Lego Inventory Tracker Final Report

Team Name:S2G8

Members: Luke Ferderer, Jamari Morrison, Jason Cramer

Signature Member #1: Luke Ferderer

Signature Member #2: Jamari Morrison

Signature Member #3: Jason Cramer

Table of Contents

1. Executive Summary

This final report provides a discussion of the driving problem the Lego Inventory Tracker project aims to solve and the solutions reached to solve it. The discussion of the design includes front-end and back-end parts of the system, details about the database design, and the challenges in creating the design. Finally, an analysis of the design will be given including its strengths and weaknesses. An appendix is also included which shows the entity relationship diagram, relational schema, and an explanation of the ER diagram. For reference purposes, a glossary of key terms, a list of references, and an index have been included at the end of this document.

2. Introduction

This document is the final report on the Lego Inventory Tracker project. This document will discuss the problem which inspired the creation of the project, the implemented solution, and evaluate the effectiveness of the solution. In the evaluation, strengths and weaknesses of the project, as determined by the creators, will be given.

3. Problem Description

Building with Lego bricks has been a popular hobby for millions of people since the mid-20th century [1]. Overtime, Lego has incorporated many different types of bricks and Lego sets to accommodate a plethora of hobbies and themes. Due to this fact, there are a large number of different bricks and brick sets that need to be inventoried. For many Lego collectors, it is currently a difficult process to keep track of bricks and sets within their collection. Our proposed solution is a website that allows users to track Lego bricks and sets via an inventory. This inventory would include information about what the users want, what they own, what sets they have built, and what bricks they have in use.

The overall purpose of our project is to streamline the Lego management process. Currently, it is a tedious process to determine what sets can or cannot be built using the Lego bricks one already owns. Normally, a user would have to manually compare the bricks they own to the bricks in the set they would like to build to see if they have all the needed parts. For all bricks they do not own, a list of parts to purchase could be created. This process takes too much time to be worthwhile. Our project provides a solution to this problem by making the process of comparing the bricks a user has to the bricks of any set instant, and allow them to create a wanted list of parts they need to complete a given set in a few clicks.

The Lego Inventory Tracker has the following features:

- Users can interact with our system through a React front-end connected to our database.
- Users can create accounts, delete their account and change their account password.
- Users can sign in and add Lego sets and bricks to their collections and wanted lists.
- The web interface can show users which sets they can build using their current brick supply and inform users of missing bricks they would need to acquire to build a specific set.
- Users can search for a set to see the difference between the bricks they have and the bricks required for a set.
- Users can mark their bricks as in use and their sets as built and filter their collections view to unused bricks

4. Solution Description

4.1 Front-End Discussion

Our front-end solution consists of a React web application with the goal to make the website as simple to use as possible. Our web application features a simple UI with an easy to navigate sidebar menu. Since our application uses Redux, we are able to have a sitewide user authentication state allowing the user to stay persistently logged in. The interface is centered around a user collection object which contains: the user's wishlist, the user's owned bricks or sets, and a searchable collection of either bricks or sets depending on the page the user has selected. The interface is uniform per entity to make it simple for the user to learn how to navigate the website and modify their collections to fit their collections and desires. For the wishlist and owned collections, the CRUD operations are supported through form input modals which prompt for the appropriate fields to create or update. A user is also able to search the database for entities (i.e bricks and sets) with different search modifiers such as search by name, id, or requirements. The user can also query the search to show sets they can build with the current available items in their collection.

4.2 Back-End Discussion

The back-end solution for our web application is an express REST api. The API provides the ability to execute our stored procedures and forward the data to the front-end application in it's expected format. The back-end has three primary controllers: bricks, sets, and users. The users controller is responsible for registering, login, authentication token generation, updating the password, and deleting an user's account. The API contains two custom middleware: an authentication manager and a login rate limiter. The login rate limiter is used to block an IP from making too many invalid login attempts within a given period of time. The authentication manager is used to verify and extract the user information from a JWT(JSON web token) and to prevent invalid authentication tokens from being used to access non-permissible data.

4.3 Database Discussion

The database which the back-end connects to provides all data-related functionality to the website. This includes storing user information such as accounts and inventories as well as Lego brick and set information such as names, photos, and IDs. To populate the database with Lego brick and set information, a web scraper was made to gather information from brickset.com [2]. Using this web scraper allows the database to have almost 500,000 rows of data about Lego products. Using the front-end, users add user-related data to the database which is done through the back-end running stored procedures. A more detailed discussion of the database design can be found in section 6 and additional diagrams can be found in section 8.

5. Key Challenges

1. Originally we created a query to get all sets that can be built with an user's available sets, however, it had a 7 minute runtime. After some consideration we devised an algorithm to simplify the runtime to nearly instant. The new query takes advantage of joins where our original method had a while loop iterate through all possible sets. The new query algorithm is broken into two main parts. First it finds the count of the bricks satisfied by each set. It does this by performing a join of the owned individual bricks and the set contains tables where the brick ids are the same and the available quantity(total quantity minus the quantity of use) of the owned bricks is greater than or equal to the quantity of the brick required for the set. Once generated, the query will count the total number of unique bricks in each of the sets that have at least one brick satisfied by the previous step. Lastly, the query will select the sets where the count of satisfied bricks are equal to the total number of distinct bricks in the set. Through this experience we learned a lot about how to approach query optimization. Instead of approaching the problem like an algorithm task in Java, we had to find alternative methods suited to SQL to generate a query without the use of iteration.
2. Our database requires hundreds of thousands of different rows. Collecting all of this data was one of their first challenges we faced. To solve this problem, we created a webscraber to get Lego brick and set information from brickset.com [2]. This led to more problems because brickset.com inconsistently rendered webpages. Within our web scraper, we had to create a lot of checks to make sure the data we were scrapping was valid; the Lego database we scraped had a very inconsistent layout. Furthermore the web scraper took days scraping before it acquired all the needed data. For this, we used someone's computer that they were able to leave running for a whole weekend and wrote the web scraper so that it wrote data to xml files as it scraped meaning it could be terminated at any time without data loss. This ended up saving hours of scraping time because the computer running the scraper crashed the second day in.
3. Our database was originally going to have a split between sets and models, where sets are the sets of bricks that can be purchased, while models are the things that can be built. This design was intended to differentiate between sets of assorted bricks and model sets, such as star wars ships. Unfortunately, we were not able to find a good way

to make this distinction in our data, as all of the Lego databases available saw what we defined as models and sets as the same thing. In the end, we made the decision to follow the lead of other Lego databases and just call everything sets.

6. Database Design

6.1 Security Measures: Stored Procedures & User Permissions

As discussed in our Database Security Analysis [3], our database is protected from denial of service attacks, SQL injection attacks, and user password compromisation. To prevent injections, the only way for the back-end to interact with the database is through stored procedures. This means nothing a user entered is ever entered into a SQL query directly which prevents SQL injection attacks. On top of this, our front-end checks data inputs into the stored procedures; it also ensures users are logged in and can only run user related queries on their account data.

The back-end is protected from Denial of service attacks as well. The express back-end implements user sessions to prevent an unauthenticated accessor of our front-end from making modifications to the database. It also limits the speed at which someone can run stored procedures on the database.

Prior to storing any user password in our database, we use bcrypt [4] to securely hash and store the passwords in the database. The hash is generated with a minimum cost factor of 13 for the salt. This ensures that if the database becomes compromised, user information will not be able to accessed. With hashed passwords, the only way to acquire someone's password is through brute force attacks. To prevent this attack, we limit the amount of failed password attempts to three every 10 minutes. The failed attempts are stored in a session variable and the server deauths the client if the failed attempts reach the failure threshold.

6.2 Integrity Constraints

6.2.1. Entity Integrity

1. The User table must contain a username that is a custom username type (varchar(20)) which is the primary key. Hashed passwords are stored as a nvarchar(70). Both fields must be not null.
2. A custom datatype for ImageURL was created to ensure tables with ImageURL fields are varchar(100) that are nullable.

3. The LegoSet table contains a varchar(20) ID which cannot be null and is the primary key. Name is a varchar(80) and cannot be null
4. The LegoBrick table contains an ID with type BrickID which is up to length 20 and is a numeric string of the format '#####/#####' where the amount of digits before and after the '/' are variable but non-zero. The name field is limited to 80 characters and is not null. Color can be up to 30 characters and is nullable.
5. The Contains table contains LegoSet which is a varchar(20) that is a foreign key that references a Lego set id and cannot be null. LegoBrick is a BrickID type that is a foreign key that references a LegoBrick entity and cannot be null. Quantity is an integer that must be greater than zero and cannot be null. LegoSet and LegoBrick from the primary key.
6. The WantsSet table contains a username which is of type username and cannot be null. LegoSet is an integer foreign key that references the id of a LegoSet entity and cannot be null. Quantity is an integer that must be larger than zero and not null. The combination of Username and LegoSet form the primary key.
7. The WantsBrick table contains a username which is of type username and cannot be null. LegoBrick is a BrickID type foreign key that references the id of a LegoBrick entity and cannot be null. Quantity is an integer that must be larger than zero and not null. The combination of Username and LegoBrick form the primary key.
8. The OwnsSet table contains a username which is of type username and cannot be null. LegoSet is a varchar(20) foreign key that references the id of a LegoSet entity and cannot be null. Quantity is an integer that must be larger than zero and not null. QuantityBuilt is an integer that must be larger than zero, not null, and less than or equal to Quantity. The combination of Username and LegoSet form the primary key.
9. The OwnsIndividualBrick table contains a username which is of type username and cannot be null. LegoBrick is a BrickID type foreign key that references the id of a LegoBrick entity and cannot be null. Quantity is an integer that must be larger than zero and not null. The combination of Username and LegoBrick form the primary key.

6.2.2. Referential Integrity

Foreign key relations require a behavior on update and delete. All update operations were set to cascade with deletes being split between cascade and reject. In explaining the reasoning behind the referential integrity behavior choices, tables will be grouped if the reasons for their choices are the same.

1. The WantsSet, WantsBrick, OwnsSet and OwnsBrick tables are almost identical in purpose: they describe a relationship between a user and a Lego product this user wants or owns. If the username is updated, the change is cascaded because the relationship still exists now with a different username

representing the same user. If the username is deleted, the user has been deleted from the database and their want and ownership of Lego products is no longer relevant to the database. For this reason, cascade was also chosen on delete. For the relevant Lego product for the table, if the ID is updated, a cascade operation occurs because the wants and owns relation still exists, now with a different ID for the Lego product. On deletion of the Lego Product, the change is rejected because people still want or own that product and it should not be removed from the database until those relations are removed. This prevents users from losing data if a Lego product is removed unexpectedly.

2. The Contains table describes a relationship between a LegoSet and the bricks it contains. If the LegoSet's ID is updated, the update is cascaded because it still contains the same bricks. If the LegoSet is deleted, the delete is also cascaded because the relation defining what bricks it contains is no longer relevant. If the brick ID changes, the change is cascaded because the LegoSet still contains that brick, now with a new ID. If the brick is deleted, the change is rejected because there is still a LegoSet that is using that brick. The brick should be removed from each LegoSet before deletion from the LegoBrick table.

6.2.3. Business Rule Integrity

When a set is added to a user's inventory, each brick that set contains is added to the user's individual brick inventory by either creating or increasing the quantity of the OwnsIndividualBrick relationship. When a model is deconstructed, the bricks required to build that model have their QuantityInUse decreased in the OwnsIndividualBrick relationship. When a set is removed, each brick from that set is removed from the user's individual brick inventory by either deleting or decreasing the quantity of the OwnsIndividualBrick relationship. When a model is built, the bricks required to build that model have their QuantityInUse increased according to their quantities in the model contents.

6.3 Stored Procedures

Our database uses stored procedures to help protect against SQL injection attacks and mask database complexity. Below is a list of all the stored procedures our database uses and their function.

Procedure Name	**Description**
bricksNeededForSet	Takes set and user as input, returns the information about the additional bricks that user would need to build the set.

delete_LegoBrick	Takes a Lego Brick ID as input and deletes the Lego brick with that ID from the LegoBrick table.
delete_LegoSet	Takes a Lego Set ID as input and deletes the Lego set with that ID from the LegoSet table.
delete_OwnsIndividualBrick	Takes a username, and Lego Brick ID as input and removes the Lego Brick from the target user's collection (from the OwnsIndividualBrick table)
delete_OwnSet	Takes a username and Lego Set ID as input and removes the Lego Set from the target user's collection (from the OwnsSet table)
delete_User	Takes a username as input and deletes the user with that username from the Users table.
delete_WantsBrick	Takes a username and Lego Brick ID as input and removes the Lego Brick from the target user's wanted list (from the WantsBrick table)
delete_WantsBrick	Takes a username and Lego Set ID as input and removes the Lego Set from the target user's wanted list (from the WantsSet table)
getAvailableBricks	Takes a username as input and returns all of the bricks that user has currently available to build with.
getBuildableSets	Takes a username as input and returns all of the sets the user's brick collection can build.
getOwnedBricks	Takes a username as input and returns all of the bricks in the user's collection.
getOwnedSets	Takes a username as input and returns all of the sets in the user's collection.
getPaginatedBricks	Takes a target page and the number of items per page as input and returns the chunk of brick data at the target page.
getPaginatedSets	Takes a target page and the number of items per page as input and returns the chunk of set data at the target page.
getSetContents	Takes a target set as input and returns all of the bricks

	in that set
getWantedBricks	Takes a username as input and returns all of the bricks in that user's wanted list
getWantedSets	Takes a username as input and returns all of the sets in that user's wanted list
insert_Contains	Takes a set id, Lego brick ID, and quantity as input and inserts that information into the Contains table
insert_LegoBrick	Takes a Lego Brick ID, Name, Image URL, and brick color as input and inserts that information into the LegoBrick table.
insert_LegoSet	Takes a Lego SetID, Name, and Image URL as input and inputs that information into the LegoSet table.
insert_OwnsIndividualBrick	Takes a username, quantity, quantityInUse and Brick ID as input and inserts that relationship into the OwnsIndividualBrick table.
insert_OwnsSet	Takes a username, quantity, and Set ID as input and inserts that relationship into the OwnsSet table.
insert_User	Takes a username and encrypted password as input and inserts that information into the user table.
insert_WantsBrick	Takes a username, quantity, and Brick ID as input and inserts that relationship into the WantsBrick table.
insert_WantsSet	Takes a username, quantity, and Set ID as input and inserts that relationship into the WantsSet table.
login_user	Takes a username as input and returns the password hash for that user
searchBricks	Takes a name string as input and returns bricks with names that contain the input name string.
searchBricksColor	Takes a color string as input and returns bricks with colors that contain the input color string.
searchBricksID	Takes an ID string as input and returns bricks with names that contain the input ID string.

searchSets	Takes a name string as input and returns sets with names that contain the input name string.
searchSetsID	Takes an ID string as input and returns sets with names that contain the input ID string.
setContents	Takes a set ID as input and returns the bricks that make up the contents of that set.
update_LegoBrick	Takes a Brick ID, name, Image URL and Color (at least ID, but not necessarily all) for a Lego Brick as input and updates the LegoBrick table entry with the given ID with the new information, only changing fields which were provided.
update_LegoSet	Takes an ID, name, and Image URL (at least ID, but not necessarily all) for a Lego Set as input and updates the LegoSet table entry with the given ID with the new information, only changing fields which were provided.
update_OwnsIndividualBrick	Takes a username, quantity, quantityInUse, and Brick ID (at least username and Brick ID, but not necessarily all) for an ownsIndividualBrick relationship and updates the table entry with the corresponding username and brickID, only changing values which were provided.
update_OwnsSet	Takes a username quantity, quantityBuilt, and SetID (at least username and SetID, but not necessarily all) for an ownsSet relationship and updates the table entry with the corresponding username and SetID, only changing values which were provided.
update_User	Takes a username and password hash as input and updates the information in the user table row with the related username.
update_WantsBrick	Takes a username, BrickID, and quantity as inputs and updates the quantity in the corresponding row of the WantsBrick table.
update_WantsSet	Takes a username, SetID, and quantity as inputs and updates the quantity in the corresponding row of the WantsSet table.

6.4 Views

Our database does not implement any views. This is because we were able to get all of the main benefits of views (focus data for users, mask database complexity, and simplify database management) using stored procedures.

6.5 Indexes

Our database has indexes on each attribute that has few recurring values and is commonly searched on, in addition to the default indexes set up on all primary and foreign keys. These indexes are on LegoBrick's name and LegoSet's name. These smoothen the user experience by speeding up the rate at which search results can be returned from the database.

6.6 Triggers

Our database does not use triggers because we were able to get the primary benefit of tiggers, table/data management, by handling all database editing through stored procedures that handle all the work that would be in triggers.

7. Design Analysis

7.1 Strengths

- The system maximizes security by limiting all data CRUD to operations done through stored procedures.
- The system's optimizations allow it to do complex operations on extremely large amounts of data, such as calculating what sets a user can or cannot build with their collection, in just a few seconds.
- The front-end interface allows for users with no sort of technical background to have a pleasant and intuitive user experience.
- All static passwords are stored in an encrypted state, maximizing user security.

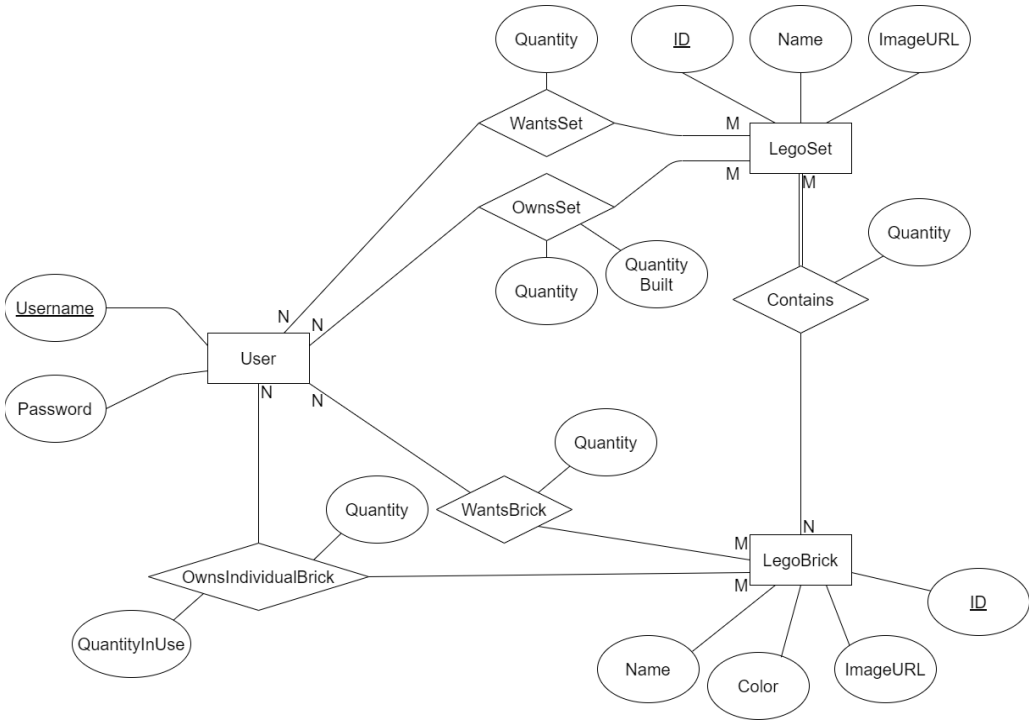
7.2 Weaknesses

- Since the database is working with very large amounts of data, it takes up a large amount of memory space (although this is helped by storing image links rather than full images).
- Adding a set to one's inventory or adjusting the amount of a set in one's inventory causes the website to freeze for a couple of seconds due to the execution time of the stored procedure.
- Users must enter the exact ID of Lego bricks and sets they wish to add to their inventory or wishlist. There is no way for users to more easily perform this action.

- The database does not automatically update with new Lego sets and bricks as they are released. It must be manually updated by a system administrator.
- Due to being unable to find good data, some sets are not included in the database. It is mostly decades old sets for which this was a problem but could affect some users if they have old collections.

8. Appendix

8.1 Entity Relationship Diagram



8.2 Relational Schema

User(Username, Password)
LegoSet(ID, ImageURL, Name)
LegoBrick(ID, ImageURL, Name, Color)
Contains(LegoSet, LegoBrick, Quantity)
OwnsSet(Username, LegoSet, Quantity, QuantityBuilt)
WantsSet(Username, LegoSet, Quantity)
OwnsIndividualBrick(Username, LegoBrick, Quantity, QuantityInUse)
WantsBrick(Username, LegoBrick, Quantity)

Contains(LegoSet) -> LegoSet(ID)
Contains(LegoBrick) -> LegoBrick(ID)

OwnsSet(Username) -> User(Username)
OwnsSet(LegoSet) -> LegoSet(ID)
WantsSet(Username) -> User(Username)
WantsSet(LegoSet) -> LegoSet(ID)
OwnsIndividualBrick(Username) -> User(Username)
OwnsIndividualBrick(LegoBrick) -> LegoBrick(ID)
WantsBrick(Username) -> User(Username)
WantsBrick(LegoBrick) -> LegoBrick(ID)

8.3 Explanation Of ER Diagram and Relational Schema

This system is based around the user, which interacts with Lego bricks and sets in various ways. The user can own a quantity of bricks and has the ability to mark a quantity of bricks as in-use. The user is also able to own a quantity of sets. Each set has a list of bricks and quantities that it contains and when a user adds a set into their inventory, all of the bricks contained in that set are added into their inventory as well. A user is able to mark a set as built, which also marks all of the bricks required to build that set as in use. The user is able to add quantities of sets and bricks into their wanted list, which acts as a sort of wish list inventory.

9. Glossary

Lego - Mainly consisting of interlocking plastic bricks, Legos are a toy created to allow for building and creation. Also refers to the LEGO Group, the company that makes Lego toys.

Brick - An individual Lego piece which is commonly referred to by its size.

Lego Set - A set of bricks that one buys from Lego that, when the instructions are followed, can be assembled into a specific object. An example of a Lego set would be the Star Wars X-Wing.

Web Scraper - An application that gathers information from websites.

Stored Procedure - A set of SQL statements that execute together. This functions like a function in most programming languages. It can take in input variables and return a status and the results of a SELECT statement.

React - A JavaScript framework for making interactive UIs, state management, and dynamic data rendering.

Express - A JavaScript library for handling HTTP requests

Json Web Tokens (JWT) - A token format that allows for a verification signature, JSON object (payload), and a header(contains the algorithm for the signature). It is a standardized format usually used for authentication tokens.

10. References

[1] The history of Lego: https://www.lego.com/en-us/aboutus/lego-group/the-lego-group-history
[2] Brickset.com - the website we used for acquiring data: https://brickset.com/
[3] Database Security Analysis: https://docs.google.com/document/d/1ZUL8CQQLU4k0yhEtJi4d6zk4noBVZN3F_eb5qChKMUc/edit
[4] Password security term Definitions: https://www.theguardian.com/technology/2016/dec/15/passwords-hacking-hashing-salting-sha-2#:~:text=Salting%20is%20simply%20the%20addition,same%20salt%20for%20every%20password.

11. Index