|  |
| --- |
|  |
| Pathfinder Random Encounter Generator |
| Padraic Cashin, David Yambay |

|  |
| --- |
| 4/19/2015 |

**Table of Contents**

1. Project Description
2. Database Implementation
   1. Schema
   2. ER Diagram
3. DDLs (Create Table Constructs)
4. GoLang Code
5. Screenshots and Sample Queries
6. Logs for Future Extensions, Features left out
7. Team Member Responsibility description
8. Lessons Learnt
9. **Project Description**

Pathfinder is a table-top role playing game developed by Paizo, similar to Dungeons and Dragons. Managing random monster encounters is one of the most time consuming, and difficult aspect of running the game, especially if the game master wants to maintain a certain level of variety among the encounters.

A good encounter should have the correct difficulty for the party, should contain monsters that are reasonable based on the current terrain, for example the generator should never return polar bears if the players are currently in an arid desert, and should contain a good mix of different monsters across many encounters. The game allows for this variety by allowing the game master to mix different monsters to generate an encounter up to a difficulty threshold. Some monsters can also have class levels, similar to the players, which could be added if time allows.

Our initial implementation should include a graphical interface that will handle the query creation and displaying of the encounter. The database should contain a table of monsters, a table of monster types, a table of books, and a table of traits common to all monsters. Since monsters share many of same properties, those values could be moved into other tables or relations if it makes sense to do so. For the initial database, it will contain the four tables, and a series of functions that will help us build a valid encounter. The only input from the user should be encounter level, terrain type, and preferred monster type.

The rest of this paper is as follows. Section II discusses the creation of the database itself including the ER diagram and Schema of the database. Section III examines the code to populate the database with the monsters. Section IV showcases a few sample queries and the outputs of the query. Section V and VI cover the GoLang code used the implement the database as well as screenshots of the interface. Section VII contains information regarding changes to the project and future work that would be implemented. Finally the paper concludes with a discussion of what each group member contributed to the project as well as what the group has learnt throughout this project.

Pathfinder Role Playing Game is released under the Open Gaming License (OGL).

1. **Database Implementation**

The database implementation for the Pathfinder Random Encounter Generator revolves around the four tables; Monsters, Types, Book, and Attacks. The largest of these tables is the Monsters table. The monsters table contains all relevant information regarding the monsters that can appear in the world. The type table gives information regarding the how much health specific monsters types have and the attacks table contains a list of different attack types present by monsters and the damage that they do based on the size of the creature

**II.1 Tables**

As described above the schema for this database contains three main tables with two relationships stemming from the central monster table to the other tables in the database. The monsters table has many different attributes that are a part of it and the most pertinent will be described to give a better understanding of what the attribute covers. The Monster table schema is shown below.

-- -----------------------------------------------------

-- Table `PathfinderEncounter`.`Monster`

-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS `PathfinderEncounter`.`Monster` (

`idMonster` INT NOT NULL,

`Name` VARCHAR(45) NOT NULL,

`CR` INT NOT NULL,

`Alignment` VARCHAR(20) NOT NULL,

`Size` INT NOT NULL,

`Class` VARCHAR(25) NULL,

`TypeName` VARCHAR(45) NOT NULL,

`Initiative` INT NOT NULL,

`Armor` INT NULL,

`Shield` INT NULL,

`Deflection` INT NULL,

`SizeAC` INT NULL,

`NaturalArmor` INT NULL,

`Dodge` INT NULL,

`MiscAC` INT NULL,

`HitDie` INT NOT NULL,

`Fort` INT NOT NULL,

`Reflex` INT NOT NULL,

`Will` INT NOT NULL,

`BaseSpeed` INT NOT NULL,

`Space` INT NOT NULL,

`Reach` INT NULL,

`Spell-Like Abilities` VARCHAR(1500) NULL,

`Spells` VARCHAR(1500) NULL,

`Str` INT NULL,

`Dex` INT NULL,

`Con` INT NULL,

`Inte` INT NULL,

`Wis` INT NULL,

`Cha` INT NULL,

`BaseAttack` INT NOT NULL,

`CMB` VARCHAR(45) NULL,

`CMD` VARCHAR(45) NULL,

`Feats` VARCHAR(1500) NULL,

`Skills` VARCHAR(1500) NULL,

`Languages` VARCHAR(200) NULL,

`Special Attacks` VARCHAR(1500) NULL,

`Environment` VARCHAR(45) NOT NULL,

`Attack1` VARCHAR(45) NULL,

`Attack2` VARCHAR(45) NULL,

`Attack3` VARCHAR(45) NULL,

`Attack4` VARCHAR(45) NULL,

`Attack5` VARCHAR(45) NULL,

`Book\_idBook` INT NOT NULL,

PRIMARY KEY (`idMonster`, `Book\_idBook`),

INDEX `fk\_Monster\_Book1\_idx` (`Book\_idBook` ASC),

CONSTRAINT `fk\_Monster\_Book1`

FOREIGN KEY (`Book\_idBook`)

REFERENCES `PathfinderEncounter`.`Book` (`idBook`)

ON DELETE NO ACTION

ON UPDATE NO ACTION)

ENGINE = InnoDB;

A number corresponding to each monster is used as the primary key. While the monster name looks as though it could have been the primary key there are situations where this would not be allowed. The main scenario is that a monster can have additional attributes added to in the form of a class. The class is rarely used for monsters as the main purpose of classes are for player characters, but the addition of a class to a monster changes the monsters statistics and in essence creates a new monster but the name is the same and thus would not be unique in the table.

The most pertinent attributes for the monster are the monster name which gives the user knowledge on the monster, the challenge rating or CR of the monster which determines how tough a monster is to fight, the monster type which determines many attributes of the monster including attack types and hit die size, and finally the environment in which the monster is found. The CR, type, and environment are used for database queries and thus come as no surprise as to why these are considered the most important statistics of the monster. If a game master is not looking for a specific monster to have the adventuring party encounter, the Challenge Rating, type, and Environment are the limiters that determine what would actually appear before the group.

The type is an important factor of the monster as evidenced previously and each monster type has some factors about them that are particular to the monster type. The item that always holds across all monsters of a type is the size of the hit die. The Hit die for a monster is a way of calculating the health points of a monster and is monitored by a certain number of die of a specific size. The Monster table contains the HitDie attribute which shows how many die are used in the monsters health, and the Type table contains the size of the hit die. Thus a complete monster would have health equal to “Monster.HitDie”d”Type.HitDie”, which translates out to rolling a die with Type.HitDie sides, Monster.HitDie times and adding the values. The Type table is shown below.

-- -----------------------------------------------------

-- Table `PathfinderEncounter`.`Type`

-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS `PathfinderEncounter`.`Type` (

`idType` INT NOT NULL,

`TypeName` VARCHAR(45) NOT NULL,

`HitDie` INT NOT NULL,

PRIMARY KEY (`idType`))

ENGINE = InnoDB;

The final table in the database is the attacks table. Different attack types follow similar damage values and all damage values are based off adjusting a value up or down by the size of the monster. For example, a medium-sized monster using a bit attack would deal 1d6 points of damage on an attack. If that monster increases in size to Large, the strength of the bite increases to 1d8. The attributes of this table are the id of the attacks, the attack name, and the different damage values for the attack based on the size of the monster.

-- -----------------------------------------------------

-- Table `PathfinderEncounter`.`Attacks`

-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS `PathfinderEncounter`.`Attacks` (

`idAttacks` INT NOT NULL,

`AttackName` VARCHAR(45) NULL,

`D` VARCHAR(45) NOT NULL,

`F` VARCHAR(45) NOT NULL,

`T` VARCHAR(45) NOT NULL,

`S` VARCHAR(45) NOT NULL,

`M` VARCHAR(45) NOT NULL,

`L` VARCHAR(45) NOT NULL,

`H` VARCHAR(45) NOT NULL,

`C` VARCHAR(45) NOT NULL,

`G` VARCHAR(45) NOT NULL,

PRIMARY KEY (`idAttacks`),

UNIQUE INDEX `Attack Name\_UNIQUE` (`AttackName` ASC))

ENGINE = InnoDB;

The book table is the least important of the tables included in the database. The purpose of this table is to show to the user what book from the pathfinder series the monster originates and whether the book is a third party book or not.

**II.2 Relationships**

Beyond the three tables, the database contains two relationships. These relationships are Monster\_has\_Attacks and Monster\_has\_Type. These relationships compare values of Monster to the Attacks and Type tables in order to generate complete information about the monster. The Monster\_has\_Attacks relationship looks at the ID of the monster to determine which attack IDs belong to that monster and also takes the monsters size to locate from where on the table to pull the damage value.

-- -----------------------------------------------------

-- -----------------------------------------------------

-- Table `PathfinderEncounter`.`Monster\_has\_Attacks`

-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS `PathfinderEncounter`.`Monster\_has\_Attacks` (

`Monster\_idMonster` INT NOT NULL,

`Attacks\_idAttacks` INT NOT NULL,

PRIMARY KEY (`Monster\_idMonster`, `Attacks\_idAttacks`),

INDEX `fk\_Monster\_has\_Attacks\_Monster1\_idx` (`Monster\_idMonster` ASC),

INDEX `fk\_Monster\_has\_Attacks\_Attacks1\_idx` (`Attacks\_idAttacks` ASC),

CONSTRAINT `fk\_Monster\_has\_Attacks\_Monster1`

FOREIGN KEY (`Monster\_idMonster`)

REFERENCES `PathfinderEncounter`.`Monster` (`idMonster`)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT `fk\_Monster\_has\_Attacks\_Attacks1`

FOREIGN KEY (`Attacks\_idAttacks`)

REFERENCES `PathfinderEncounter`.`Attacks` (`idAttacks`)

ON DELETE NO ACTION

ON UPDATE NO ACTION)

ENGINE = InnoDB;

The Monster\_has\_Type relationship pulls the monster ID to determine the appropriate hit die of the monster based on the id of the type.

-- -----------------------------------------------------

-- Table `PathfinderEncounter`.`Monster\_has\_Type`

-- -----------------------------------------------------

CREATE TABLE IF NOT EXISTS `PathfinderEncounter`.`Monster\_has\_Type` (

`Monster\_idMonster` INT NOT NULL,

`Type\_idType` INT NOT NULL,

PRIMARY KEY (`Monster\_idMonster`, `Type\_idType`),

INDEX `fk\_Monster\_has\_Type\_Type1\_idx` (`Type\_idType` ASC),

INDEX `fk\_Monster\_has\_Type\_Monster1\_idx` (`Monster\_idMonster` ASC),

CONSTRAINT `fk\_Monster\_has\_Type\_Monster1`

FOREIGN KEY (`Monster\_idMonster`)

REFERENCES `PathfinderEncounter`.`Monster` (`idMonster`)

ON DELETE NO ACTION

ON UPDATE NO ACTION,

CONSTRAINT `fk\_Monster\_has\_Type\_Type1`

FOREIGN KEY (`Type\_idType`)

REFERENCES `PathfinderEncounter`.`Type` (`idType`)

ON DELETE NO ACTION

ON UPDATE NO ACTION)

ENGINE = InnoDB;

**II.3 Triggers**

Beyond the tables in the schema there are multiple triggers created for the different table to catch errors that could occur when updating, inserting, or deleting from the tables. The Monsters table has two different types of triggers; before insert and before update. Both of these have the same checks, but need to be applied both before data is entered into the database and edited in the database to ensure the data is within the bounds required. If the program finds any values outside of the appropriate scales an error message is flagged pointing the user to what problem occurred so that they may address the issue. The different error checks that occur in these two triggers are as follows. First the size is checked as the size needs to be a value from -4 to 4; any other values are outside of the range and cannot be assigned to the size. There are 9 alignments present in the game thus if an alignment is entered that does not fit the 9 alignments, it will raise an error. While the CR of monsters can have fit on a wide range of values, for the purpose of this database we are working with a subset where the CR is from 1 to 10 and thus the trigger checks to see if the CR is within those values. Armor values besides size and dexterity cannot be a negative number and thus there is a check to see if the armor, shield, deflection, natural armor, dodge, or miscAC are negative. A creature cannot have 0 Hit die nor can they have a negative speed, space, or reach and these values are examined. The monsters base stats (Str, Dex, Con, Inte, Wis, Cha) cannot go below 0 in the game and are checked. Lastly if a monster is updated or inserted that has an attack type not present in the attacks table, an error is raised to contact the database manage in order to have that attack type registered.

CREATE DEFINER = DatabaseManager TRIGGER `PathfinderEncounter`.`Monster\_BEFORE\_INSERT` BEFORE INSERT ON `Monster`

FOR EACH ROW

Begin

DECLARE msg VARCHAR(255);

DECLARE found\_it INT;

IF NEW.size < -4 OR NEW.size > 4 THEN

set msg = "Error: Size categories must be an integer between -4 and 4";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

IF NEW.alignment <> 'LG' AND NEW.alignment <> 'NG' AND NEW.alignment <> 'CG' AND NEW.alignment <> 'LN' AND NEW.alignment <> 'N' AND NEW.alignment <> 'CN' AND NEW.alignment <> 'LE' AND NEW.alignment <> 'NE' AND NEW.alignment <> 'CE' Then

set msg = "Error: Invalid input for alignment";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

IF NEW.CR < 1 OR NEW.CR > 10 THEN

set msg = "Error: CR must be an integer between 1 and 10";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

IF NEW.Armor < 0 OR NEW.Shield < 0 OR New.Deflection < 0 OR NEW.NaturalArmor < 0 OR NEW.Dodge < 0 OR New.MiscAC < 0 Then

set msg = "Error: All Armor Class values must be greater than 0";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

IF NEW.HitDie < 0 OR NEW.BaseSpeed < 0 OR New.Space < 0 OR NEW.Reach < 0 Then

set msg = "Error: Hit Die, Speed, Space, and Reach must be nonnegative";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

IF NEW.STR < 0 OR NEW.DEX < 0 OR New.CON < 0 OR NEW.INTE < 0 OR New.WIS < 0 OR NEW.CHA < 0 Then

set msg = "Error: Stats must be nonnegative";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

if new.attack1 is not null then

SELECT COUNT(1) INTO found\_it FROM Attacks

WHERE AttackName = NEW.Attack1;

IF found\_it = 0 THEN

set msg = new.attack1;

-- set msg = "Error: Attack1 type not in database. Contact Database Manager for Assistance";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

end if;

if new.attack2 is not null then

SELECT COUNT(1) INTO found\_it FROM Attacks

WHERE AttackName = NEW.Attack2;

IF found\_it = 0 THEN

set msg = "Error: Attack2 type not in database. Contact Database Manager for Assistance";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

end if;

if new.attack3 is not null then

SELECT COUNT(1) INTO found\_it FROM Attacks

WHERE AttackName = NEW.Attack3;

IF found\_it = 0 THEN

set msg = "Error: Attack3 type not in database. Contact Database Manager for Assistance";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

end if;

if new.attack4 is not null then

SELECT COUNT(1) INTO found\_it FROM Attacks

WHERE AttackName = NEW.Attack4;

IF found\_it = 0 THEN

set msg = "Error: Attack4 type not in database. Contact Database Manager for Assistance";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

end if;

if new.attack5 is not null then

SELECT COUNT(1) INTO found\_it FROM Attacks

WHERE AttackName = NEW.Attack5;

IF found\_it = 0 THEN

set msg = "Error: Attack5 type not in database. Contact Database Manager for Assistance";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = msg;

END IF;

end if;

END;$$

Another trigger is set on the Attacks table before data can be deleted. The trigger checks to see if that attack type is in use by a monster in the database. If a monster currently in the database has that attack type than an error is raised and the attack cannot be deleted.

There are two triggers in the Book table that both accomplish the same task. The trigger is both before insert and before update on Book and checks to make sure that the value for is Third Party is either a 0 or 1 representing that it either isn’t or is a third party book.

The last two triggers occur on the Type table. The first occurs before an update or insert can occur on the table. The size of the Hit Die for a monster cannot be less than 1 or that would cause a creature to have no health points, thus there is a check to ensure that the HitDie attribute is at least 1. The other trigger checks before a value is deleted from the Type table and looks to see if a monster currently in the database is of that type. A type cannot be deleted if a monster currently in the database has that type.

**II.4 Users**

The database also gives certain permissions to different users in the database. The database as it stand contains three different users; DatabaseManager, Engineer, and User. The DatabaseManager is the owner of the database and thus is given all privileges. The Engineer can select, insert, trigger, update, and delete on tables as well as execute a routine. The User can only select on table.

DELIMITER ;

Flush Privileges;

CREATE USER 'DatabaseManager' IDENTIFIED BY 'database';

GRANT ALL PRIVILEGES ON `PathfinderEncounter`.\* TO 'DatabaseManager';

CREATE USER 'User';

GRANT SELECT ON TABLE `PathfinderEncounter`.\* TO 'User';

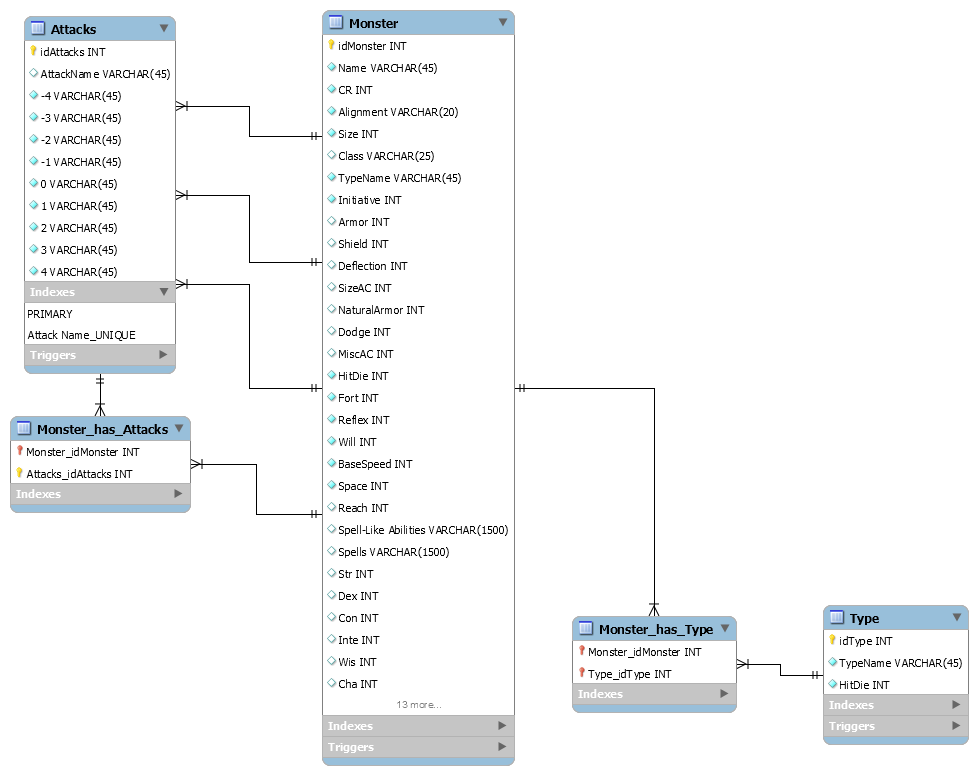
CREATE USER 'Engineer';

GRANT SELECT, INSERT, TRIGGER, UPDATE, DELETE ON TABLE `PathfinderEncounter`.\* TO 'Engineer';

GRANT EXECUTE ON ROUTINE `PathfinderEncounter`.\* TO 'Engineer';

**II.5 ER Diagram**

The ER diagram shows the tables and relationships used in this database.



1. **DDLs (Create Table Constructs)**

In order to test the database reasonably, a small test dataset of monsters was used. We used 4 total monster types (Plant, Animal, Undead, and Magical Beast), and 10 monsters inside of each type for 40 total monsters. Each of the 10 monsters in each type was a different CR value from 1 to 10.

1. **GoLang Code**

We chose to go with Google's GoLang for a few reasons, it has been implemented to allow for fast and easy web application development and it is a more modern approach to creating a web based interface. Golang is a simple, efficient language with support for MySQL and HTML templates, as well as the ablility to link a function to a specific web page.

The root webpage is handled by landingHandler, a small function that loads a small form with two buttons that will take the user to either a query building HTML form or a insertion form. The code for the landingHandler is short and charaticistic of all of the functions.

func landingHandler(w http.ResponseWriter, r \*http.Request) {

t, err := template.ParseFiles("landing.html")

if err != nil {

http.Error(w, "Internal Server Error", 500)

return

}

t.Execute(w, nil)

}

All of the functions take a http.ResponseWriter and http.Request, use the ParseFiles function to read in a specific template and then call Execute on the template to serve that webpage up to the user. The second value passed to Execute is a list of values that can be used to replace specific tags in the template files, which look like {{.Name}}. In this case the form page does not require editing so nil is passed.

Beyond the landing page there are a few functions. iformHandler serves up the monster insertion form and is bound to /insert. IformHandler takes the user to /confirm, which triggers the newMonsterHandler. newMonsterHandler configures the values from the previous form and generates a call to the database to insert the new monster, if there are any errors the system closes and a message is logged, otherwise the user receives a confirmation page saying that the new monster has been successfully added to the database. In the other use case, the user is sent to /find with launches the pformHandler. From here queryHandler is trigged when the user submits their query, which brings them to /query. This page shows the data returned by the query in a table, each monster name is a link to /monster/[name]. When clicked the name links take them to a dynamically generated page that displays all the information about a monster in an standardized format. /monster/[name] is generated by the dataHandler function.

All of the functions look similar to the landingHandler except for dataHandler, queryHandler, and newMonsterHandler which also have to process raw data into a human redable state, or take human redable data and encode it properly for insertion into the database. Each of these comminicates to the database through the same series of calls.

rows, err = db.Query(query)

if err != nil {

log.Fatal(err)

}

rows.Next()

err = rows.Scan(&attribute1, &attribute2, &attribute3)

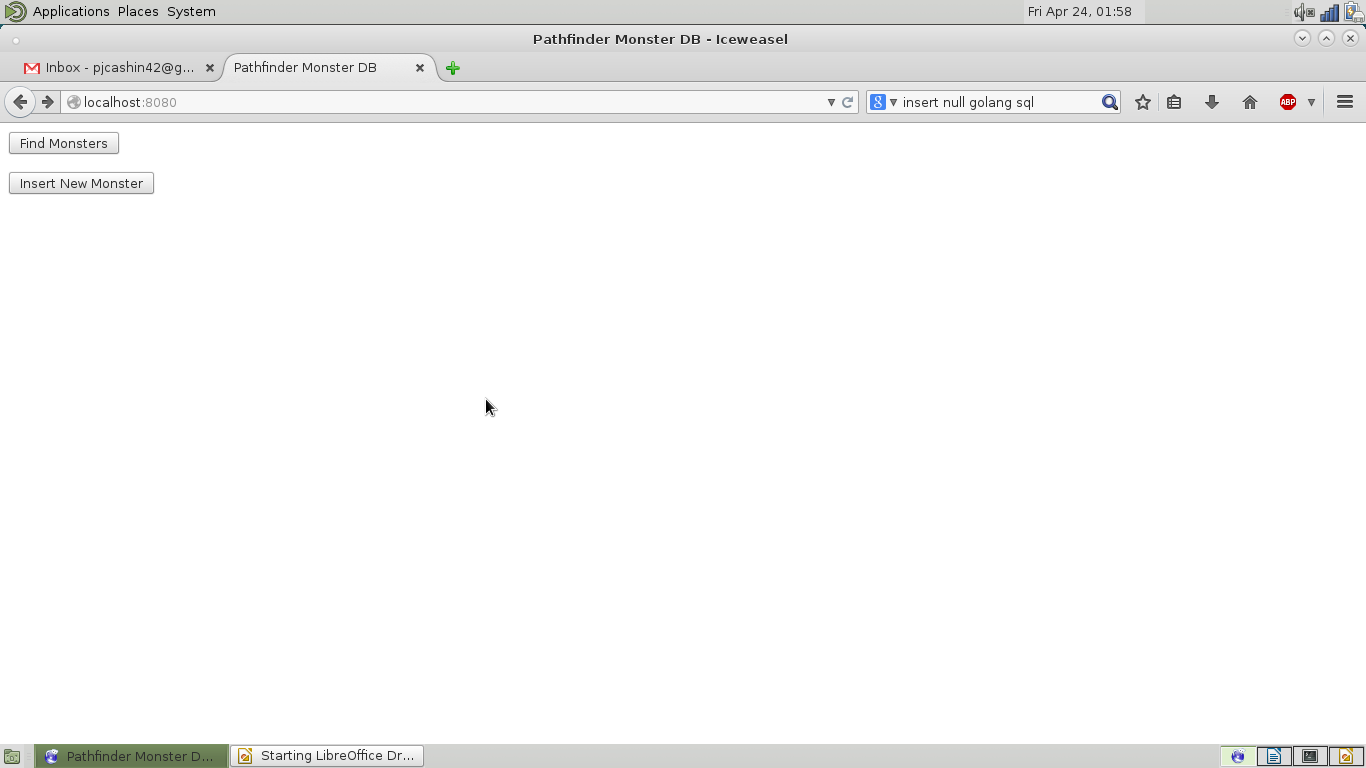
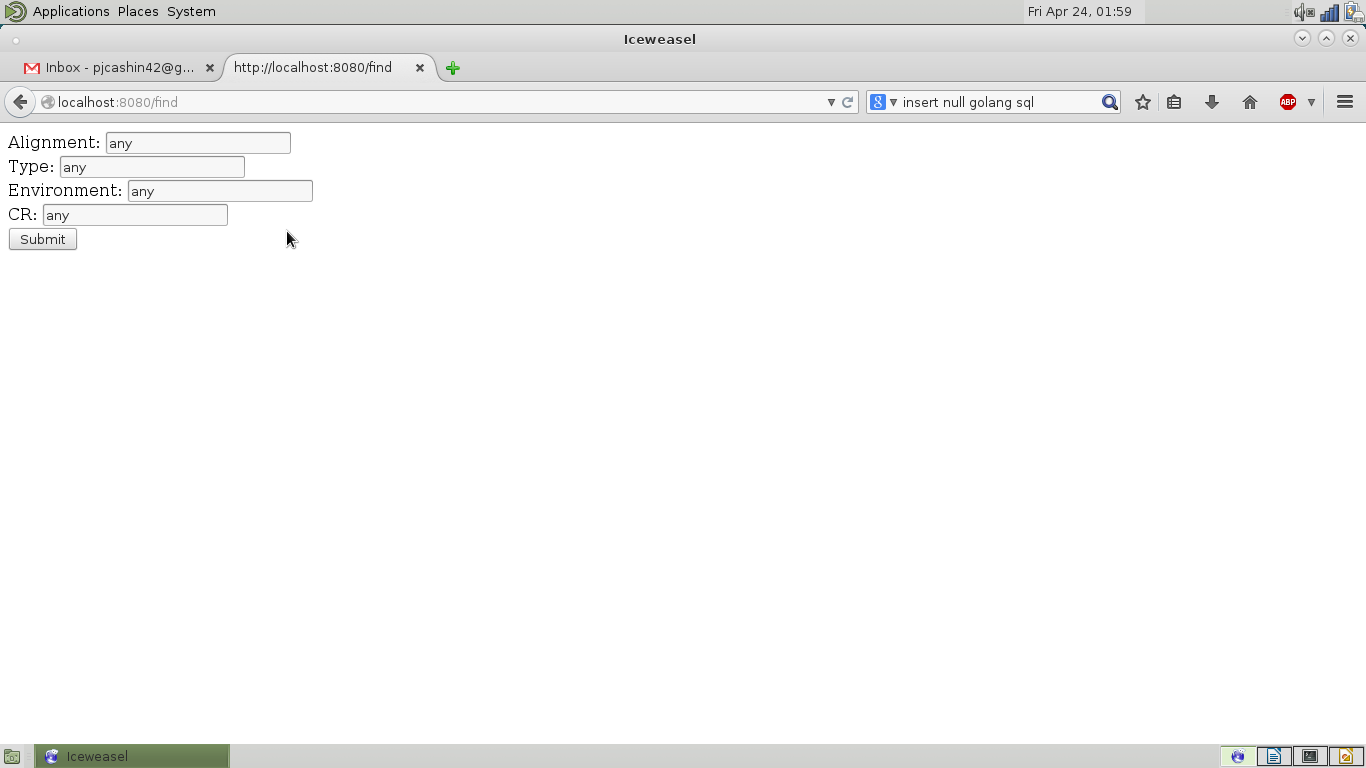
if err != nil {

log.Fatal(err)

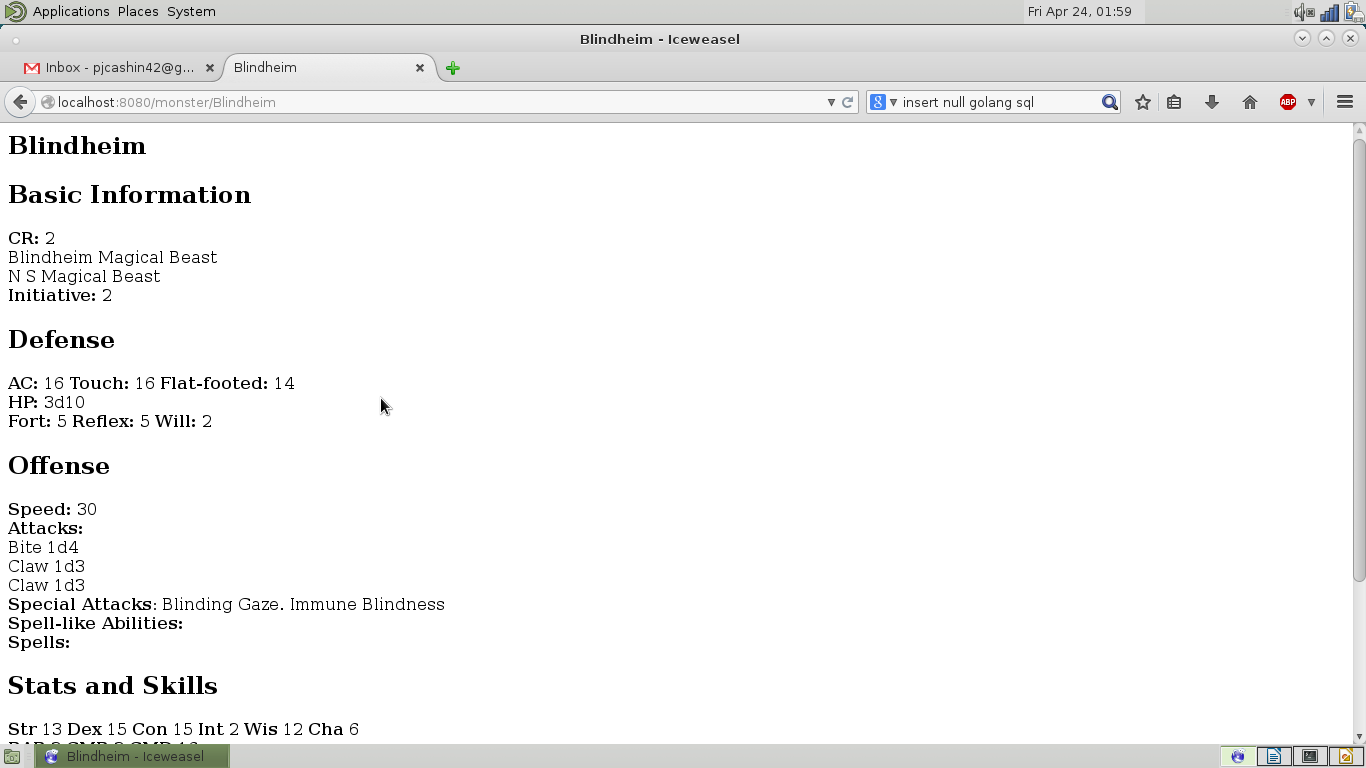
}

The above code sends a string, query, to the database and stores the response in rows, if there is an error the error is logged and the program terminates. Next() allows use to crawl through the data and Scan() parses the specific row, storing the column information in the given arguments, in the order in which the columns appear in the database. Calls to edit the database are similar, except we only catch the error code in case an update or insert fails.

1. **Screenshots and Sample Queries**

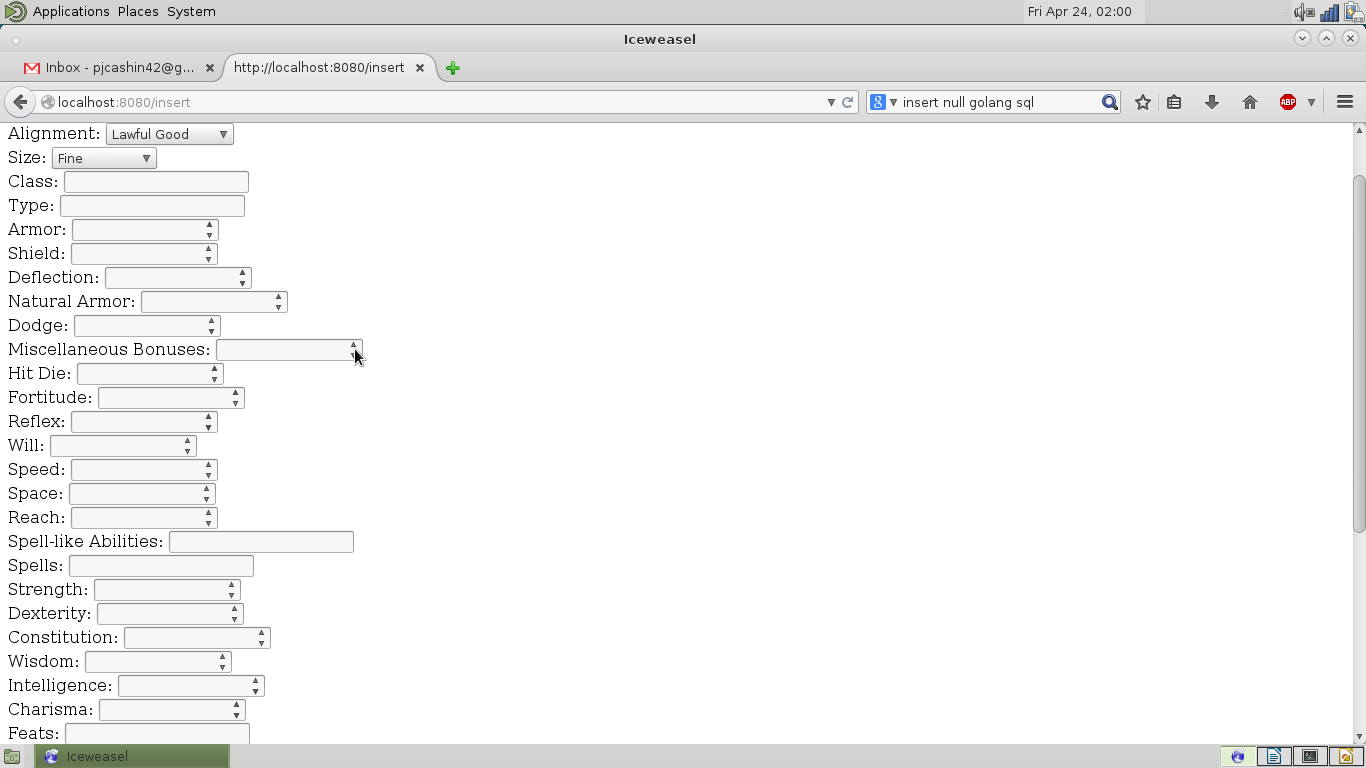


Above is the landing page used to direct users to the different functionality of the application. From the Find Monsters button, users are brought to the find page which has a simple form that allows the user to search for monster based on a subset of the monsters data. We decided to limit the control the user had over the queries because in an actual use case the user would only want to be shown monsters that fit these four requirements.



Above is the Monster Data page, this page is not stored by the server but built from a template per request. It contains all of the information about the monster in a more usable, standardized format.

This is a small image of the much larger insertion form. A monster is composed of roughly 40 different fields. All of which need to populated in order to create a valid entry. This feature should be available to the maintainer of the database only. Most users will be just generating previous pages. Since new material for the game is published about once a year, it would probably be more useful to generate a way to batch load a group of monsters into the database rather than insert them one at a time. If the monster is successfully inserted, a small confirmation page is posted.



1. **Logs for Future Extensions, Features left out**

The project as it stands is a barebones monster generator. There are a lot more intricacies that go into the creation of monsters in the Pathfinder RPG. Future Extensions of this project will incorporate special attacks into their own table with complete descriptions of the special attacks to give the game master much more information regarding how the monster can act. Also in future extensions, similar aspects will be used for both spells and spell-like abilities.

Future Extensions will also have a lot more dependencies on attributes that allow more customization of monsters. Currently the monsters that will be generated by this are stock monsters, however allowing dependencies in the future allows this to become richer. Such as the armor class of the monster combines many different traits that are outlined in the table and changing the type of armor that a monster has equipped will change their armor class, however the base generator created does not allow for the system to change the monsters armor to different types.

Templates were left out in this database due to the amount of dependencies needed to make templates work. Since templates change a lot of the attributes of the monster, every other attribute that would have a dependency with an altered attribute would also need to be changed, however that involves a much higher depth of logic in the system that was not appropriate at this time. As dependencies are introduced in the future, templates will be a natural table to include in this database and will give a new dimension to the generator.

1. **Team Member Responsibility Description**

The amount of work was split evenly between the two group members. David Yambay designed the overall database and programmed the schema including tables, triggers, users, and ER diagram. David Yambay also wrote all sections of the paper that involved the database. His other contribution to the work was the creation of which monsters would be used in the sample database.

Padraic Cashin designed the interface for the database using GoLang. There was a lot of in-depth work that went into the creation of the interface to make it function with the mysql database. Beyond that Padraic created the test scenarios for the queries and ran queries and generated screenshots of the interface running. Padraic and David shared the responsibility of writing down all of the monster information that populated the database.

1. **Lessons Learnt**

The biggest lesson that we have learned from this project is that adding in dependencies and logic into the mysql database is a large endeavor and requires a vast consumption of time. More knowledge of how to program using mysql will play a large role in the efficiency of making databases in the future. Our group only had basic knowledge of the language and thus every step took many tries to cause it to function.