Brandon Messineo

Dr. Jackson

SmartList Senior Project Paper

We live in a world where technology is used frequently. We use technology to tell the time, predict the weather, write a paper, or communicate with others. Technology can help make our lives easier. Even though people have adopted technology in nearly every part of their lives, many improvements can be made. Most people still use primitive tools to create lists. They write down lists on paper or try to remember them in their head. Although simple and easy to create, there are many limitations to managing lists this way. It is cumbersome keeping track of collections or other large-scale lists. Information about each item in the list must be accounted for as well. SmartList solves this problem. It is a web service that keeps track of the collections so that a person does not have to. It makes their life easier by gathering and then organizing the information. However, the website has more to offer than a place to merely create and manage lists.

SmartList is designed so that each list is a global table and all information is publically available and editable. There is also a personal aspect to the design. Users can add an item to their profile to build personal lists, such as using the book table to create a list of books they have read. A picture can be labeled private so that only the uploader sees that picture. Information can also be personal such as rankings and ratings for any attached item. One final, yet critical, aspect of SmartList is the sharing capability. It is a community-driven site because all knowledge is shared among users. The website can provide up-to-date information to the user. People can view other user lists to compare and discover new items to add to their own lists. The development of the website reflects these key factors.

The website is hosted on my personal web server to allow for the most up-time. The service's flexible design allows any type of list to be created. HTML, JavaScript, and CSS skills are needed as a basis for the foundation. The higher-level languages and frameworks used to make the service are PHP, MySQL, jQuery, and Bootstrap. These additional languages and frameworks provide the tools to develop a feature-rich experience. Security is of major concern because the website will process sensitive information. Efficiency is also very important because the website will handle large amounts of data.

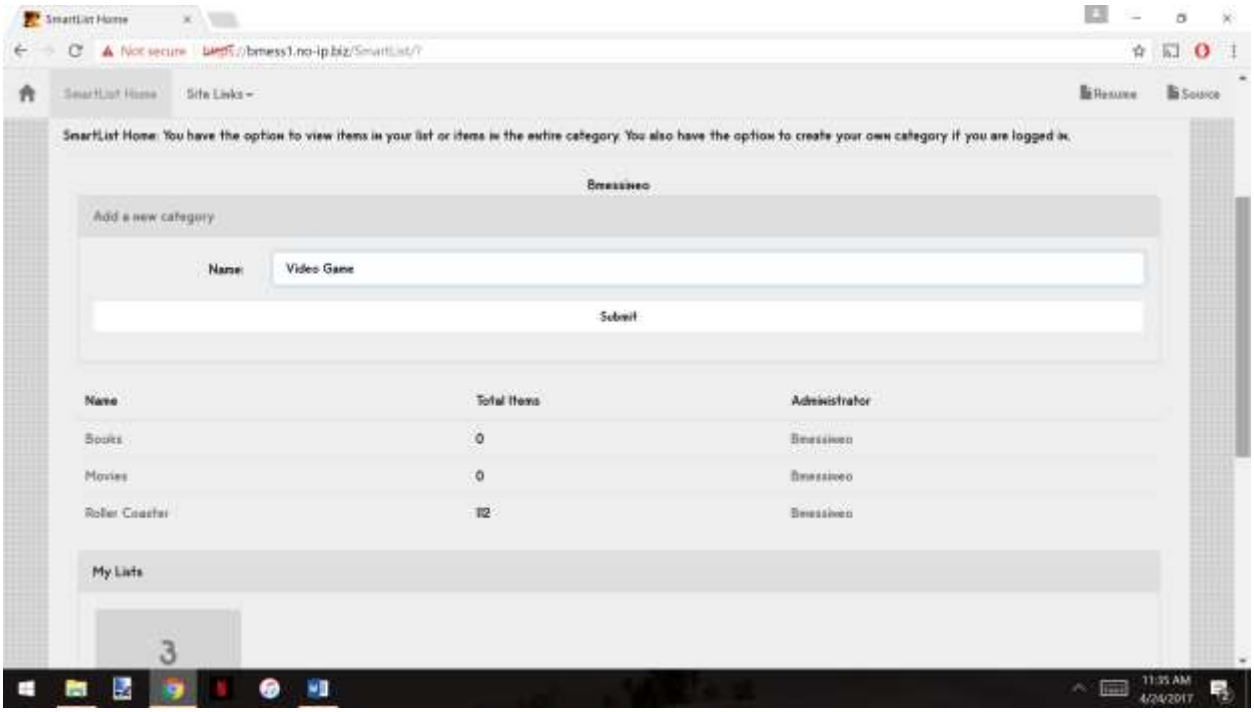
I had some existing code that I built on for the current project. I previously created a website to keep track of roller coasters that have been ridden on. The code was an important reference for development. Some functions such as file management and image storing were already created and aided with the design process.

Demonstration

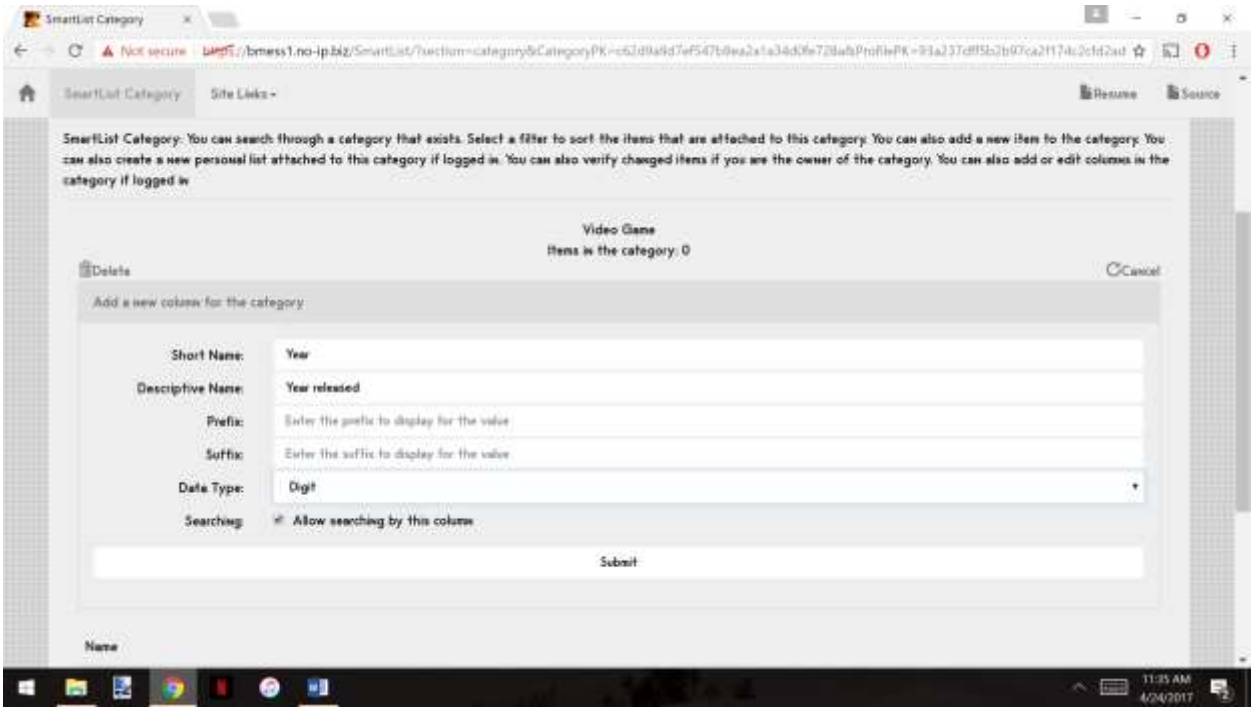
In order to understand the features and terminology of the project, an example is given. A user wishes to keep track of video games that they own. Items are individual entities. In our example, a video game is considered an item. Categories are a collection of items. For example,

video games are considered a category. Columns are the information provided for an item. An example of a column is the year an individual video game was released.

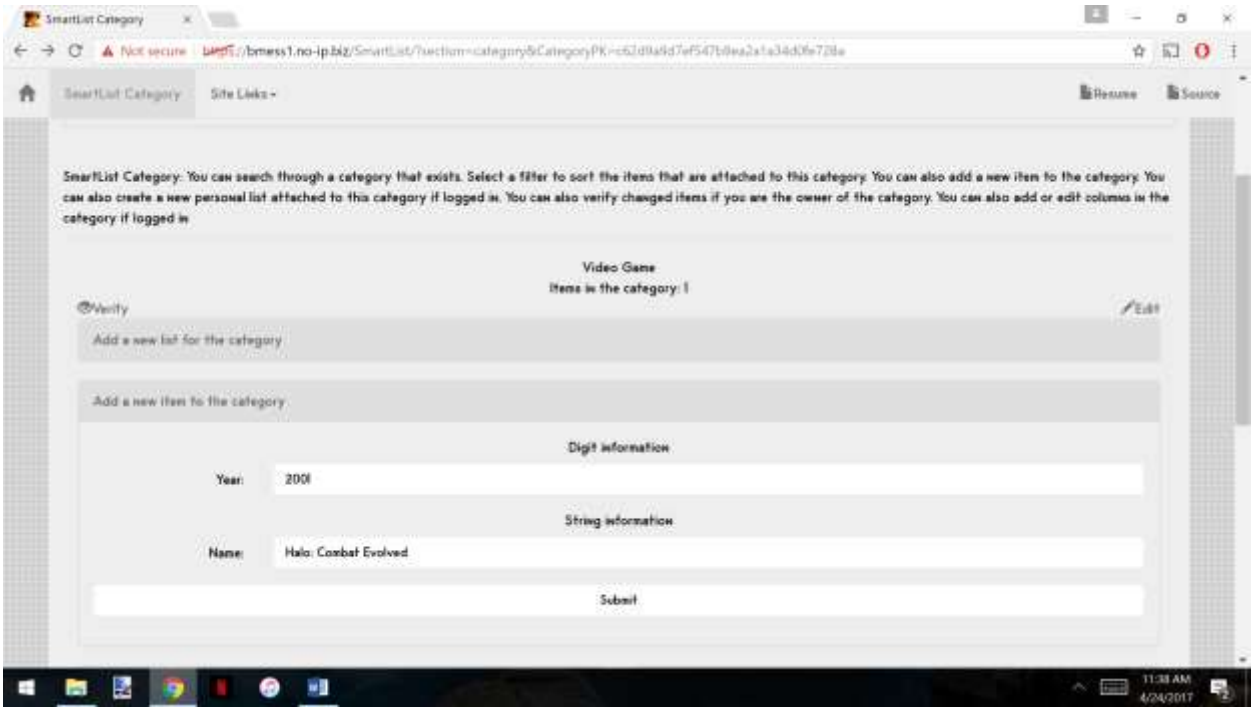
Upon logging in, a user is presented with the current categories that exist in the database. We want to create a “Video Game” category so we simply type in the name of the category and click submit. A picture of this action is displayed below:

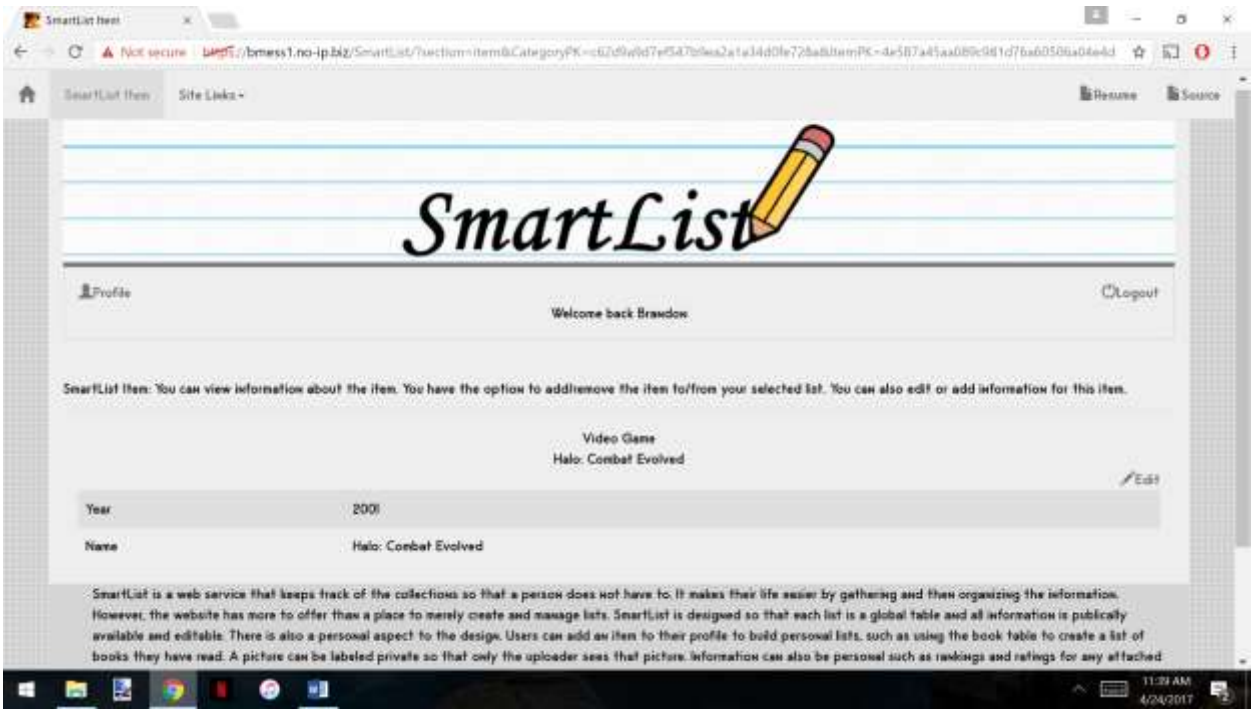


When a category is created, its “name” column is also created. We want to add more columns because we want to keep track of more than the name of the item. When a user selects the option to add a new column, they are presented with another form. The column short name represents a one-word or two-word name of the column. The column descriptive name represents a more detailed name of the column. The prefix is a string that can be placed before every value that is displayed. Similarly, the suffix is placed after every value. Units of measurement are a typical use of the prefix or suffix input. The data type represents the value type of the column. The data type options are digit, number, object, and string. Digit and number columns are both integers. The only difference is that digits are not formatted with a comma. String columns are self-explanatory. Object columns are described as groups. A genre is a typical example of an object column because multiple items can belong to the same genre. Finally, the searching checkbox allows the column to be one that is searchable. In our video game example, we want to keep track of the year that a video game was released. We type in “Year” for the short name input, “Year released” for the descriptive name input, and “Digit” for the data type. Finally, we want to allow searching for this column so we check the “Searching” checkbox. We then click submit. A picture of this step is displayed below:



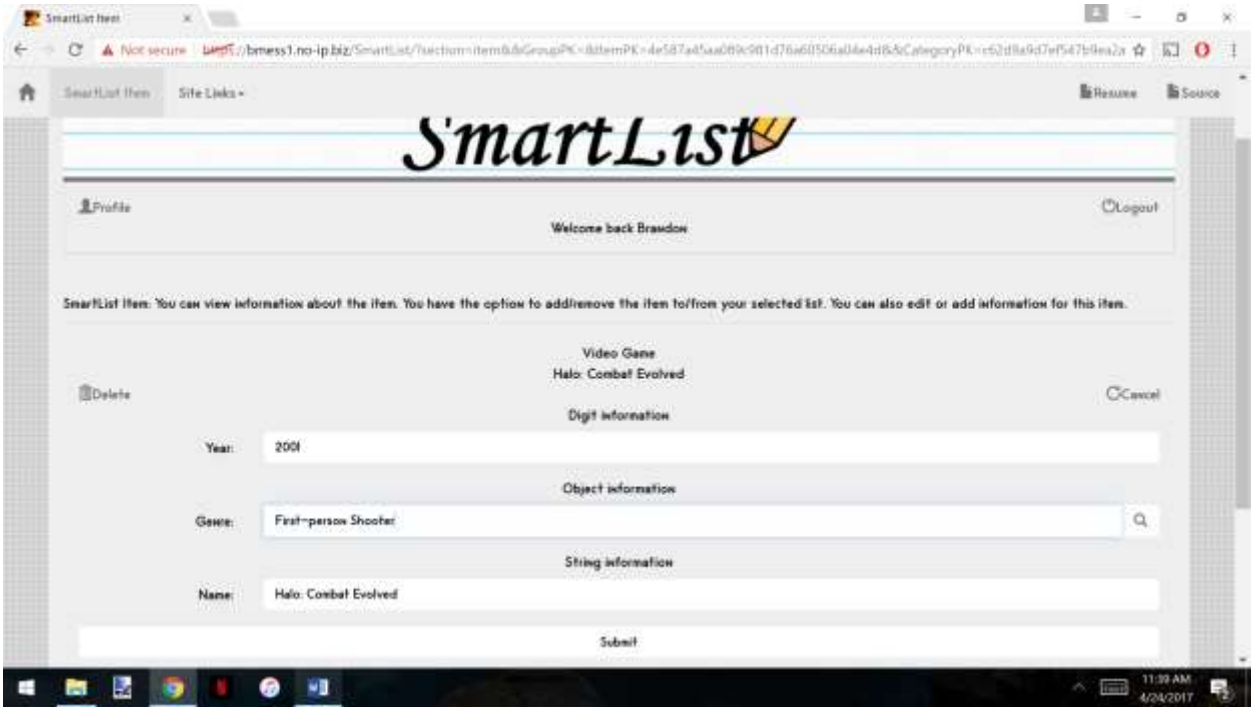
Now, we can track video games by name and year released. We additionally want to keep track of the video game genre so a column is created for that too. After the genre column is created, our category setup is complete. The next step is to add an item to the category. A form is displayed with all of the columns that exist in the category, except for object columns. The reason is that an item needs to be created before it can be attached to groups. We want to add "Halo: Combat Evolved" to the category so we fill in the form. We type in "Halo: Combat Evolved" for the name input and "2000" for the year input. We then click submit to add the item. The next page shows the new values for the item. Two pictures are displayed below for this step:

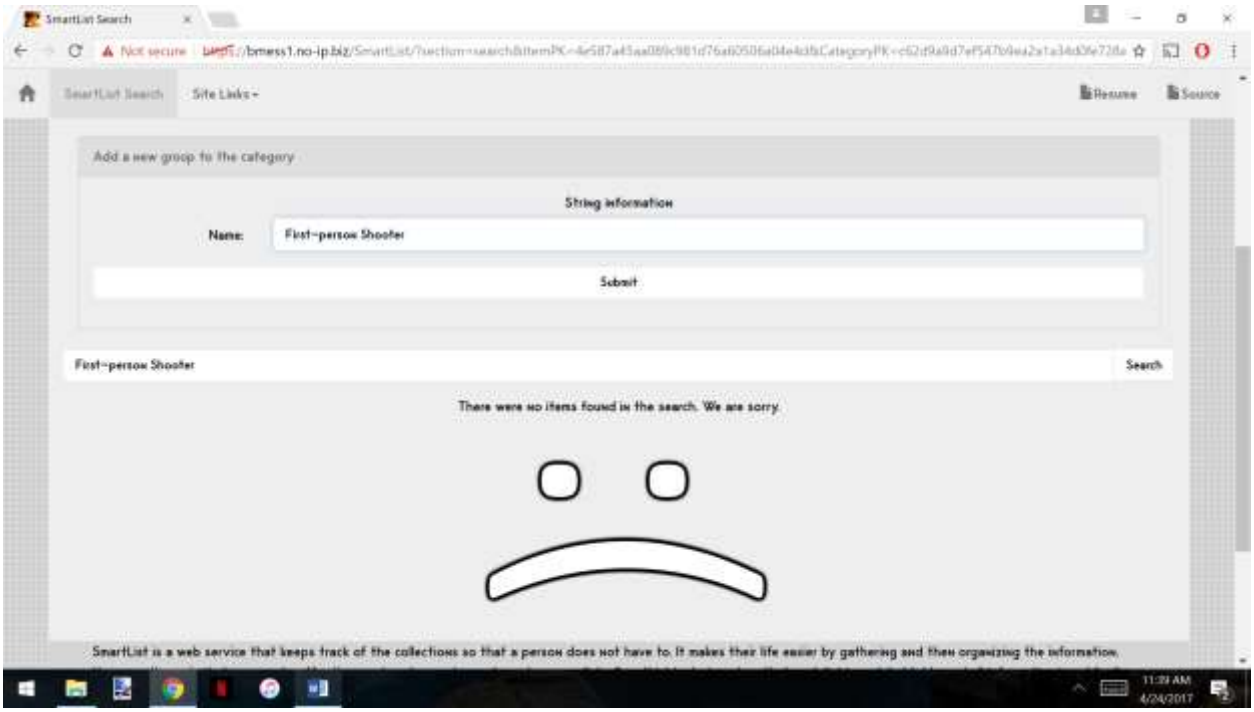


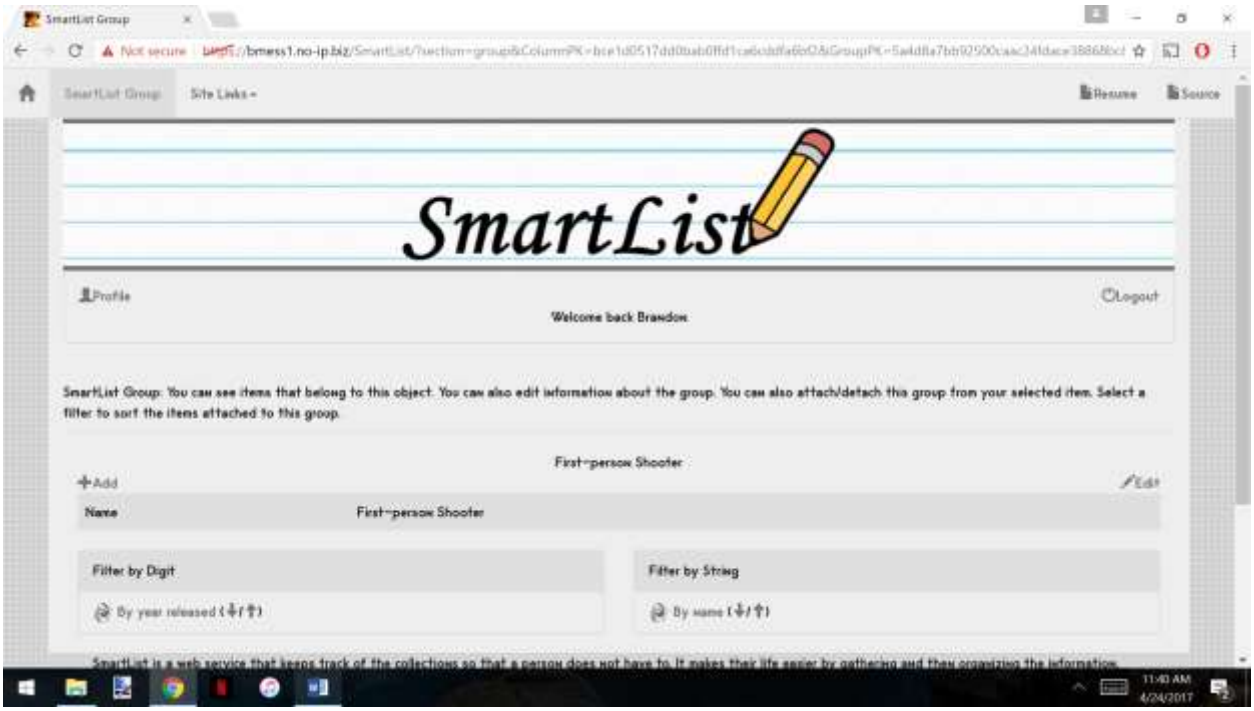


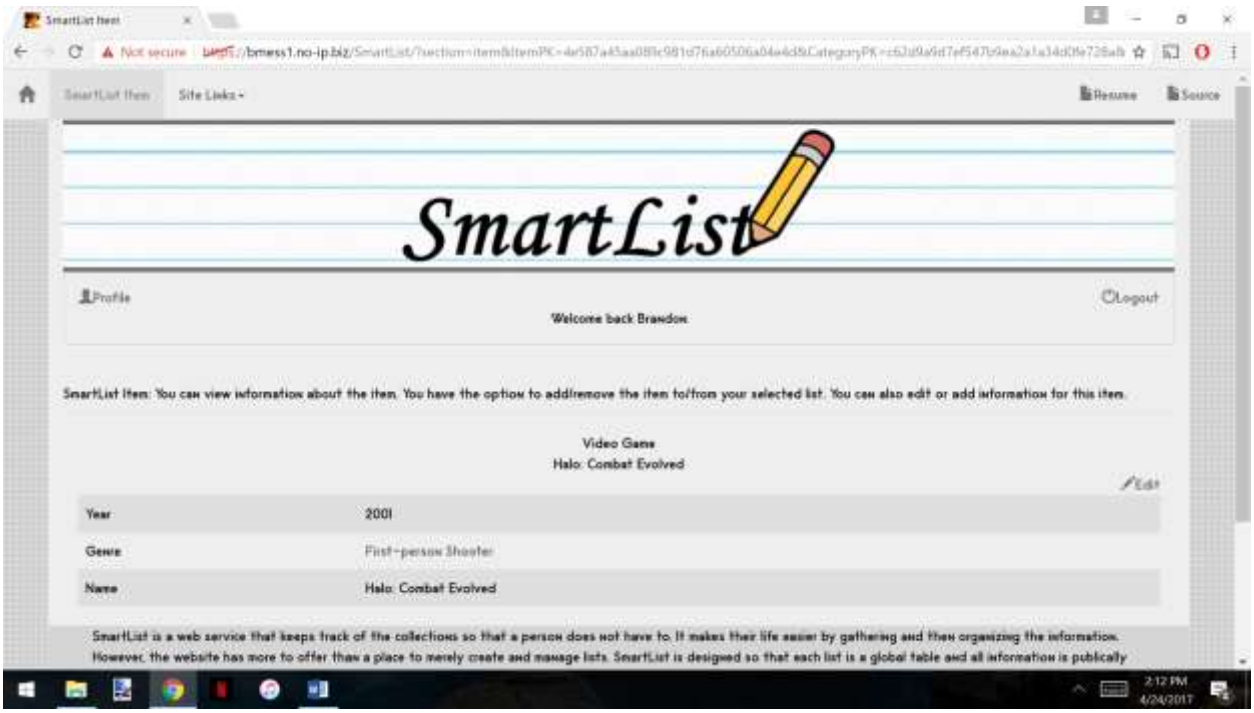
We then want to add the genre for this item so we click on the edit button. The same item form is presented but with the option to add objects. These inputs operate a little differently from other inputs. After typing in a value, the search button is clicked. The user is then presented with the objects matching the search. If there is no match, the user can add a new object. Finally, when an object is selected/created, the user is redirected to the object page. To attach the item to the object, they simply click “Add”. In our example, we want to attach “Halo: Combat Evolved”

to the “First-person shooter” genre. We type in “First-person shooter” into the search box for genre and click search. Since it is a new category and no genres exist so far, we need to create the genre. We type in the name of the genre and click submit. The “First-person shooter” genre now exists so we simply select the “add” button. Now, “Halo: Combat Evolved” belongs to the “First-person shooter” genre. Below are four pictures displaying this step:

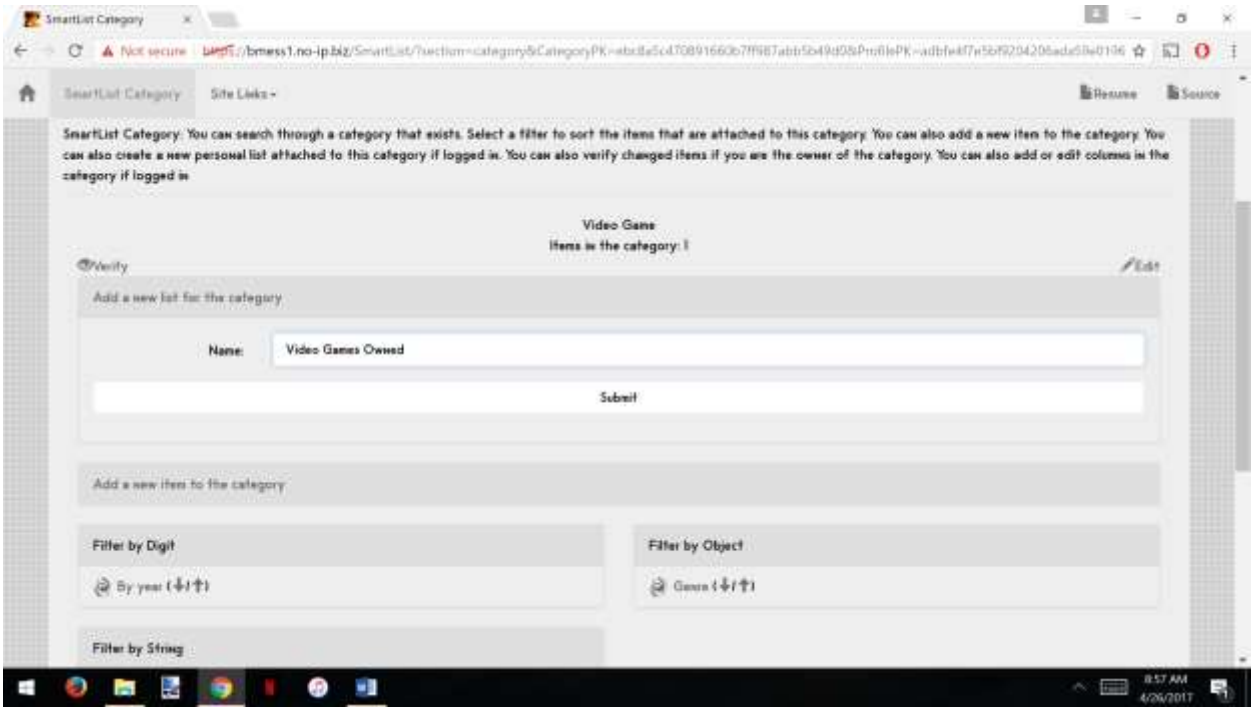




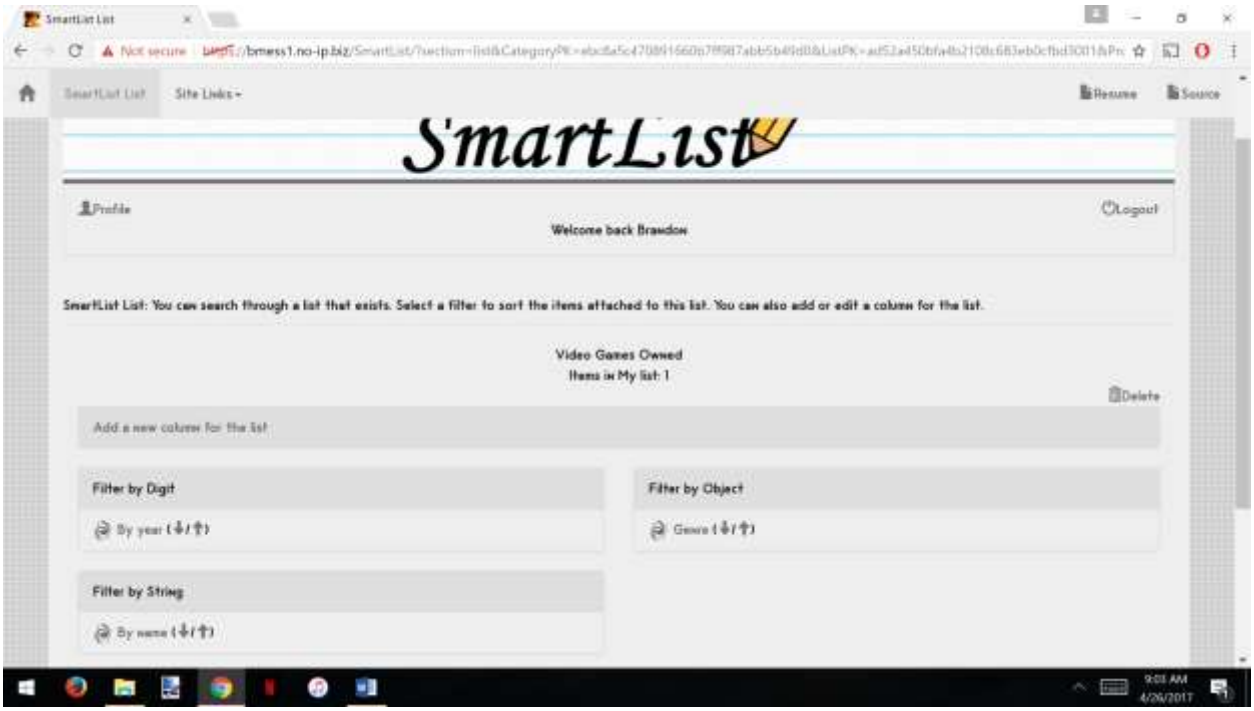


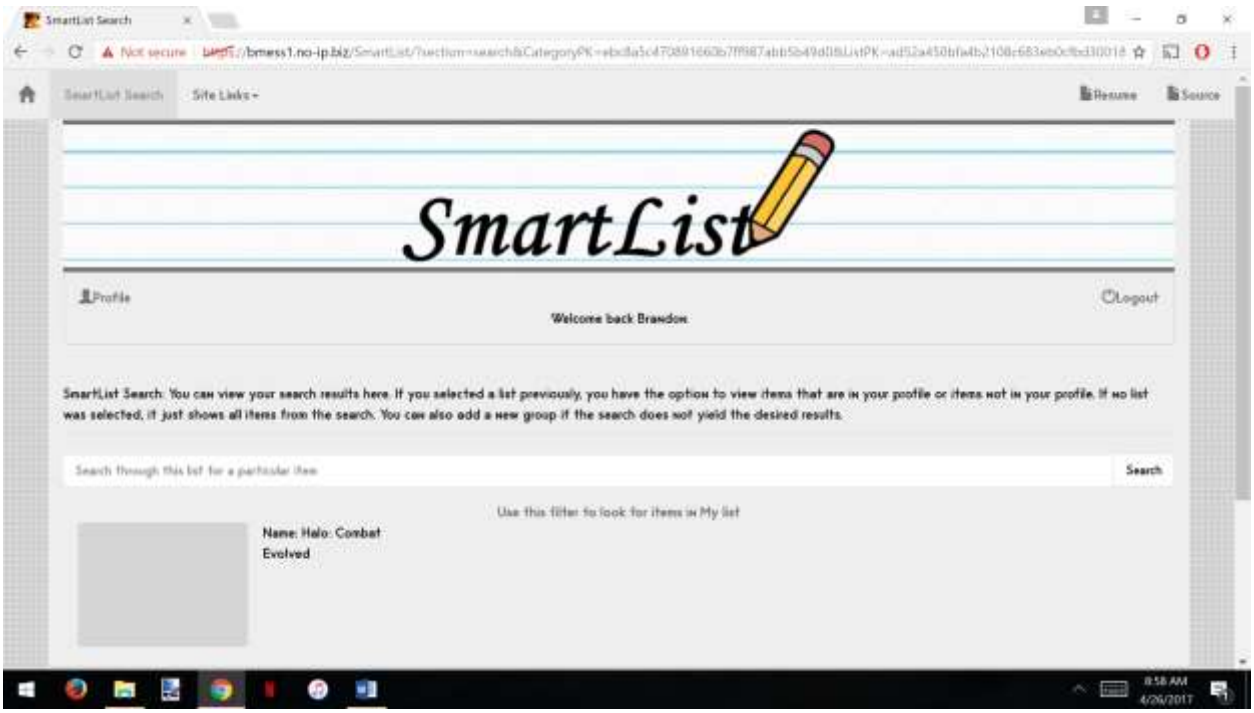


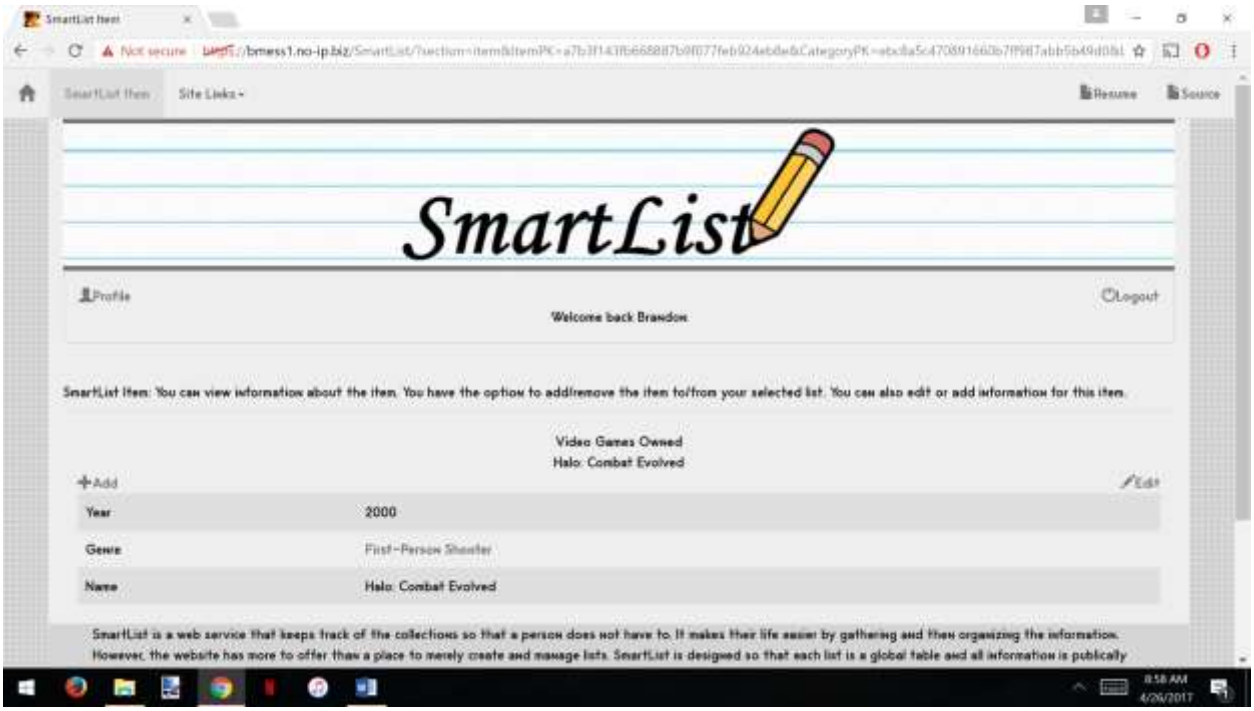
Now that items exist in the category, we next want to build a list. To create a list, a user types in a name and then clicks submit. In our example, we name our list “Video Games Owned” and click submit to create the list. A picture of this step is displayed below:



Lists can have their own columns so the option to create one is available. Possible list columns for “Video Games Owned” are “Rank”, “Rating”, or “Date purchased”. After the list is created, we want to add “Halo: Combat Evolved” to our list. We search through the category using a column. In this example, we choose to search “By name”. The results are brought up. We choose to search by items not in the list. The results include the “Halo: Combat Evolved” item. After selecting the item, we simply click “Add” to attach it to the list. Three pictures are displayed explaining this step:


SmartList List
Not secure
SmartList List
Site Links
Resume
Source
SmartList
Profile
Logout
Welcome back Brandon
SmartList List: You can search through a list that exists. Select a filter to sort the items attached to this list. You can also add or edit a column for the list.
Video Games Owned
Items in My list: 1
Delete
Add a new column for the list
Filter by Digit
By year
Filter by Object
Genre
Filter by String
By name
9:03 AM
4/26/2017

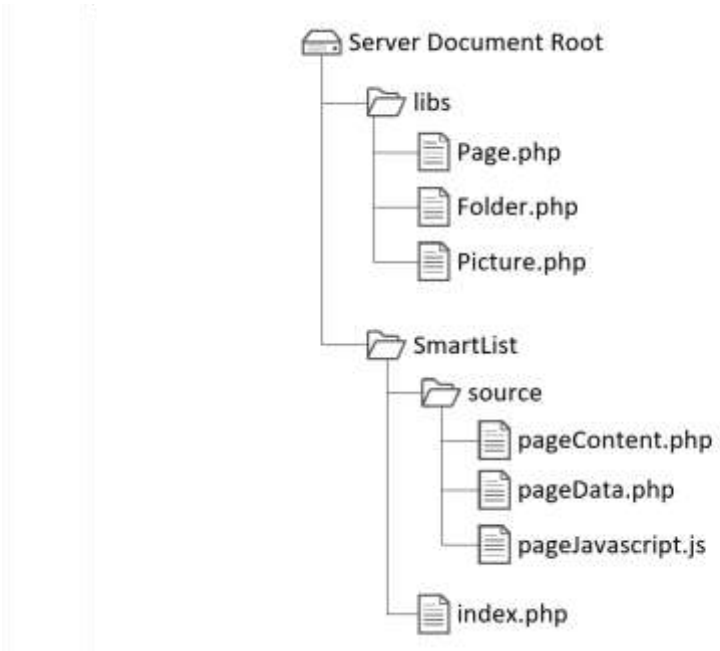

SmartList Search
Not secure
SmartList Search
Site Links
Resume
Source
SmartList
Profile
Logout
Welcome back Brandon
SmartList Search: You can view your search results here. If you selected a list previously, you have the option to view items that are in your profile or items not in your profile. If no list was selected, it just shows all items from the search. You can also add a new group if the search does not yield the desired results.
Search through this list for a particular item
Search
Use this filter to look for items in My list
Name: Halo: Combat Evolved
8:58 AM
4/26/2017



The item can be removed from the list in the same way. They click on the "Remove" button which replaces the "Add" button on the picture above. One final feature of SmartList is the option for category owners to verify changes to items. Since any user can change column values for an item, these changes are not permanent. The category owner has the option to verify all changes. This concludes the demonstration.

Framework Basics

SmartList is designed to work with an already-created framework so that development is quicker. Some of the functions are integrated with the framework, including file management procedures, so it is imperative to explain how this works at a high-level. For a page to work with my framework, multiple files need to be placed in specific folders. At the document root of the server, the framework files reside in a *libs* folder. In addition to those core framework files, all websites must include these files in its directory: *index.php*, *pageContent.php*, and *pageData.php*. The index file just includes *Page.php* to designate its directory as a web page. *pageContent.php* includes a class that contains all of the view functions from the MVC architecture. *pageData.php* includes a class that contains all of the controller functions from the MVC architecture. They are both instantiated by the Page object and then perform their operations. These two files, *pageContent.php* and *pageData.php*, must be placed inside of the *source* folder (Located inside the directory that *index.php* resides in) for everything to work. Additionally, *pageJavascript.js* is placed inside of the source folder as well so that it is automatically included for the page. A diagram of this structure is provided below to allow for a request on "/SmartList/" to work.



The beauty of the framework is that the files are automatically included and executed. One interesting function in the Page object is *getFileInDirectory()*, which handles looking for those required files in the *source* folder. As an example, it can process a *pageContent.php* and *pageContent.html* file. The function looks for the filename that satisfies the extension whitelist and then performs operations on that file. It includes the PHP file, attaches the JS/CSS file, or prints out the HTML file. The method also uses recursion so that nested website data can be processed as well. SmartList, however, does not utilize this feature internally. The code for the method is below:

```php
// Get the component of the page from the file
public function getFileInDirectory($origFileName, $serverFromDirectoryPath, $serverToDirectoryPath, $pageData,
$objArray){
    // Check for file in current directory
    list($fileName, $fileExtension) = $this->getFileNameExtensionInDirectory($origFileName,
    $serverFromDirectoryPath);
    if($fileExtension != ""){
        $serverFilePath = $this->makeFilePath($serverFromDirectoryPath, $fileName, $fileExtension);
        // Html files are printed directly as html code
        if($fileExtension == "html")
            echo "<div class='container-fluid'>".file_get_contents($serverFilePath)."</div>";
        // Php files are represented as objects and are pushed into arrays
        else if($fileExtension == "php"){
            include $serverFilePath;
            $classConstructor = "$className";
            // Push an object to the array that is initialized under the class definition (And carrying the
            pageData as well as previous objects found in the recursion)
            array_push($objArray, new $classConstructor($pageData, $objArray));
        }
        // Js files are printed as links
        else if($fileExtension == "js")
            $this->printHeaderJavascript($this->getFilePathClient($serverFilePath));
        // Css files are printed as links
        else if($fileExtension == "css")
            $this->printHeaderCss($this->getFilePathClient($serverFilePath));
        else
            array_push($objArray, $serverFilePath);
    }
    // Check for the next file if the path is not yet the destination
    if($serverFromDirectoryPath != $serverToDirectoryPath)
        return $this->getFileInDirectory($origFileName, $this->getNextDirectory($serverFromDirectoryPath,
        $serverToDirectoryPath), $serverToDirectoryPath, $pageData, $objArray);
    return $objArray;
}
```

One more flexible concept used by the framework is generating the response. The response from the server is dependent upon the “response” GET parameter in the request. The response types used by SmartList include “source” and “json” (as well as a blank value). In cases where $_GET[“response”] = “” (or it is not set), the response is an HTML document generated from the pageContent object. Normal page visits will utilize this response type. In cases where $_GET[“response”] = “source”, it will again respond with an html document containing source code for the project. The Page object just prints out the source code for every file residing inside of the *source* folder. Finally, in cases where $_GET[“response”] = “json”, it will respond with a JSON object. When this happens, a specified array generated from the pageData object will be encoded in the response as that JSON object. This response type is used for error notifications and page redirects within SmartList.

The website is much easier to maintain by using the framework. The Page object automatically creates nested navigation links, a header, and footer (if the request allows for it). Other notable functions include handling picture management and presentation, which will be discussed in more detail in the picture section.

Development

In the past, I have completed two other large-scale projects, *Monopoly* and *CoasterChecklist*. *Monopoly* was a multiplayer web game replicating the original board game.

CoasterChecklist was a very limited website similar to this project. These prior projects were grueling to complete because I never thought that the development process was important. I learned that I was very wrong with that assumption.

When making *Monopoly*, I would start coding and never think of the design. This was a mistake because I needed to make many drastic changes. I initially used the filesystem to save the game and then, after completing the code, decided to use a database to save the game. I also initially used GET requests to play the game and then, after completing the code, decided to use POST requests and AJAX. When I started *CoasterChecklist*, I put a little more effort to the initial design but the same problem happened. With *SmartList*, I poured a heavy amount of effort into the design phase. I thought intensely about the solution I wanted to develop. This worked to my advantage because I encountered little trouble during development. Design is very important in the development process because it minimizes the number of errors and changes that have to be accounted for.

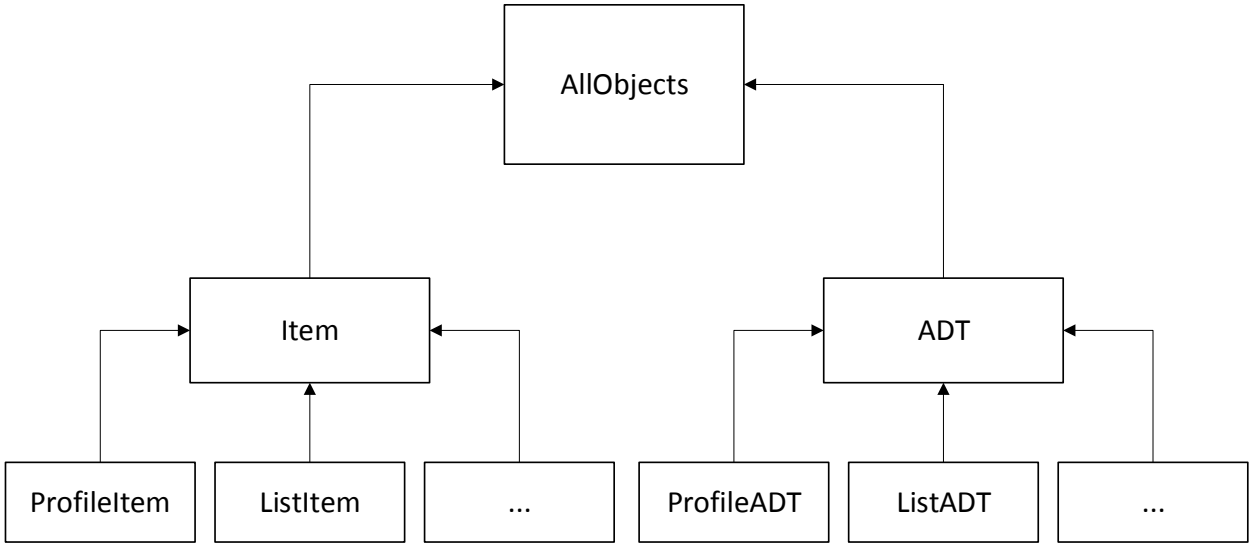
Another lesson learned was the importance of separating the development from deployment. When making *Monopoly*, I developed on the same code that was available to the public. This allowed for immediate updates however, the game broke frequently. When I started *CoasterChecklist*, I tried to separate the development code from the working code by calling different methods within the same file. This was very difficult to maintain and it still caused problems. With *SmartList*, I completely separated development from deployment and it worked very well. I was able to make changes on my development copy, which created no effect on the stable copy. Because of this sandboxing, it was much easier to maintain my code.

One more important part of the development was testing the code. When making *Monopoly* and *CoasterChecklist*, I did not use any type of automatic testing. I would have to manually test everything when I made a change to the code. With *SmartList*, I started unit testing with PHPUnit. This made the development much quicker and more efficient. Instead of trying to manually test, I would just run the unit tests. The PHP unit tester is much better than I am at finding errors. Good unit testing is very beneficial to the developer. I caught many security holes in my code which could have been detrimental. I would have never found them without unit testing.

Class Structure

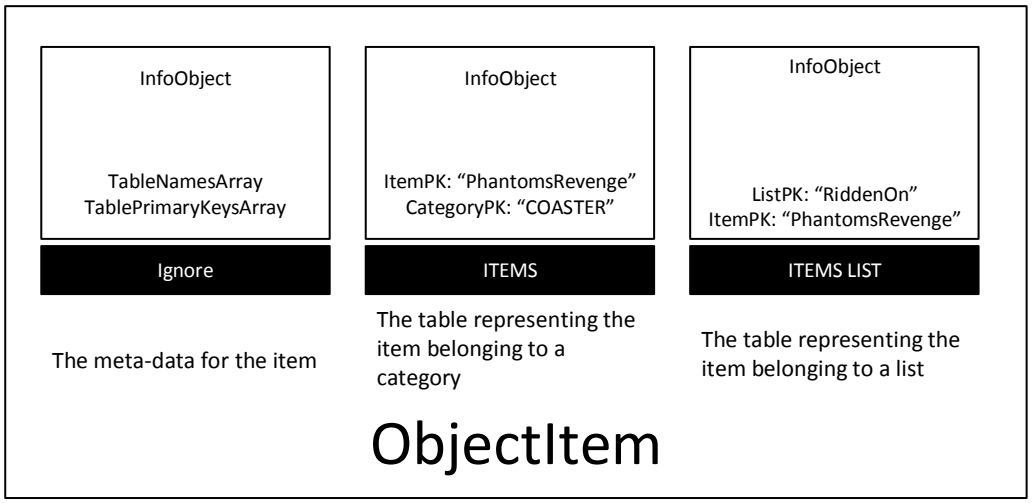
The structure was designed with a high level of abstraction in-mind. The *AllObjects* class provides basic functionality, such as the ability to perform SQL queries. The next class is *ADT*, which extends *AllObjects*. *ADT* can be described as a set of operations on some table. One example function in *ADT* is getting rows from the table. The *Item* class also extends *AllObjects*. *Item* can be described as a row that exists in the table. One example function in *Item* is getting information about a table row. Children of these two objects represent a particular table or item,

such as the case with *ProfileADT* and *ProfileItem*. A visual description is displayed below of the inheritance diagram:



This usage of abstraction and inheritance is very important because it tidies up the code. There was no need to create a method to retrieve items from the table for each ADT. Instead, one method was written and placed in the *ADT* class and all children receive that method. Using abstraction allows for better flexibility and easier maintenance of code.

Data is stored in each of the above objects through an array of *Info* objects. The *Info* class contains variables that can be set through the use of several functions. Typically, an *Info* object represents a table in the database. One notable exception is the "ignore" *Info* object, which contains meta-data. This design allows for any *Item* to contain rows from different tables. It also allows for structural changes of the database. This flexibility was very beneficial when the structure was changed halfway through development. A visual example of an *Item::ObjectItem* design is pictured below, where it links to the *ITEMS* and *ITEMS LIST* tables:



POST Requests

POST requests are used in SmartList to perform operations on the server, including profile logins and item additions. In JavaScript, the *sendOperation()* method is used to send each POST request to the server. All checks are done on the server-side because it allows for the most security. Regex checking and character escaping are some of the ways to prevent security breaches. After the request is sent and the JSON response is received, the browser will use JavaScript to either redirect or show an error message.

In *sendOperation(),* the JSON response will contain information about the new GET parameters for the redirect. Using jQuery, the method then extracts each property and builds the new URL to redirect to. This new URL is a refresh of the current page but with new or changed parameters. If one of the parameters in the JSON object is labeled “errorMsg”, then the controller detected an error in the operation and will not redirect. Various errors include incorrect form input or login credentials. When the “errorMsg” key is detected, its property value is displayed in an error message. This allows the user to change the information before re-trying the operation. The code for *sendOperation()* is below:

```
// Send a post request and then redirect to another page using the JSON response or show an error message
function sendOperation(getParameters, postParameters){
    var postURL = "/SmartList/?response=JSON" + (getParameters != "" ? "&" + getParameters : "");
    $.post(postURL, postParameters, function(data){
        var newURL = "/SmartList/?";
        var newParameters = "";
        var isError = false;
        // Get the new headers for the redirect
        $.each(data, function(propertyName, propertyValue){
            if(newParameters != "")
                newParameters = newParameters + "&";
            // Show the error message if the website detected an error in the request
            if(propertyName == "errorMsg"){
                isError = true;
                // Show the error message on the page
                errorShow(propertyValue);
                // Break out of the each loop
                return false;
            }
            else
                newParameters = newParameters + propertyName + "=" + propertyValue;
        });
        // Reload the page if the request was successful and did not generate an error
        if(!isError){
                window.location = newURL + newParameters;
        }
    });
}
```

Profile

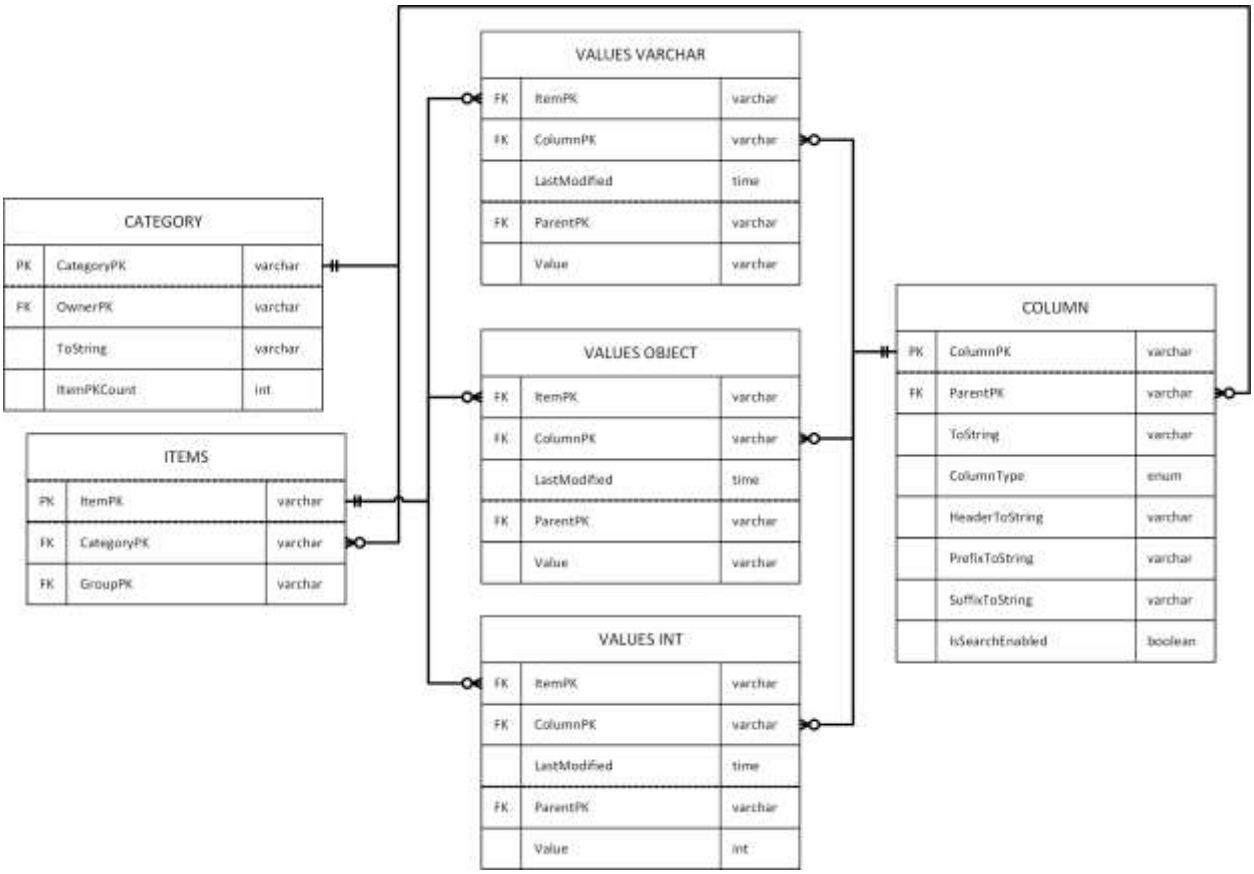
Profiles are used so that people can keep track of their own collections. On the website, a person can create a profile directly from the website. They supply information such as their name, email, birthday, phone number, display name, user name, and password. To ensure security on the server, the password is hashed using BCrypt instead of being stored plaintext in the database. This information is also verified on the server-side using regular expressions and returns an error if they fail the checks. Users who are logged in can also change their information. Much of this code is shared with creating a profile so that it is easier to maintain.

Logins need to be secure so several steps were taken. The website can only be viewed using an https connection. This ensures that the client's POST message containing the password is not exposed to data snooping. Once the password is verified, the profile primary key from the database is set as a PHP session property. The controller uses this session property to detect the user.

Database

The database was initially designed so that every category and group had its own table. Although efficient, the previous design posed many risks and problems. For this reason, refactoring was used to re-construct the database structure. Every item and group belongs in the *ITEMS* table, where the primary id is "ItemPK". The values of each item belong to *VALUES INT*, *VALUES OBJECT*, and *VALUES VARCHAR*. The separation of these tables allows for an

item to have multiple data types for its values. The composite key of these tables consists of "ItemPK", "ColumnPK", and "LastModified". "LastModified" is required for editing and verification purposes. The database diagram for the item structure is displayed below:



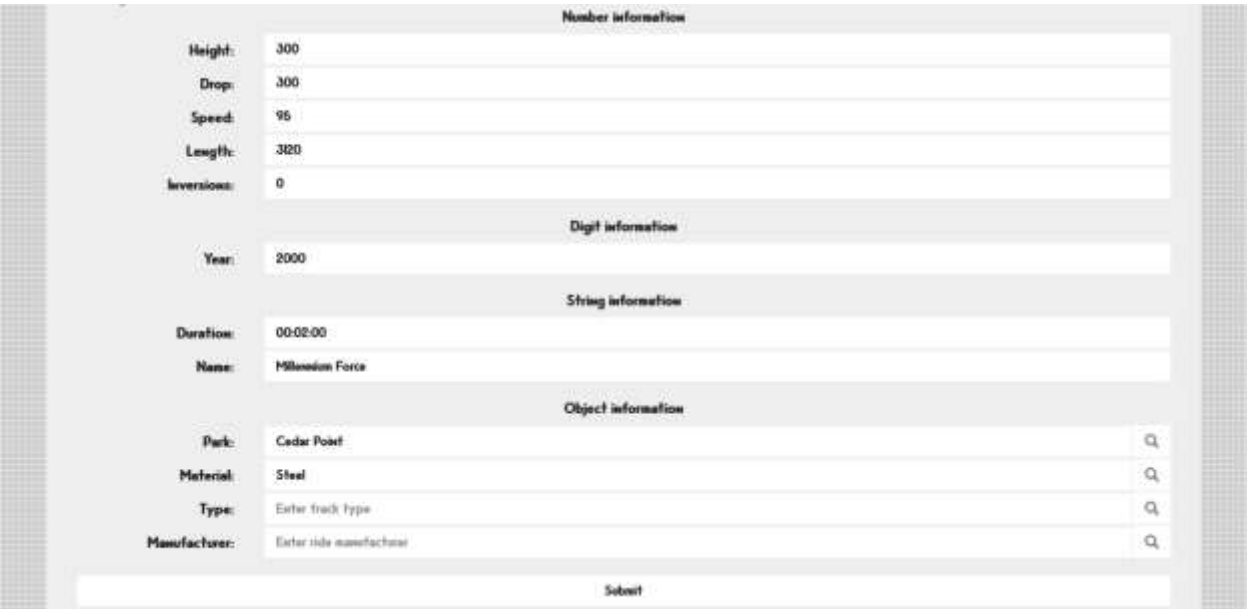
When categories are created, an entry is inserted into *CATEGORY*. The "OwnerPK" is set to the "ProfilePK" of the user. The "ItemPKCount" is set to zero. Categories can contain multiple columns. Columns were implemented in the database by using a separate *COLUMN* table. When a column is added to a category, the "ParentPK" is set to the "CategoryPK". All other information is supplied by the user. Group columns exist in *COLUMN* as well. They have the same information except the "ParentPK" is set to another value. In this case, the "ParentPK" is set to "ColumnPK".

When a user wants to add an item to a category, they type information into a form. The fields of the form are defined by the category administrator. It may also include columns that are personal for the list, if one was specified. These fields exist in the *COLUMN* table as rows. The SQL query retrieves these rows, groups them by their data type, and then generates the form. An

example of the form for a category is presented below:

The fields in the form are grouped together because it allows for better efficiency. The presentation is much better with this grouping as well. Finally, the user knows what values should be in the fields by seeing the grouped fields. When a user clicks the submit button, a row is inserted into the *ITEMS* table. This represents a new item in the category. Additionally, the values are set for the item by inserting new rows into *VALUES INT* and *VALUES VARCHAR.* The insertions have some properties set so that verification can be implemented however that will be discussed in another section.

Any user can edit items in the category. When an item is selected and the “Edit” button is clicked, the same form appears. This time, however, it is filled in with previous information. Any of the fields can be changed and when the Submit button is clicked, its information is changed in the database. An example of the form used for editing an item is presented below:



Object information is discussed below

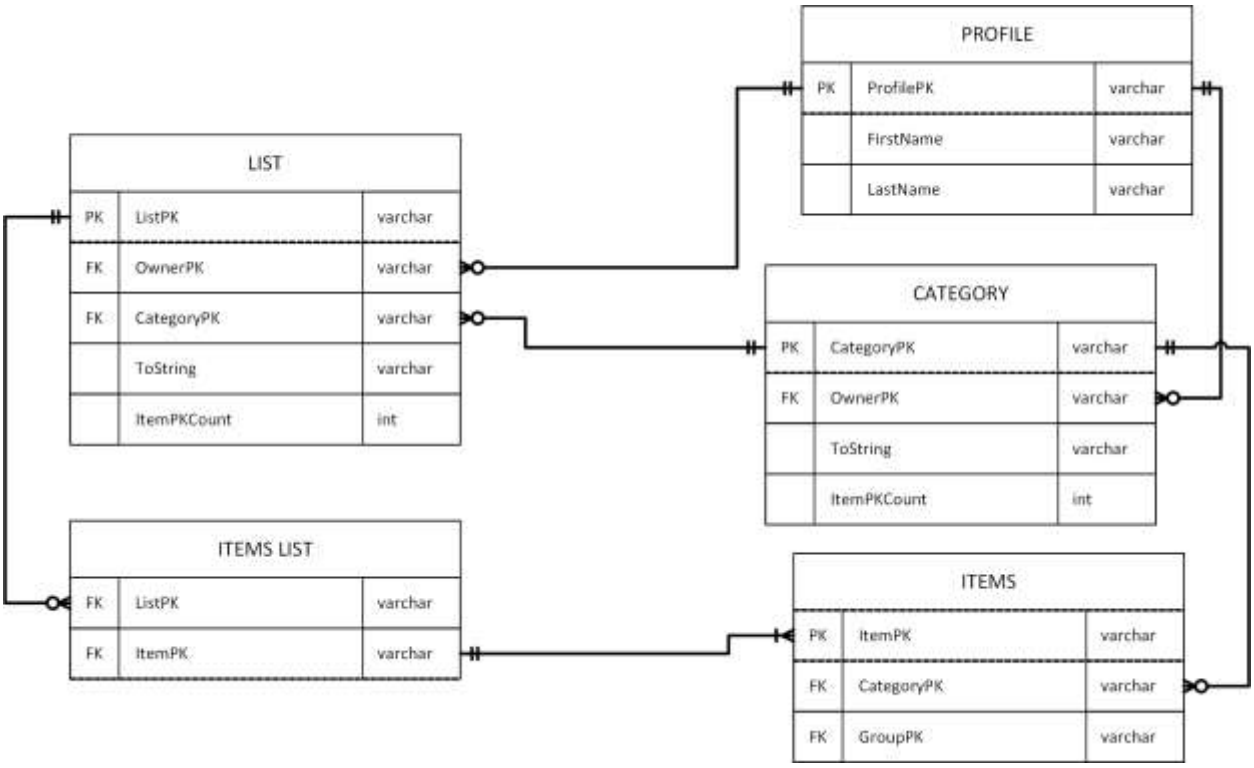
When the submit button is clicked, JavaScript is used to determine if a field was changed from its previous value. An SQL query is then processed to update item information on these changed fields. Rather than updating the rows in *VALUES INT* or *VALUES VARCHAR*, a new row is inserted with additional properties set (to allow for verification).

There is one type of item information available on the form not introduced yet. An item may belong to many groups. For this purpose, "Object Information" fields can be set for items as well. Adding/removing an item is achieved by using the search function to find groups. Once a group is selected, it can be attached to the item. If the group does not exist, a new group can be created. The implementation is similar to item creation/editing. Groups belong in the *ITEMS* table but have their "GroupPK" property set to the value of "ColumnPK" instead of NULL.

Lists

Profiles can create any number of lists for a category. As an example, a user may want to keep track of the books that they have read or want to read. The user selects the option to create a list and specifies a list name. Upon clicking submit, the list is created for that user and they can then add items to it.

The *LIST* table contains references to all of the lists that were created. *LIST* is utilized for simple tasks such as getting the name or number of items in a given list. Additionally, there is an *ITEMS LIST* table that represents items belonging to some list. The composite key of *ITEMS LIST* consists of "ItemPK" and "ListPK". This table contains every item that is attached to a list. The database diagram of the list relationships is displayed below:



A user simply selects the "Add" button, on an item page, to add that item to the list. To remove an item from their list, they simply select the "Remove" button. When a user adds an item to their list, a new entry is inserted into *ITEMS LIST*. Additionally, the "ItemPKCount" attribute is incremented for the row in *LIST*. When an item is deleted from a list, the opposite occurs. When a list is deleted from a category, all of the entries for that list are removed from *ITEMS LIST* and it is removed from *LIST*.

Columns can be created solely for lists. They operate much like a category column with the exception that they are only displayed for a list. To handle this requirement, the "ParentPK" of the column is set to the "ListPK". With list columns, there is no need for verification. To handle this requirement, the "LastModified" of the value item is set to the default value and the value is just updated instead of inserted.

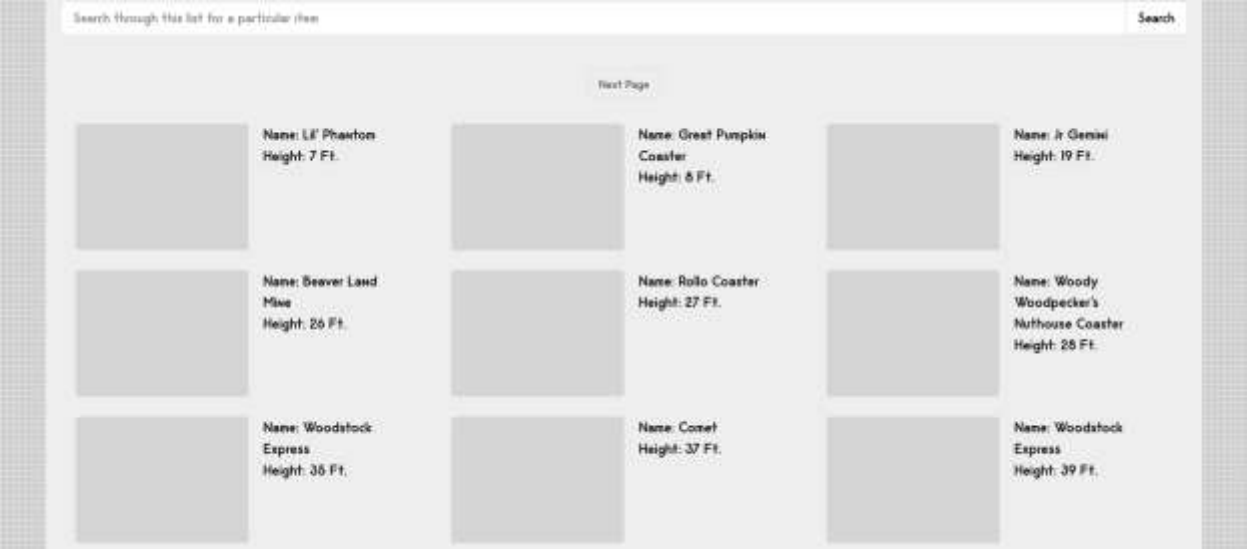
Searching and Viewing

The strongest feature of the web application is the extremely powerful searching that can be performed. Every column in the table can be used as a filter. The minimum requirement is for a column filter and category to be selected. Results can also be limited to items belonging to a list. Alternatively, the results can be limited to items not belonging to the list. It can also filter by groups and then allow for nested searching within a group. Finally, a substring search through the results can also be made. Any combination of these filters can be used. An example of the many filters supported by the function is described in the function call below:

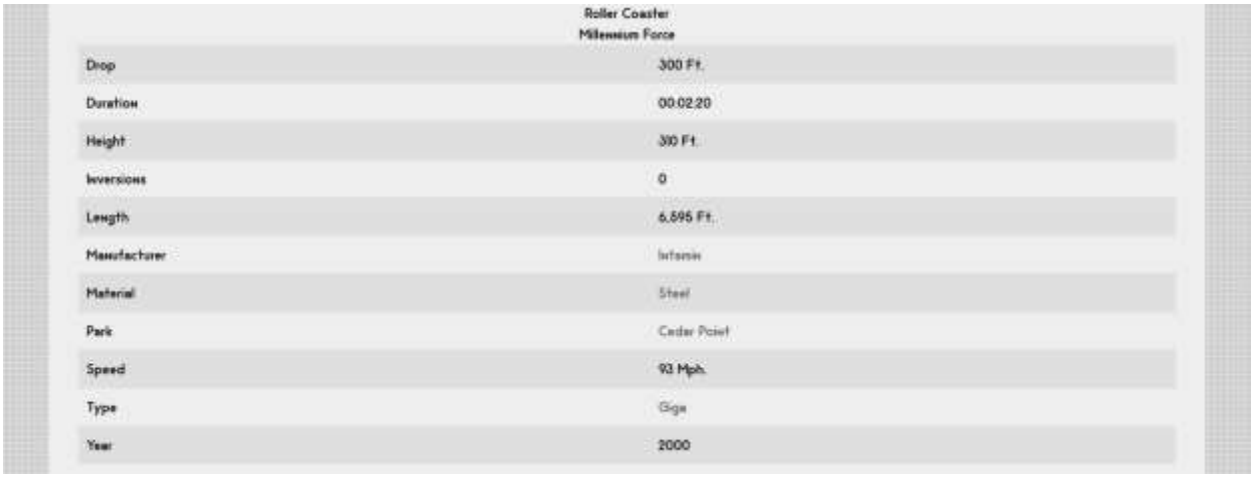
```php
// Get the items for the search page using a variety of options
public function getItemsForSearch(&$categoryInfoObject, &$groupInfoObject, &$listInfoObject, &
$columnInfoObject, $isASC, $startAt, $offset, $isDiscovery, $searchString){
    $SQLQuery = $this->getItemsForSearchQuery($categoryInfoObject, $groupInfoObject, $listInfoObject,
    $columnInfoObject, $isASC, $startAt, $offset, $isDiscovery, $searchString);
    return $this->getItems($SQLQuery, $this->getTableName("main"));
}
```

Security is a major concern when crafting SQL queries because of SQL injection. A malicious user could attempt to alter information in the database or retrieve private information. For this reason, all SQL values are properly escaped.

The result size can be very large with searches on large tables. A limit is enforced on each query and a paging function is implemented. There are “Start” and “Next Page” links for each result page. A unique function was used to calculate if there was a next page. The SQL query implements a limit equal to the visible limit + 1. PHP checks if the result size is greater than that visible limit. This PHP function can decide if there is a next page without retrieving all of the items from the search. An example of a search page is presented below:



When an item is selected from the search, the next page shows all of the item’s information. All of its information is gathered through the *COLUMN*, *VALUES INT*, *VALUES OBJECT*, and *VALUES VARCHAR* tables. These SQL queries are loaded into *Info* objects belonging to the *Item* object. The item values are gathered from *VALUES INT*, *VALUES OBJECT*, and *VALUES VARCHAR* and they retrieve the row with the latest “LastModified” value for each “ColumnPK”. The user sees the most recent value for each column, whether it is verified or not. The column header, prefix, suffix, and type are gathered from *COLUMN* and allow for a way to format the presentation of the value. An example of the item page with information is pictured below:



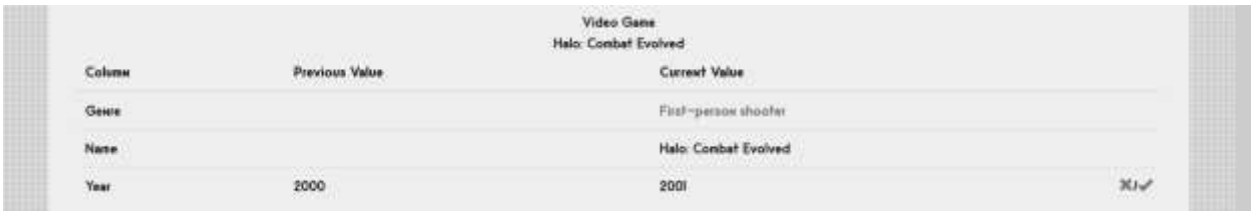
Roller Coaster
Millennium Force

Drop	300 Ft.
Duration	00:02:20
Height	310 Ft.
Inversions	0
Length	6,595 Ft.
Manufacturer	Intamin
Material	Steel
Park	Cedar Point
Speed	93 Mph.
Type	Giga
Year	2000

Verification

Verification was created to allow for the owner of a category to filter out information that is not valid. Verification is possible because all changes to values are insertions instead of updates in the *VALUE* tables. The initial value for an item is set to NULL and "LastModified" is set to "2000-01-01 00:00:00". This allows for a default value.

Verification works by showing all items/groups that have more than one change on a column. The owner of the category is allowed to verify changes. They can see the history of changes. To accept or reject a change, they simply select the check (To accept the current value) or cross (To reject the latest changes). A picture is displayed below of the verification process:



Video Game
Halo: Combat Evolved

Column	Previous Value	Current Value	
Genre		First-person shooter	
Name		Halo: Combat Evolved	
Year	2000	2001	✗✓

When a value is accepted, all other values for that column are deleted from the table. Similarly, when a value is rejected, the same process happens. However, instead of keeping the latest value, the oldest value is preserved. Verification works very well with the case of adding a value that has not been set previously. With new items, all of their column values are initially set to NULL. This ensures that there will always be at least two entries when a value is changed. An example of changes to a column value is displayed below:

Future Additions

Despite including many features in the project, I still have many features to add. I will continue working on this project in the future. One possible addition includes adding picture support for items and groups. I could include pictures, whether public or private, with an item. I can also add picture verification for the category owner. Another possible addition is including machine learning for recommended items. I can suggest new items to add to a list with this feature. Another possible addition is creating web crawlers to create items from previously-existing database web applications. Many of these databases exist, such as IMDB for movies or RCDB for roller coasters. Another addition is allowing a better way for category owners to customize category presentation. This feature seems more useful with the addition of pictures. One final addition is creating a more efficient structure. My current design works but it can become very slow due to the large size of the tables.