Gaming Laboratories International

Richard Figueroa Erickson, Benjamin Fraser,
Carson Sharpless, Richard Thompson
June 16th, 2019

Introduction:

Gaming Laboratories International(GLI) is one of the foremost testers of gaming machines in their industry. These gaming machines, more commonly known as gambling machines due to the nature of the casino business, must be tested thoroughly. GLI receives software, hardware, and firmware for potentially new machines from a manufacturer and test against multiple criteria. They test whether or not the machine's paytables and features perform properly, as well as living up to a set of rules in the jurisdiction in which it will be operating. For example, Las Vegas casinos have different policies enforced upon their games than in Oklahoma or Colorado. GLI tests for 475 jurisdictions worldwide. For each machine, the testing engineers must develop a test to ensure that the paytables and features act as advertised. This causes multiple developmental issues as individually creating a test for each machine slows down the overall process.

GLI currently must search through emails and files to find similar machines that have been tested. But first individuals must remember information about the machine they are looking for in order to be successful. As a result, the GLI has seen the need for a way to search back through their past machines in a quick and efficient way. Currently, when an employee is looking to see if they have tested out a machine that has similar features and paytables, they have to do it based off of memory. They must try to locate the file by searching through files and past tests. GLI is lacking in an easy, quick way to search through there previously tested machines. As a result, they have tasked us to make them something that would solve this issue.

The project GLI gave to our group was to make a modular database which connects to a web application. They would like to be able to edit their search to have as wide or narrow specifications as they would like. GLI has asked that both a user and admin will be accessing this page. The user is limited to only searching the page, while the admin can add/remove different fields, users, as well as saving a backup of the database file. Our group’s solution is made up of a backend database which connects to a C# framework, to then communicate with the front-end Vue.js javascript web application.

Requirements:

The database itself is hosted on Microsoft SQL Server 2008. The front-end GUI application runs in a web browser, while the back end runs on their server. Additionally, GLI created a git repository to contain all the code and documentation.

Functional Requirements:

- 3 parts
    - Front end
        - GUI
        - Variant for user and admin
    - Database
    - Back end
        - Actual logic, controls differences in permissions
- Front end has several types of search fields
    - Drop Down
    - Text
        - Keyword
        - Exact
        - Wildcard
            - Can have multiple wildcards
- User and Admin permissions based on Windows login info
    - User can only search database or export search results to text/excel
    - Admin can modify database
        - Add new dropdown or text field
        - Add data to existing dropdown or text field
        - Removing a dropdown or text field
        - Removing data from existing dropdown or text field
        - Add an entirely new entry to the database
        - Removing an entry from the database
        - Make it easy to add/remove people and change permissions
- Base fields needed
    - File number (text w/ wildcards) - Primary key
    - Manufacturer (dropdown)
    - Name of theme (text with wildcards)
    - Component ID (text with wildcards)
    - Progressive information (dropdowns)
        - WAP vs Linked vs Standalone
        - Inc. rate and startup value

                - Hard coded or configurable
            - Number of levels
            - Can it be disabled
        - Different features (dropdown w/ multiple selection)
            - Pick bonuses
            - Theme related
            - Free games
        - Modifications (Text w/ wildcards, enterprise search?)
        - Features/themes waived by NV(Text w/ wildcard)
            - JIRA ID
        - Machines required (dropdown)
        - Number of paytables (Exact text)
            - Bet type (Dropdown)
                - Multiplier
                - But a pay
            - Game Type (Dropdown)
                - Slot
                - Keno
                - Poker
                - Etc.
        - Paytable ID (text w/ wildcard)

Non-Functional Requirements:

- Developed for windows
- Front end is web based
    - Javascript w/ Vue.js framework
    - Runnable on chrome or IE
- Back end is in C#
    - Entity framework
- Database uses Microsoft SQL Server
- Exported file has consistent formatting
- Documentation and instruction manual
    - Code development
    - “User” use
    - “Admin” use
- App properly runs on their environment
- Git repository containing all the code and documentation

System Architecture:

The main brunt of the design was spent on the back-end of our program in order to create a modular database. The database is able to have fields which can be added or removed; which, was a key point in what GLI wanted in the final product. Below in tabular format is the list of our database fields with their corresponding parent entity.

Parent_entities table:

name	submission_relation
submission	self
progressive	n_to_m
paytable	n_to_m
nevada_waiver	n_to_m

Fields table:

name	lookup_type	field_type	parent_entity
file_id	exact_text	attribute	submission
manufacturer	single_dropdown	fkey_parent	submission
theme	wildcard_text	xref	submission
component	wildcard_text	xref	submission
progressive_type	single_dropdown	fkey_parent	progressive
increment_rate	single_dropdown	fkey_parent	progressive
startup_value	single_dropdown	fkey_parent	progressive
number of levels	exact_text	attribute	progressive
can_disable	single_dropdown	fkey_parent	progressive
feature	multiple_dropdown	xref	submission
modification	wildcard_text	xref	submission
jira_id	wildcard_text	attribute	nevada_waiver

waived_feature	wildcard_text	attribute	nevada_waiver
machine	single_dropdown	xref	submission
bet_type	single_dropdown	fkey_parent	paytable
paytable_id	wildcard_text	attribute	paytable
game_type	single_dropdown	fkey_parent	paytable

This allows the front end to easily give data that can generate a search query into our database table. The front end had two main aspects of design. Together with GLI, we decided on a separate admin and user page. The main thought behind the design of these pages was twofold. The application needed to be primarily user friendly as well as being nice to look at, due to view the page for an extended period of time. Below are images of what both the User and Admin page look like for our web application.

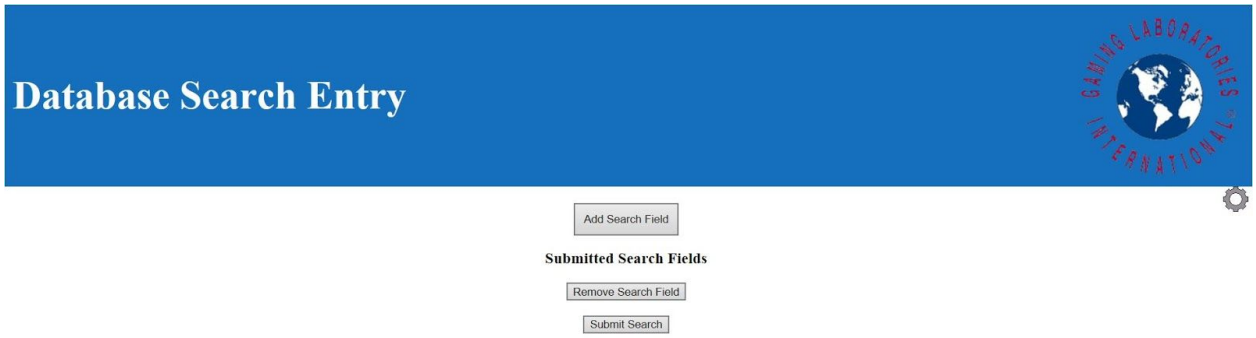

Figure 1: The user GUI

Figure 2: The admin GUI

As shown above, the buttons and options for both the user an admin are very straightforward to use. We focused on making clear buttons and search fields so it would be easy for anyone to pick up to use.

Technical Design:

The most difficult technical aspect of the project was the modular database design that needed to be made to fit all of GLI's requirements. Since fields are able to be added and removed, this required the back-end to be able to procedurally generate search queries without running into errors or crashing the program. As a result, we based our design around two metadata tables, being "Parent Entity" and "field".

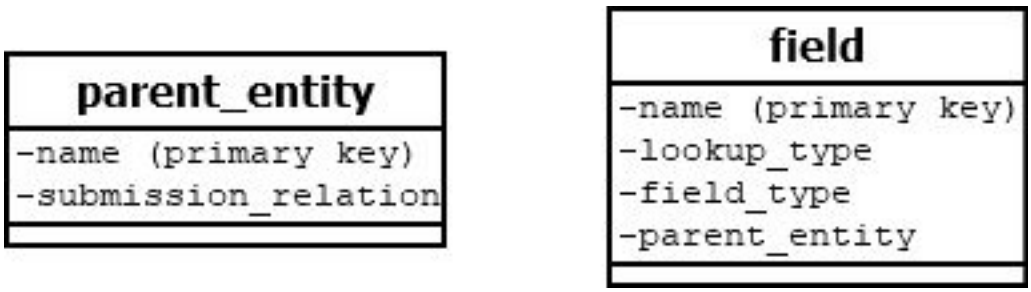

Figure 3: Metadata Tables

"Parent Entity" in this context refers to any object modeled in the database which has more than one attribute that is relevant to the front end and is not a foreign key. A parent entity may have multiple fields associated with it. The entities given in the slides are submissions,

progressives, and paytables. The database begins with only the submission entity, which has file_id as its primary key.

The parent_entities table has the following attributes:

- name (primary key)
- submission_relation

Submission relation has the following possible values: self, one_to_one, one_to_n, n_to_one, n_to_m

The 'parent_entities' table starts with a single row having name = "submission", submission_relation = "self".

To add a new entity, an admin must enter the following information:

Name of the entity to add, as a singular instance of the entity. i.e, 'progressive'
Can a single instance of this entity be associated with more than one submission?
Can a single submission be associated with more than one instance of this entity?

Depending on the relationship between submission and the new entity, one or more new tables are created.

- 1 to 1: Create a new table '<name>s', with id attribute as primary key, and foreign key to submissions.file_id.
  Add attribute '<name>_id' to 'submissions' table which is foreign key to the primary key of the new entity.
- 1 to N: Create new table '<name>s', with id attribute as primary key, and foreign key to submissions.file_id.
  Create count field for new entity.
- N to 1: Create a new table '<name>s', with id attribute as primary key.
  Add attribute '<name>_id' to 'submissions' table which is foreign key to the primary key of the new entity.
- N to M: Create new table '<name>s', with id attribute as primary key and value attribute.
  Create new table '<name>_submission_xref' with foreign keys to submissions.file_id and primary key of new entity, letting both foreign keys be a combined primary key.
  Create count field for new entity.

The primary keys of all parent entities, except the submissions, will be generated by the database and never used by the front end.

Fields:

Begin with 'fields' table having the following attributes:

- name (primary key)
- lookup_type
- field_type
- parent_entity (foreign key on parent_entity.name)

Add row to 'fields' with name = 'file_id', lookup_type = 'exact_text', field_type = 'attribute'.

To add new search field, an admin must enter the following information:

Name of the field to add, as a singular instance of the field. i.e, 'progressive'

Lookup type of field: exact_text, wildcard_text, single_dropdown, multiple_dropdown, count

What is the parent entity of the field?

Can a single instance of this field be associated with more than one instance of the parent entity?

Can a single instance of the parent entity be associated with more than one instance of this field?

The new field is then added to the 'fields' table.

Each of the attributes is defined based on the parameters provided by the admin

name: the name provided, formatted if necessary

lookup_type: the provided lookup type

field_type: attribute, fkey_parent, fkey_other, xref, or count, determined by the relationship between the parent entity and new field

- 1 to 1: attribute
- 1 to N: fkey_other
- N to 1: fkey_parent
- N to M: xref

parent_entity: the provided parent entity name

Then, update the database immediately depending on lookup_type

- attribute:
  add attribute <name> to '<parent_entity>s' table.

- fkey_other:
  Create new table '<name>s', with id attribute as primary key, value attribute, and foreign key to parent entity primary id.
- fkey_parent:
  Create new table '<name>s', with id attribute as primary key and value attribute.
  Add attribute '<name>_id' to '<parent_entity>s' table which is foreign key to the primary key of the new entity.
- xref:
  Create new table '<name>s', with id attribute as primary key and value attribute.
  Create new table '<name>_<parent_entity>_xref' with foreign keys to the primary keys of the new and parent entities, letting both foreign keys be a combined primary key.
- Count:
  nothing

The primary keys of all fields is generated by the database and never used by the front end.

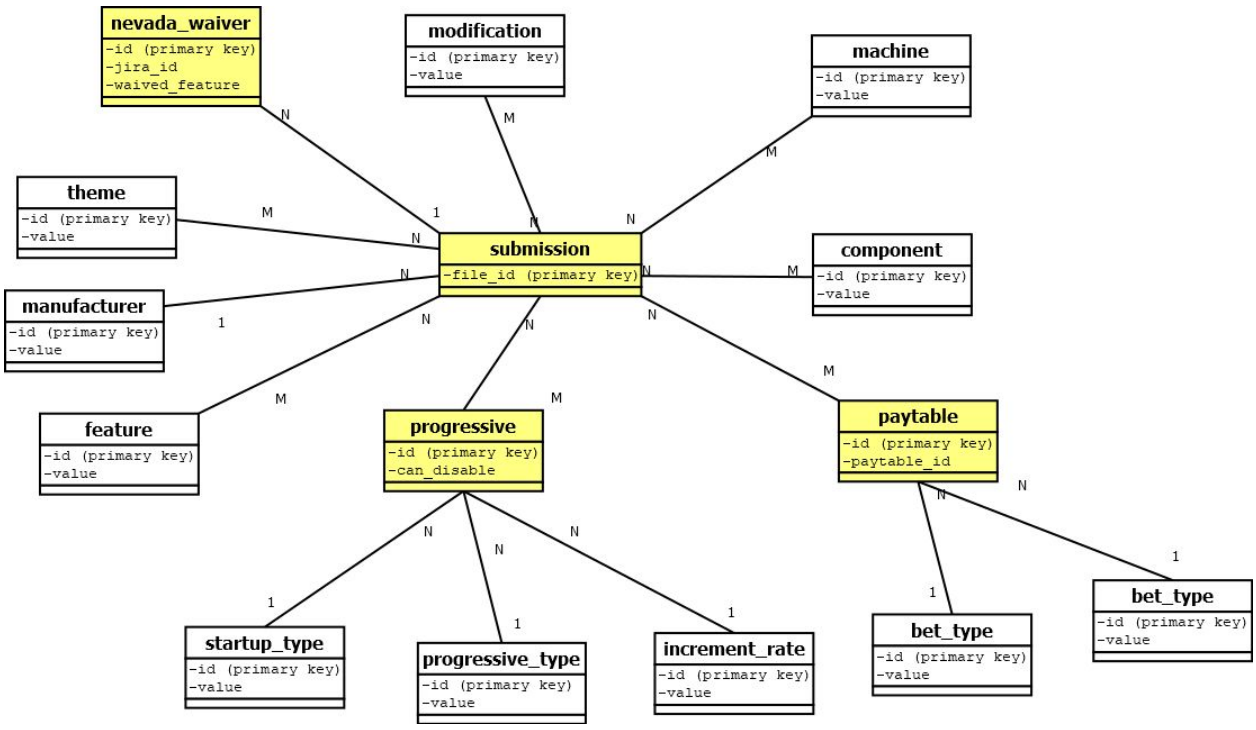

Figure 4: Database relationship diagram

Adding Data:

To add the data from a single submission to the database, all parent entities and fields must have already been created as described above. After that, the user can manually enter data by creating instances of entities.

A user can create an instance of a parent entity, which requires them to enter all information of the fields of that entity. For text fields, the user enters the data as text. For dropdown fields, the user can either select an existing option from a dropdown menu or enter a new one as text. When a user creates an instance of any parent entity other than a submission, they are also required to specify which submission or submissions are associated with the new instance. (Note - this may require the front-end application to be able to perform searches during the insertion process).

Querying the Database:

GLI has stated that they would like to be able to select a search field on the front-end, and have all data shown for all submissions matching the data entered into the search field. Based on the chosen field and the user input, SQL queries will be generated to return the appropriate information. The following is a general layout of the query generation:

- The query will SELECT the primary id attribute FROM the table of the parent entity of the chosen field, giving a list of all instances of the parent entity which match the user search.
- The WHERE clause has a logical statement based on the lookup type of the field: exact text or single dropdown will be something like "WHERE value = '<input>'. Wildcard text uses "LIKE" rather than "=". Multiple dropdown will use "WHERE value IN <all selected dropdown options>".
- An appropriate join of the field and its parent entity is conducted based on the field type.
- Once the above query has been generated, if the parent entity of the selected field is not the submission entity, another select query with another join will be generated based on the submission_relation of the parent entity, which will return the file ids of all submissions matching the user input.
- Once the list of relevant files is obtained, more select queries can be performed in order to provide meaningful output to the user.
- If the user then chooses to narrow the search with another field, as is specified in GLI's requirements, the above process will be repeated to select submissions matching the new search from the list of submissions returned by the previous search.

Quality Assurance Plan:

The components of our project are split into four layers: The GUI layer, the input sanitizer layer, the SQL generator layer, and the database layer. Our testing involved unit testing on each of the layers individually, as well as integration tests involving all of the layers together.

The unit tests on the GUI layer ensured that the GUI performs user input and output correctly. This layer must be able to read in single text strings from text boxes, numerical values, and single or multiple selections from dropdown menus. The GUI must also, when given an array of values, correctly display the array in a dropdown menu. Finally, when given the information on a database object, it must correctly display this information in a tabular format.

The input sanitizer layer is meant to protect against SQL injection attacks. Its unit tests require it to successfully detect when an input string presents a risk of SQL injection, and to output a string that is as similar to the input as possible but does not pose such a risk.

The SQL generator layer was the most thoroughly tested, as its logic is the most complex. In general, the SQL generators will be passed user inputs, as well as the definitions of search fields and parent entities, and will output statements containing valid SQL syntax as strings. Some of the objects included in this layer are the query generator, field generator, and table reader. Unit tests on the query generator assert that, given a field definition and example data, the query generator returns a SELECT statement which correctly searches for the given data in the given field. Unit tests on the field generator will assert that, given a field definition, the field generator will return a series of SQL statements which add the given field into the database. Unit tests on the table reader assert that, given a .csv file containing information on submission files, the table reader will return a list of SQL statements which correctly insert the data into the database.

The database connection layer executes SQL commands that are passed as strings. Unit tests on this layer assert that the objects can successfully connect to the database and execute queries on it.

Integration tests investigated the functionality of the application as a whole, with all layers working together. To pass integration tests, the user must be able to enter data into the database using a .csv file, and then query the database for that data. They must be able to add a field to the database and then perform a search on the database using that field.

Results:

The goal of our project was to create a database, along with an application allowing users to perform operations on it. The list of functions that needed to be performed by the final project is as follows:

- Differentiate between user and admin
- Perform search on database
- Add entity/field to database (admin only)
- Remove entity/field from database (admin only)
- Input data into database from .csv file (admin only)
- Remove files from database by file id (admin only)
- Backup and restore database (admin only)
- Run database cleanup (admin only)
- Create an enterprise search for the database(stretch goal)

The database itself would consist of two metadata tables, containing the information of the search fields and their related entities. For each search field and entity, additional tables and/or columns would be procedurally generated based on the available metadata. This design would ensure compliance with the client's requirement that the final product be able to add and remove search fields. The primary entity would be the submission, which would represent a single file submitted by a manufacturer. All other entities would have a defined relationship to the submission entity, and all search fields would have a defined relationship to a single entity. The front end, a gui built on Vue.js, would communicate to C# code on the back end, which would execute procedurally generated queries on the database using the Entity framework. The only goal we were not be able to complete was the stretch goal GLI set for us, to integrate an enterprise search within the database. The six week time constraint as long as all of our other goals made us unable to complete this goal.

Lessons Learned

The primary lesson for our team to take was to attempt to understand where we might encounter troubles with our code. While our progress was good throughout most of the six weeks, we were caught off guard by how difficult connecting the front-end and back-end would be to complete. This caused a delay in our program as testing needed to be held until this issue was resolved. Looking back at our design document, some of us expected this to be a small issue, yet this turned into the primary error in our whole program. The main lesson learned was

being able to look at aspects of the project beforehand and understand the repercussions of what could happen if that part of the project goes awry.

Additionally, the overall project help give us a good look into what working with a client while being a software engineer would be like. We had weekly meetings with GLI to update and show off the progress of our project. During these times, feedback was given on how we had designed our project, causing us to explain how and why we did things a certain way. This would result in minor changes or requests that the client would have, which we would add in time for the next meeting. The give and take relationship for working for a client was something none of our group had really ever experienced. It was a good simulation of how the real world does require constant communication between the software team and its clients.