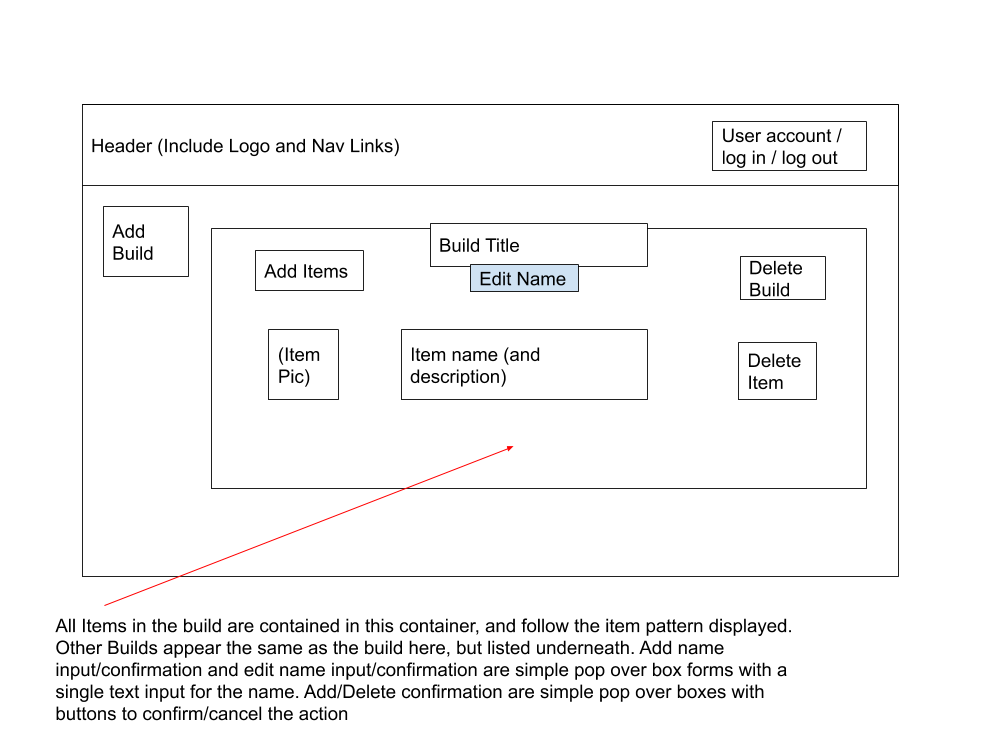
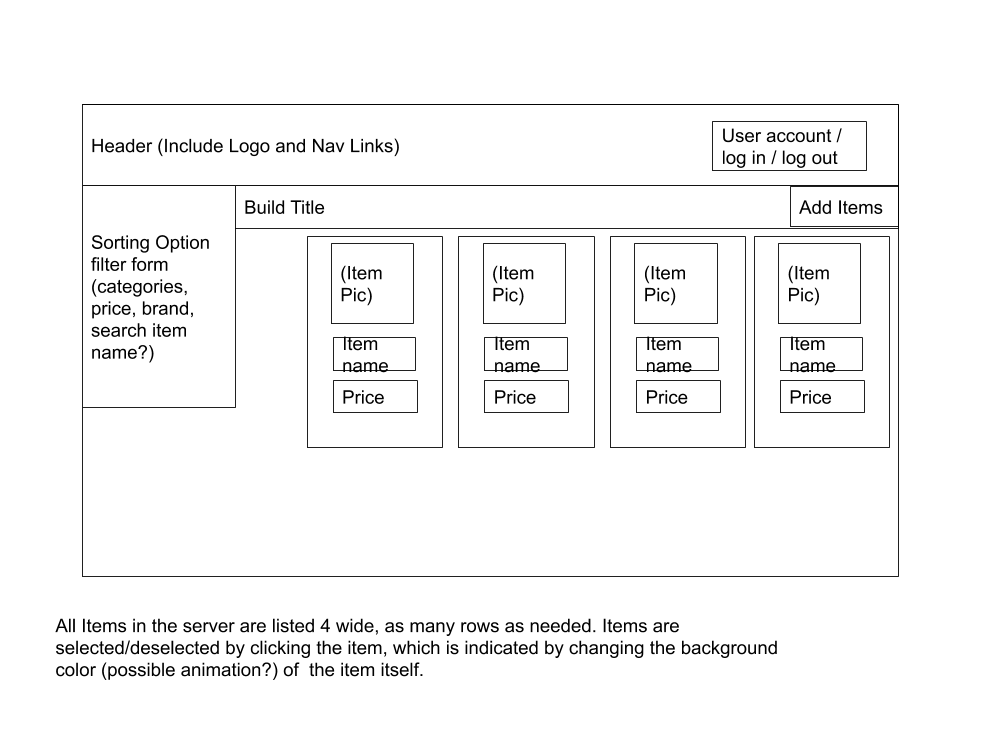
Passion Project Descriptions/Wireframes/Database Layout

Christopher Jones

N01352022





For GymBuildr, I intend(ed?) to implement a user system, so that users could create, edit, and delete only their builds, but they would be able to view other user’s builds, and possibly share their builds as well. I limited myself out of that scope due to my concerns about being able to finish/integrate everything by the submission deadline, but the method in which I have implemented my system/database still leaves the possibility of integrating the user system completely open, as all it would need is including the userid as a foreign key in the builds table, and then using the userid of the logged in user to display the associated builds for them, and controlling **C**r**UD** access likely using a build in token authentication system.

My Initial Database architecture was as follows:

gymitems

itemid = int 10 PK

brandid = int10 FK //MVP gymitembrandname varchar 255

itemprice decimal(10,2)

itemname = varchar 255

itemurl = varchar 255 nullable

//MVP gymitemimagepath varchar255 nullable?

images

imageid PK int10

//MVP itemid FK int10

imagerelpath varchar 255

imagealttext varchar 255

itemsximages

itemximageid int10 PK

itemid int10 FK

imageid int10 FK

brands

brandid int10 PK

brandname varchar255

brandurl varchar255

users (built in)

userid PK

userfname varchar255

userlname varchar255

useremail varchar255

userpassword varchar255

usertype tinyint

wishlists

wishlistid int10 PK

userid int10 FK

wishlistname varchar255

itemsxwishlists

itemxwishlistid int10 PK

itemid int10 FK

wishlistid int10 FK

Ultimately, the users table was excised for time constraints, the brands table was excised because I ultimately felt that it was an unnecessary table, as what it would allow me to do would be no better than simply storing the brand name in the item itself, and using a LIKE query if needed. Wishlists got it’s name changed to Builds, but it fundamentally remained the same. The Images and ImagesxItems got excised due to redundancy after your recommendations, as the images were instead just stored in a folder, and their naming convention allowed just the Image sub URL to act as the pointer from the Items table. Also, a categories and a itemxcategories table was added, where the bridging table is just linking the foreign key IDs, and the categories table is like the tags table, where it’s simply the primary key ID and a varchar Category name for each.

In this system, one gym Build can contain many items, but those same items can be used in multiple different builds, so it’s an obvious many-many relationship, and as such necessitates a bridging table.

Although it got excised, the original motivation behind the many-many relationship for images and items was accounting for the possibility that an image could feature more than one item, such as if a user was able to upload a picture of their build, and wanted to tag all the items in the photo, and if there was a feature that allowed you to see user pictures of an item, we wouldn’t want a photo getting constrained to a single item.

All the choices for datatypes are fairly self evident, int10s for the IDs, Ints for the integer datatypes, ItemPrice is stored as a Decimal(10,2) because it’s expressing a price in dollars and cents. And string data is stored as Varchar 255s.

Any Naming differences between the plan and the implementation was due in part to Visual Studio’s name autogeneration when creating the tables from the context, and my lack of understanding and unwillingness to go and try to change the table/column naming out of fear of messing the system up, so I instead proceeded with the naming structure provided by Visual Studio.

Going further with this project, I’d definitely want to integrate AJAX for the filtering of items on the modify page, as well as adding in a user system, and possibly some sort of script that would allow users to click, and it would open all the urls of the items in their build list, for when they want to actually buy their build.

If you want to import all the Item Data I used (that was collected from the Rogue website into a JSON object using a custom script and ParseHub’s framework and servers, and then converted into insert into statements using custom javascript I wrote), simply run the SQL file named ‘datainsert.sql’ in the root folder. The category linkings were manually added, because I intended to write a script and/or interface to handle it at a later state, so you will have to manually add some categories and link some items to said categories if you want to check it out (if the server implementation I have on this computer doesn’t get passed through automagically on GitHub). Ultimately, there is a lot to add/improve on this project before it could reasonably be viewed as portfolio-worthy/deployable, but a personal decision will need to be made if I want to continue this idea in the .NET framework, or if I want to rewrite the concept in a different language/framework.