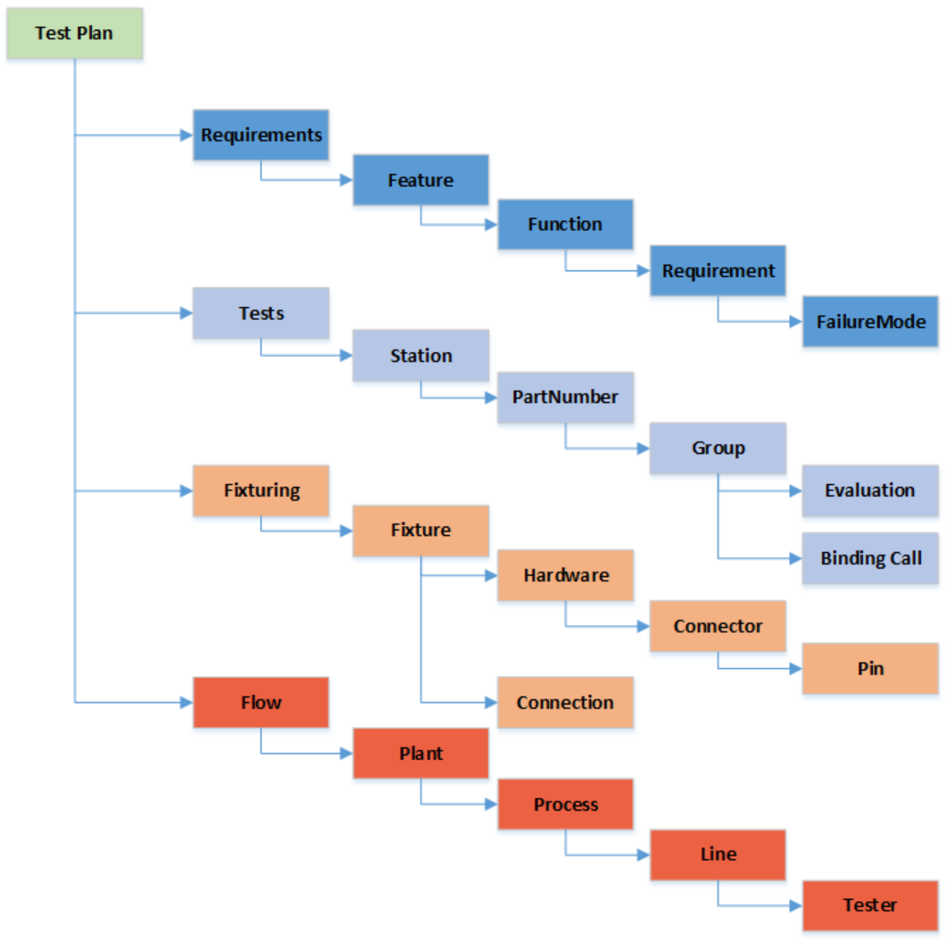


Figure 1: Structure of a Test Plan.

Requirements Root Element

The requirements root element of the test plan describes test coverage and ensures that current testing methods cover all possible failure modes of a product or subassembly. Feature elements are different functional and nonfunctional features of the device under test, or DUT. Function elements define the different functions that occur on the DUT as a result of the feature. Requirement elements define the required output of each one of the features. Failure modes define any way that the requirement can fail. The goal here is to provide adequate test coverage for each failure mode of the DUT. Ultimately, each test is designed to cover one or more failure modes defined under the requirements element of the test plan.

Fixturing Root Element

The fixturing root element describes the physical elements of running a test. A fixture element is a physical test element used on a production tester or test station. This can include a simple patch cord connection defined as a connection element. It can also be a more complex hardware fixture such as a positioning fixture that a circuit board subassembly can fit into. Each tester station will have a fixture associated with it, either a hardware fixture and/or some kind of physical positioning fixture. Once a part is loaded into a hardware fixture the test plan further defines the connector element which is the physical connector of the tester to use to probe the circuit board or connect in some other way to a part. Finally, the pin element defines the test pin on the board or connector pinout to use when running the tests. For running tests, the fixture element is ultimately part of the tester and each test station in our tests element uses one or more test fixtures.

Flow Root Element

The flow root element defines the path to get to the physical location of the tester where the test is run. The plant element defines the physical plant where the test station for the test is located. There are a number of manufacturing plants and parts may move from one plant to another depending on the current step in the manufacturing process. The process element defines what process the DUT is in during the test being performed. For example, a circuit board subassembly may be in electrical assembly and a completed part may be in final assembly. The line element is simply the production line number designation of where the tester is located. Finally, the tester element is the actual tester. The tester element relates to the tests element because each tester will have one or more stations as a part of it.

Tests Root Element

The tests root element is where this application focuses. This is where the actual tests are defined and how the tester knows what tests to run on a part. Station elements define different stations on a tester, the location of which is defined under the flow root element. Also, each test station needs a fixture which is defined under the fixturing root element. Next is the part number element of the DUT. This can either be a subassembly or a complete assembly depending on where the part is at in the manufacturing process. Next is the group element. The group element is used to further break the test up into smaller parts. A brief summary of the different groups used are included in Table 1. Within each group element are two other types of elements. Binding call elements are function calls into internal test libraries that define how the tester software is supposed to execute the given test. Evaluation elements are the evaluations that are performed on the results of the binding calls performed by the tester. Evaluations can take different forms and are summarized in Table 2. The ultimate purpose of each test is for the evaluation to cover one or more of the failure modes of a DUT and provide complete test coverage for a completed assembly.

Table 1: Types of Group Elements

Group Type	Description	Example
Initialization	Any action that needs to be taken to initialize the tester before the part is run.	Clear the previous test results from the tester application memory.
Load	Actions that need to be performed once the part is loaded.	Latch the part into position with a pneumatic cylinder.
Subroutine	A way of grouping binding calls and evaluations so that the same sequence can be reused at different points in the test.	A binding call to test resistance across two points and an evaluation to store the results. These two calls need to be made 20 times during the test.
Body	The body of the test where the majority of the evaluations will take place.	A list of binding calls and evaluations that test all the diodes on a section of a circuit board.
Unload	Actions that need to be performed before unloading a part from the station.	Release clamps holding the part fixture in place
Teardown	Actions to be performed after a part is unloaded from a tester.	Light up an indicator to let an operator know that the next part can be loaded.

Table 2: Types of Evaluations

Evaluation Type	**Description**
COLLECTION	The evaluation produces a collection of data.
EXACT	Stores an exact measurement or calculation.
LIMITCHECK	Checks that a result of a binding call is above and below limits that are set.
PASSFAIL	A Boolean pass or fail value.
SCRIPTED	Executes a custom script often using data from previous function calls.
STRING	Outputs string data.
TOLERANCE	Checks if a value is above or below a set limit.

Test Plan Database Implementation

A corporate database exists to store test plan information, primarily focusing on test elements. This immediately accomplishes two goals. First, it creates a centralized place for tests to be stored. No longer are production testers programmed independently on the production floor and no longer can test procedures and limits be changed at the machine level. All this data is now stored in one central location and production testers query the test plan from the database. This is the second goal the database implementation achieves. It allows the test process to be further automated. The tester application no longer needs to be programmed to run the test. It simply uses a library that allows it to directly pull the test plan from the database, generate a QML document from that test plan (QML is a markup language that is similar to XML and part of the Qt framework that is used extensively for developing production tester user interfaces), and then use other internal software tools to parse the test plan and run the proper test library methods and check the proper test results. This means the entire test process is controlled by the test plan that is stored centrally in the test plan database. Changes to the test procedure are done in the database and immediately used by the production floor testers.

The test plan database is stored on the corporate network on a server running Microsoft SQL Server 2008. It is a redundant system that is maintained by the corporate IT team. Authentication is currently done at an application level by the current test plan editing desktop application and granted via a permissions table that is maintained by the database administrator.

Test Plan Data Model

The test plan model used in the database is meant to replicate the test plan model shown in Figure 1. The important tables that this application deals with are shown in UML notation in Figure 2. Every version of every test plan is represented in the TestPlans table. Each test plan contains one or more Elements. Each major category of the test plan has a root element whose ParentID field is 0. After that, each element has a one-to-many relationship with each element further down the tree. Each element is further defined by a one-to-many relationship with the parameter table. Parameters can define a wide range of items, both defined and custom. The Parameter table defines the editable part of a test plan within this application. The initial version of this application allows authenticated users to edit the parameter table for each element.

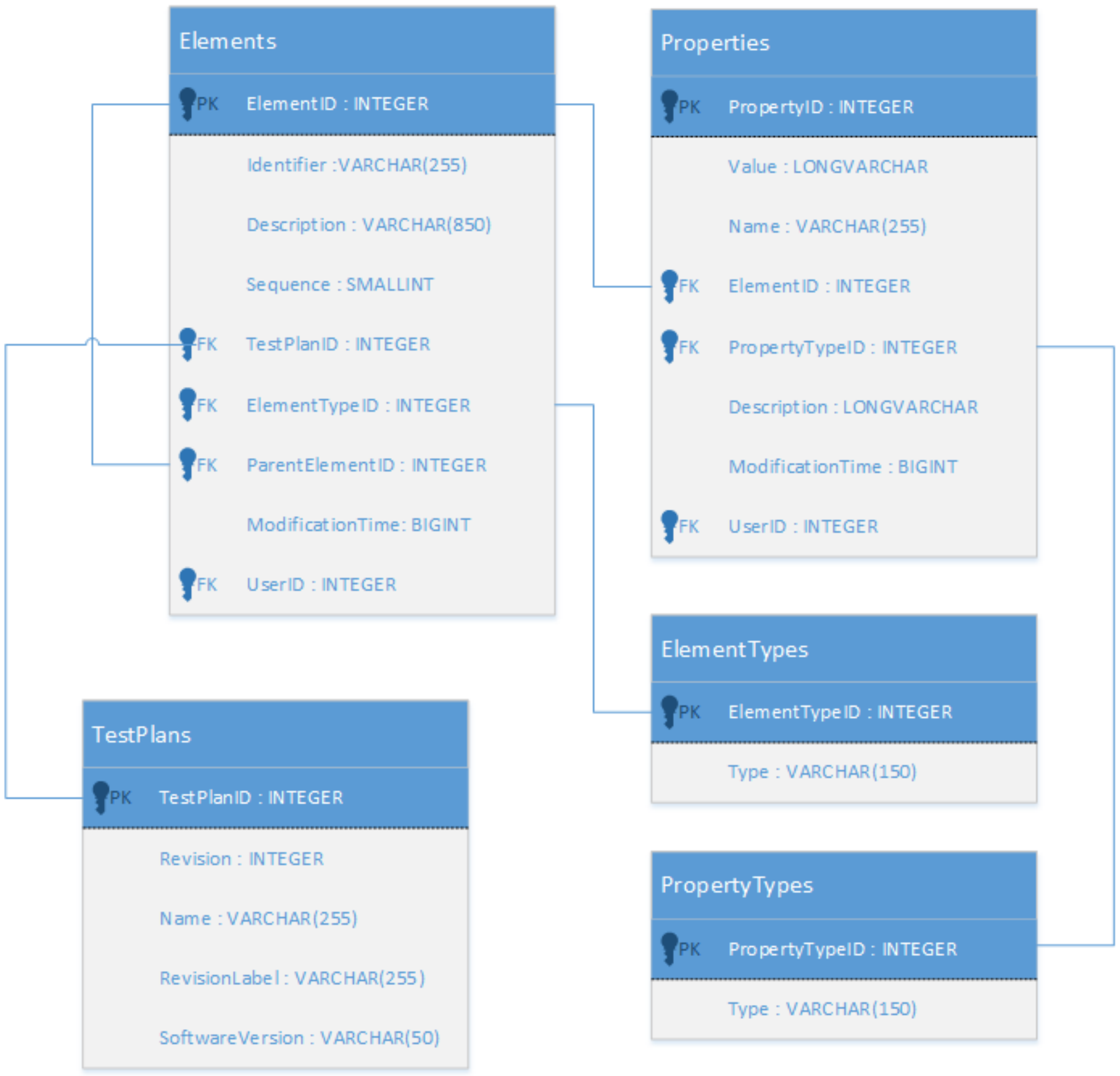


Figure 2: Test Plan Database UML

Two other tables relevant to this application are the PropertyTypes and ElementTypes tables. The ElementTypes table corresponds to the different element types defined in the test plan tree structure in Figure 1. The most important element type to the production testers and to the test engineers are the evaluation types because they ultimately define the test coverage and specify the results the production testers are trying to capture. The PropertyTypes table helps further refine what the properties mean to the element. They define the data type of each property, including a versatile variant type to encompass any property types not explicitly defined.

The database also contains a number of other tables used for defining fixture mapping, requirements mapping, and coverage mappings as well as some other supporting tables. These tables will someday be important to the overall test plan database but are not currently used in the existing environment and are therefore not relevant to this application.

There are a number of tools currently in use that interact directly with this data stored on this database. One tool is a C++ based library that defines different queries for interacting with the data. This library is used by the production testers to query relevant test plans and to package the test plan data into a QML file that the tester then uses to run relevant tests based on the part number and other attributes in the test plan.

Test Plan Editor Desktop Application

The existing Test Plan Editor software is used by test engineers to set up and define how a product is tested. They are able to define the tests that each station on each tester runs as well as set pass/fail criteria, do limit testing, and perform any type of custom operations by defining scripts inside the test. The Test Plan Editor software is written primarily in C++ and uses other custom software tools developed within the company. The most important part of the software is the ability to allow the test engineer to graphically interact with various test libraries (written and deployed to the physical testers) and offer a way to interact with and define the tests being performed in manufacturing.

Needs Beyond the Desktop Application

The Test Plan Editor desktop application has been successfully deployed and widely used in the company. It's a stable product that is used for creating, maintaining, and refining test plans used on the manufacturing floor. The application harnesses the power of other internally developed and maintained tools to deliver a robust user experience that has improved test development. However, there are some tradeoffs for the rich experience it offers.

A test plan author needs in-depth knowledge of the testing fixtures, equipment, and testing software libraries. The desktop editor offers an extensive, detailed view of the test plan with an exhaustive amount of editing options and user configurations. It allows test engineers to tell production testers how they are to test a part. It is a large part of test development for new and existing products.

There are a number of system libraries as well as a number of other libraries that are deployed with the desktop application. This makes using the software for a "quick glance" cumbersome for managers and production support personnel who do not care about the details and may not be authorized or otherwise in a position to modify or change a test plan or test limits. Many meeting rooms and other office personnel may not have a system that is configured to run the application. They either do not want or need the tool set that the desktop application requires. Having a web based application that is viewable in any browser gives everyone the ability to view test plans regardless of the system they are using. It provides a faster way to visualize the test and ensure test coverage when meeting with design teams and launching new products into production.

Another issue with the current software is that it is built for the desktop. It was not written or designed to function as a mobile or web based application. With the increased use of mobile devices in the manufacturing environment, support personnel on the production floor want a way to quickly access a test plan and see what is being tested and help with troubleshooting a failing or questionable test. The current application was not designed for this.

Program Requirements

The original proposal for this project was to design a mobile application that would initially target iOS and provide management and support personnel with a fast, mobile-friendly view of how products are being tested on the manufacturing floor. It was quickly determined that the best way to provide the mobile experience and to maximize the usefulness of the application, the target should be a mobile first web application. This creates a tool that can be used on both an iOS and Android based system and also provide an identical desktop experience that offers a scaled back feature set when compared to the much larger, expert targeted, test plan editor desktop application currently in use. By using tools such as the Bootstrap front end framework to make one application that scales to the device in use, a web application became the goal of the project.

Mobile First for Phones and Tablets

The number one purpose of this application is for it to be compatible with mobile devices. Its use will be first and foremost by users who do not have access to or the need to view the entire test plan using the desktop application. When on the production floor or in a product review meeting, users can quickly and reliably access test plan information in order to make informed decisions regarding the testing process. The most prominent mobile operating system in the application domain is iOS but there is also plenty of Android so making it a web application should allow access for all users.

Must Work With Existing Data

The production database exists on a server running Microsoft SQL Server 2008. There also exists a test bed database on another instance of Microsoft SQL Server 2008 that is used by developers of test libraries and other applications that interact with the test plan data. For this new application, the test bed exists on a local instance of MySQL server. For this reason, it is extremely important that the new application be able to seamlessly integrate with whatever type of database is in use. If the database changes, the software needs to easily accommodate that change. Also, the database structure cannot change to accommodate this new application in any way. It is important that the new software use the existing DDL and tables the way they currently exist. The current database and other supporting software will not make any modifications to accommodate a new application. There are too many tools that already make use of the existing data model and they need to continue to function properly.

Provide User Authentication

The new application also needs to incorporate some form of authentication. The current desktop editing tool for creating test plans uses a table in the database that users authenticate with. This table defines rather a user has read only or full permissions. For this new application, it is fine to continue to use this local form of authentication. There needs to be 2 levels of authentication, one for just viewing test plans, and one for editing test parameters. This new software does not support creation of new test plans or even new tests, but it does give authenticated users the ability to modify testing parameters and evaluation criteria.

Must Be Fast Even With Large Test Plans

One of the biggest challenges in creating this web application was the size of the test plans. Test plans can be extremely large by the time each element and each parameter for each element is queried. This means that the application needs to efficiently deal with large amounts of data. Since this application is web based and meant as a "quick" way to interact with a test plan, it needs to load this data efficiently and not cause a long delay when traversing thought the test plan.

Needs to Provide All Parameters of Binding Calls and Evaluations

In order to fulfill the needs of technicians and users that are setting up tests on the production floor, the application needs to allow editing of parameters. This will increase the usefulness by expanding the application from being simply just a viewer to a tool that allows real time interaction with the test plans. The desktop application contains tools that drill into test libraries and gives the user creating test plans a view into the testing libraries. This allows the user to select what functions to run and how to apply the results of those functions to the evaluation criteria. This web application will not allow creation of new function calls and evaluation nodes because it lacks the tools needed to read test library data. However, it will allow the user to modify function parameters and evaluation criteria if needed.

Implementation

The main focus of the development part of the project was to use and become familiar with new technologies. The decision was made to use a PHP based framework since it is a common programming language. Apache is used as the server application for the development environment since it is a very common server application. After doing some research on the latest PHP based web development frameworks, Laravel was chosen as the PHP framework to use. Other possibilities included PHPixie, Nette and CodeIgniter but in the end, Laravel won out mostly because it seemed to have the most online support among the candidates looked at.

The application development environment is a Windows based machine. Because of this, a Windows compatible web application development environment is needed. The decision was made to use an Apache, MySQL, and PHP software stack. There are a couple of options that seem fairly popular to use on a Windows machine. The option used for this project is an application called WampServer. It was very easy to install and setup and includes all the tools needed to get started. It was a fairly easy process to get Laravel set up and working in the WampServer environment.

In order to keep everything local and also get some experience with another new technology, MySQL was used to host the development database. It is part of the installed development environment so no special configuration was needed. Laravel's built in tools were used to generate the tables in the database and they match the tables and datatypes used in the production Microsoft SQL Server database. The issue that needed to be dealt with was the transfer of data from the production database to the local development database. To accomplish this, a small command line program was created using C# that simply queries the data from Microsoft SQL Server and inserts it into the local MySQL instance. It is a long process since the production database contains millions of records, but that is not important. The software can run overnight and, in the morning, the development environment includes a replica of the SQL Server data on the local MySQL instance. This tool would prove to come in handy as it ended up being used a couple of times as data became corrupt or otherwise unusable.

For a coding IDE, NetBeans was selected. The reason for this is that a development environment is desired that plays nicely with PHP. It is also a new technology so the experience is worthwhile. It ended up integrating well with the Laravel project and offered some code completion. There are some open source projects that attempt to add some extra Laravel specific code completion to the NetBeans environment but none of them seemed easy to set up or use so only the basic PHP code completion was used. This ended up not being a big deal but it would have been nice to have this working.

Features Used in Laravel Framework

Laravel provides a powerful command line tool called Artisan that allows the user to create templates for new items and run commands that allow the user to do things such as create database tables and set up authentication. Artisan was used frequently in the beginning when creating new items. Later on in the development cycle, familiarity with the framework allowed for quick creation of new objects without the use of Artisan. However, there were still many features of the tool that were useful including the database migration tool and the built in authentication system.

Authentication is built into the Laravel framework and is set up with a basic Artisan command that creates the needed models, views, controllers, and migrations needed to set up the user table in the database. This application uses the default settings for the database authentication table along with the default model and controller used for authenticating users and letting new users register. As this application is integrated into the test plan tool set, it will need to support some extra functionality that allows a user to be marked as administrators with full control or just viewers without editing power. Rather or not this will require a more customized approach to the authentication system is yet to be determined.

Laravel makes use of database migrations as a way of defining the database schema. Inside a migration, the user can define the database table name, column types, add constraints, add data to a table, and define how to drop the table. Artisan commands are used to either create tables and associated entities by running a make command or rollback changes using a rollback command. This is accomplished by defining two functions in each migration. The first function is Up() and it is called by the Artisan tool when the user runs a make command on the migrations. This function is used to define the table, data, and constraints. The second function defined in the migration is Down() and it is the function run by Artisan when the user runs a rollback command.

Some of the tables created by this application are used as simple lookup tables and they are populated with data by the migrations. For example, there is an ElementType table that defines what type of node in the test plan the element is. The options are limited. There are only 23 different element types that can exist so adding this information to the database through the migration makes sense. With larger tables, adding data through the migration is not efficient and does not make sense. The larger, more complex tables are populated using the seeding tool described below.

The migration tool is also used to add constraints to the data tables. This project includes a migration that does not add a new table but instead just adds all the needed foreign key constraints. This migration is run after filling the database with data. This makes updating the tables much easier since each update is not checked against the constraint. There are also several constraints and indexes in the production database that are not replicated in the development MySQL instance. This makes it easier to directly manipulate data as needed in the development environment without having to worry about constraints. Most of the constraints that are not foreign-key constraints deal mainly with how new elements are added to the test plan and for the purpose of this application, these constraints are not relevant and only add undue complexity to the development database.

Each table is represented in Laravel as a model. Models are used to create objects that correspond to each table. Laravel refers to these models as Eloquent ORM models. These models allow the application to query data from the database as well as update data stored in the database. The models also define relationships between the models. For example, each TestPlan object has one or more Element objects and each Element object belongs to exactly one TestPlan object. These relationships are all defined inside the model and they allow the user to easily access a TestPlan's elements. In the Element model the following function describes this relationship:

```php
public function testPlan()
{
    return $this->belongsTo('App\TestPlan', 'TestPlanID');
}
```

Once defined, the TestPlan object that the element belongs to is easily reference in the PHP code in the following manner:

```php
$this->testPlan();
```

Relationships defined in the models are used extensively in this project when querying data from the database. One special type of relationship that is used to navigate through the property table is a self-referencing relationship. Each Element has an optional foreign key defining a parent element. By defining this relationship, the application can quickly access the Element tree structure of the test plan.

Controllers are used in Laravel to handle routing requests. This project contains a routing file that accepts http routes from the server and then calls into the specified Controller. The controller uses models to perform queries on the referenced models and return either a view or other information to the requestor. For example, when the application requests child elements for an expanding node, an AJAX request is made on the following route:

```php
Route::get('/element_children', 'ElementController@childrenList');
```

This route calls into the ElementController function childrenList(). The AJAX request also provides 2 arguments that define the ElementID of the element to query and another argument that defines the level of the node being requested. This level argument helps to format the returned data with the proper indentation and make the html render in a nice, readable way. The data from the query along with the level data is passed to a view.

```php
public function childrenList(Request $request)
{
    return view('elements.childrenList', [
        'element' => Element::where('ElementID', $request->arg1)->get()->first(),
        'level' => $request->arg2
    ]);
}
```

The view handles the html rendering and is returned to the requester. In this case it returns a group of list items to the requesting AJAX function. Laravel views use a templating engine called blade to offer powerful and flexible html rendering. The childrenList blade template is fairly large because it ends up processing a bunch of other parameter information that help make the elements more descriptive on the generated webpage. One important thing to point out is that the parameters passed to the view are easily accessible by using a double curly brace. Lines in the code that perform functions are preceded by @. For example, the following line sets up a for loop that increments through each child element of the element returned by the query performed in the controller.

```
@foreach ($element->children as $child)
```

Then, each property of the child element can be accessed as follows:

```
@if (count($child->children))
<span id="icon{{ $child->ElementID }}" class="icon expand-icon glyphicon glyphicon-plus">
    <i id="spinner{{ $child->ElementID }}" class="fa fa-spinner fa-spin" style="display:none;"></i>
</span>
@else
<span class="icon glyphicon"></span>
@endif
```

This particular piece of code determines rather or not this child element also has children. If it does, a plus icon is added next to it signifying it can be expanded. There is also a hidden spinner that will be displayed when the AJAX call is executing that will query the elements children.

This is essentially how the entire application functions. Requests are routed through the routing file to the appropriate controller. The controller uses the defined models to query data from the database and also pass data into a view. The view uses the blade templating provided by Laravel to render the html that is returned to the requester.

Database Seeding Application

There is a built in database seeding tool in Laravel. This, however, did not fit the needs of this project. The seeding tool basically creates random data for doing development work. This application needs to operate on specifically formatted data and the work to develop PHP code that would create the properly formatted data seemed to be a waste of time. To seed the development database a small command line tool was developed using C#. This custom seeding tool simply queries the existing SQL Server database and inserts the data into the local MySQL instance. Since the migrations used to create the tables make sure the datatypes are compatible in each database, the data is easily transferable.

The major issue with this command line tool is that it is very slow. There was no attempt to make it efficient. It simply iterates through millions of records and occasionally prints out some status information to the command line. Speed, however, was not really a factor. This tool was only needed a couple of times during development. It was used once on initial setup and then once when the MySQL instance became corrupt during the development process. When it did need to run, it would just run overnight and in the morning, a full MySQL instance exists with about 2GB worth of data.

Custom Tree View for On-Demand Loading

The initial idea was to use some kind of Javascript library to display the test plan in some kind of tree view. However, there did not seem so be any tools that fit the job. Most required the entire test plan to be loaded up front and then use Javascript to display the items as needed. This works fine for smaller data sets but the size of the test plans made the application react extremely slowly and were therefore unacceptable. The decision was made to create a custom tree view and accompanying Javascript that would make the process fast and look nice.

For the custom tree view, the entire test plan is essentially one large unordered list with formatting to make it look like a dynamic, expandable/collapsible tree that initially contains only the root elements of the test plan. New elements are added as list items to the unordered list and are formatted as needed based on parameter data. They also have their id attribute set to the ElementId primary key of the element model. This gives Laravel a very easy way to reference the needed element based on interaction with the user. The list item is also given a custom attribute that holds the ElementId of the parent element. By saving the parent element id, the list can be collapsed without have to process another AJAX call.

When the user clicks on an expand icon, an AJAX call is performed. The route triggered by this click is the same as the one described in the previous section. The request gets routed by Laravel to the controller that creates one or more list items based on the number of children the clicked element has. This group of list items is returned to the requestor, which is the AJAX call, and the list items are inserted into the unordered list in the proper position using some Javascript and JQuery manipulation of the DOM. By applying the proper formatting to the returned list items, the unordered list looks to the user like an expanding tree. It also means that there is almost no upfront processing to create the tree. All child elements are queried and list items created on demand.

When the user clicks on a collapse icon it triggers a Javascript function that looks at the id of the clicked element. Since each list item also has a custom attribute that contains the id of the parent element, the function uses JQuery to find elements with the parent id of the clicked element and removes those items. This function is called recursively so that each deleted items children are also deleted. This creates the effect of having a clicked node collapse. By saving the parent element id in the DOM, the time needed to send AJAX requests back to the server is eliminated. This makes collapses almost instantaneous as opposed to having to recursively request element parent id's to determine if they should be removed from the list.

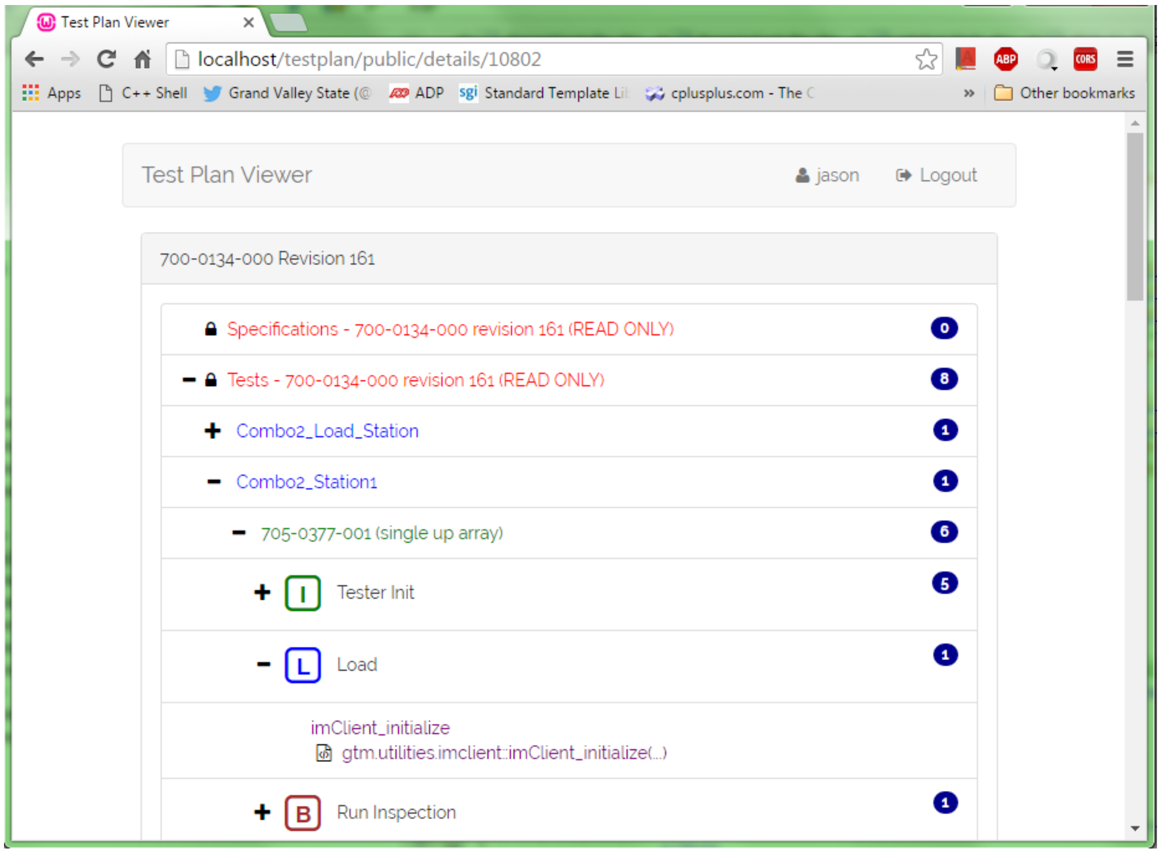
Bootstrap Front End Framework

Bootstrap is the front end framework used for this application. Using Bootstrap allows the application to be responsive giving it the same look and feel rather on a desktop browser or on a handheld device. It's important to deliver a consistent experience across all platforms and Bootstrap seems to be a very popular option for providing that functionality. Elements on the screen scale as the size of the window or the device viewing the page changes.

The application also uses Bootstrap's modal window to display a list of parameters. When the user clicks on an element, a modal loads that displays all the parameters for that element. These individual parameters are editable by the user. They can update the input and submit the change back to the server. This is handled by another AJAX call and does not require reloading of the page. There is currently no validation of parameter data. The database will accept character data so validation may be needed in the future on the server side.

How Does Application Work?

When the user connects to the home screen they are presented with a welcome message and the ability to log in or register. Once the user registers or signs in, they are presented with all the available test plans. Selecting a test plan brings up a list of all available versions of that test plan. Once a version is selected, the user is presented with a collapsed tree view of the test plan. From here, the user can interactively browse the entire plan and can view and modify the parameters of each element as needed.


Test Plan Viewer

localhost/testplan/public/details/10802

Apps C++ Shell Grand Valley State (@ ADP Standard Template Li cplusplus.com - The C Other bookmarks

Test Plan Viewer jason Logout

700-0134-000 Revision 161

Specifications - 700-0134-000 revision 161 (READ ONLY) 0

Tests - 700-0134-000 revision 161 (READ ONLY) 8

Combo2_Load_Station 1

Combo2_Station1 1

705-0377-001 (single up array) 6

I Tester Init 5

L Load 1

imClient_initialize
gtm.utilities.imclient::imClient_initialize(...)

B Run Inspection 1

Figure 3: Loaded Test Plan

Results, Evaluation, and Reflection

Initial reviews of the application have been largely positive. It meets all the initial requirements and most notably it is fast. This was the biggest obstacle to overcome and the majority of time was spent figuring out how to make it fast. This application will live on outside of this project but where or who will maintain it is an open question. It meets a need with the organization and shows what is possible with a web based application that interacts with the test plan database.

Another problem during development was just the speed of development. Using tools that were all new slowed the development time to a crawl. This limited some of the added features that would have been nice to spend time developing. For instance, it would have been nice to have it connected to the actual production database or maybe try to get authentication integrated with the local Active Directory domain controller.

The most obvious criticism of the application in its current state is that it is not yet live with production data. This proved to be a large stumbling block that was not discovered until late in the development process. The development environment this project uses is 64-bit. When trying to connect to the production server, there is a setup file in Laravel that instructs the server where to find the database. This should be a simple change, however, the Microsoft SQL Server PHP driver needs to run in a 32-bit environment. This means that the application will need to be deployed on a 32-bit server. Updating the development environment to accommodate this change was one of the items that got pushed back due to the slow development cycle.

There are also a few other changes in the development process that would have been nice. There was a lot of time and effort spent developing migrations and models for all the tables. This proved to be unnecessary and not all where used. Also, a trimmed down version of the MySQL development instance would have probably been a better way to go. Another issue that should have been dealt with is to further customize the user table and authentication system. The user table used in the development database is the only table that is not identical to the production database. Not enough time was allocated to getting this functioning with a custom user table.

Conclusions and Future Work

This project provided a good learning experience and exposure to many tools that were new and interesting to explore. The application will live on and development will continue. The immediate goal is to get the application released in a beta form and get user feedback to determine what new features and refinements are needed. A brief description of upcoming steps and possible new features is now presented.

Dedicated Server

For development the project was build and is run on a development PC running its own local WAMP stack. The project will need to be moved to a permanent home on the corporate network. The procedure to do this is to add a virtual machine to the companies server stack. The new virtual server will be registered with the internal DNS server and provide anyone on the internal network access to the application. This will also give the server a friendly name and allow users to connect using this new friendly URL.

The 64-bit problem with Microsoft PHP Driver

The Laravel framework offers a very simple solution to use whatever database is needed for the application. Databases can be changed by simply defining a connection in the settings file. As long as database schema and structure is equivalent, then integrating the current production database is a simple process. The MySQL development database used for the project is a replication of the existing production system and migrating to the production server should be fairly simple.

Active Directory Domain Server Authentication

Authentication in the current production database is local and not tied to the companies Active Directory domain server. By default, users are granted read only permission. While the viewer can and may be left visible to everyone, it needs in place the ability to validate users with the Active Directory server. This will need to be further researched and a custom authentication system may be required.

More detail on binding call parameters

The parameters of binding calls are easily viewed on the parameter detail modal. Future revisions will need to add these parameters to the function call description and subsequent evaluations will need to detail which values are passed as which parameters. Essentially, a decision needs to be made as to how much information should be shown on the tree without the necessity to drill into an element. There is a lot of information currently but there may need to be more added if users determine that the current view is not enough. It ends up being a balancing act between how much information can fit nicely on a small screen and how much information is sufficient to maximize the application's usefulness.

More Detailed Parameter Window

Certain parameters do not need to be shown and should not be editable. For example, when a fixture element is being displayed, it is not necessary to display fixture row and column information. This information is already queried and displayed as part of the list element description and does not need to be part of the properties table. Every property has a property type that can be used to further categorize where and how properties are presented. Again, the level that this needs to be refined to will be largely driven by user feedback.

Expose a Service

There already exists an extensive tool set that uses the test plan database. There may be some functionality that this application could expose as a service that could replace or enhance some of those tools. No specific plans exist but there are possibilities.

Log Changes When Editing Parameters

When parameters are edited, it would be nice to log the change and the user that made the change. The structure exists in the database to store this information and it is used by some of the tools that interact with the database. It will be an open question as to rather or not future versions of this application will allow parameter manipulation, but if they do, there will need to be some kind of system in place to hold users accountable when changes are made to a test plan.

Applying New Technology

This project has been an overall positive learning experience. It gave exposure to a number of tools that were new and offered a great deal of insight into web application development. Some tools used in this project have already been put to use outside of this project on other applications. Future development of this application should ensure that it becomes a part of the tool set used to deal with creating and maintaining test plans.

Bibliography

Bourdon, Romain. *WampServer*. Computer software. *WampServer - Apache, PHP, MySQL on Windows*. Vers. 2.5. N.p., n.d. Web.

Gandy, Dave. *Font Awesome*. Computer software. *Font Awesome, the Iconic Font and CSS Toolkit*. Vers. 4.6.1. N.p., n.d. Web.

McDade, Jack. *Laravel*. Computer software. *Laravel - A PHP Framework for Web Artisans*. Vers. 5.1. N.p., n.d. Web.

Otto, Mark, and Jacob Thornton. *Bootstrap*. Computer software. *Bootstrap - The World's Most Popular Mobile-first and Responsive Front-end Framework*. Vers. 3.3.6. N.p., n.d. Web.