Contents

1 Analysis

1.1 Problem Identification

1.1.1 Problem Description

Popular inventory management solutions are relatively expensive, and may be out of reach for individuals or small schools. Inventory systems have numerous benefits for businesses and individuals alike; a business may choose to track their supply levels where an individual may wish to catalogue their DVD collection.

My goal is to create a web-based application aimed at both businesses and individuals to manage inventory, with additional modern features such as automatic item re-ordering when stocks are running low.

Traditional inventory management solutions are typically single-user at best, whereas I intend to create a multi-user, collaborative environment.

In my view, an inventory system should be:

- Easy for end users to use.
- Cross platform
- Performant interface
- Efficient in terms of adding data
- Allow for easy cataloguing of inventory
- Allow for item scanning using QR codes / barcodes
- Be able to source data from external sources
- Support both consumable and non-consumable goods.

1.1.2 Stakeholders

Stakeholder requirements are further discussed for each stakeholder in the Stakeholder Requirements section.

Stakeholder	Description	Requirements	Capability
Claire Foley	Senior Leadership Team at The Village Prep School	Ability to manage library books. Admin and supervision of other users carrying out librarian tasks	Well-versed in computer use, at least when it comes to intuitive and well designed interfaces. Would struggle with a non-intuitive interface design.
Ella	”Head Librarian” (Pupil) at the Village Prep School	Ability to check in and out library books. User of the system; requires interface that is appropriate for her age.	Beginner user of technology, proficient in mobile applications on phones and tablets only. Rarely uses a laptop or desktop computer.
Generic Gear Rental Shop	Photography gear for hire business	Ability to manage business inventory in a fast and efficient manner.	Proficient with computers.

1.1.3 Why is it suitable to a computational solution?

1.2 Investigation

1.2.1 Preparation for interview

Question Set

- What would you consider your skill level to be regarding technology?
- Do you currently have a way to manage inventory?
- If so, what is your current solution?
- What aspects of this solution do you like?
- What aspects of this solution do you dislike?
- What features would you **require** in a custom solution?
- What features would **enhance** your experience?

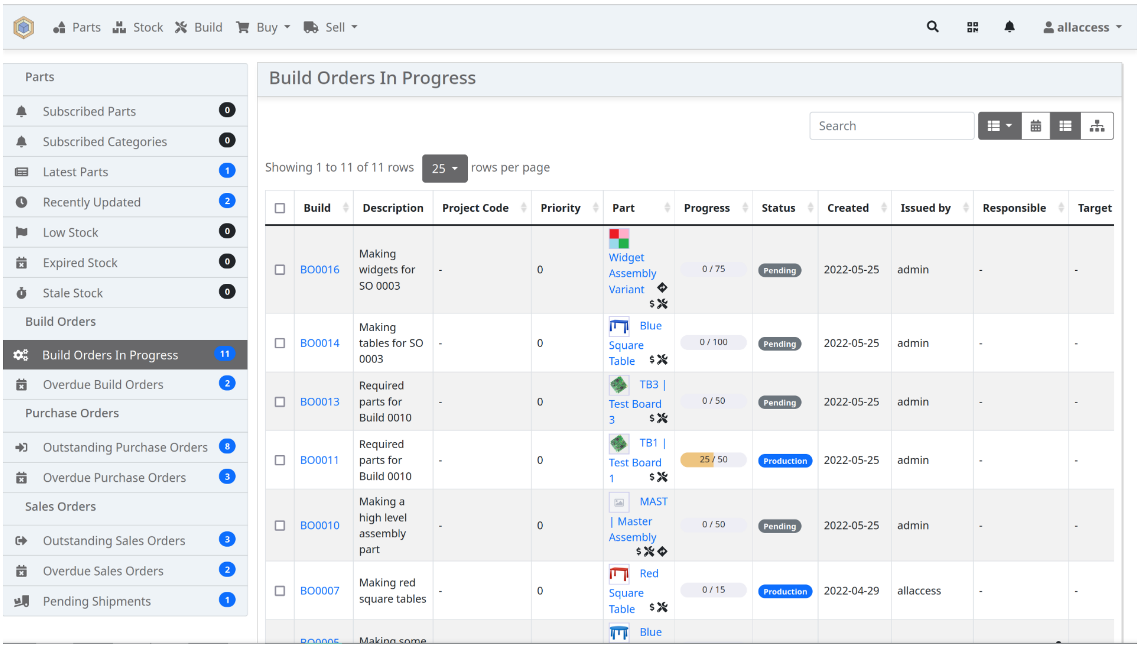
1.2.2 Interviews

1.2.3 Summary of interviews

1.3 Research

1.3.1 Existing similar solutions

InvenTree `https://inventree.org/`



Overview

InvenTree is an **open-source** inventory management system, providing *low level stock control and part tracking*. It uses a Python/Django database backend and provides both a **web-based interface** as well as a REST API for interacting with other services. InvenTree also has a powerful plugin system for custom applications and other extensions.

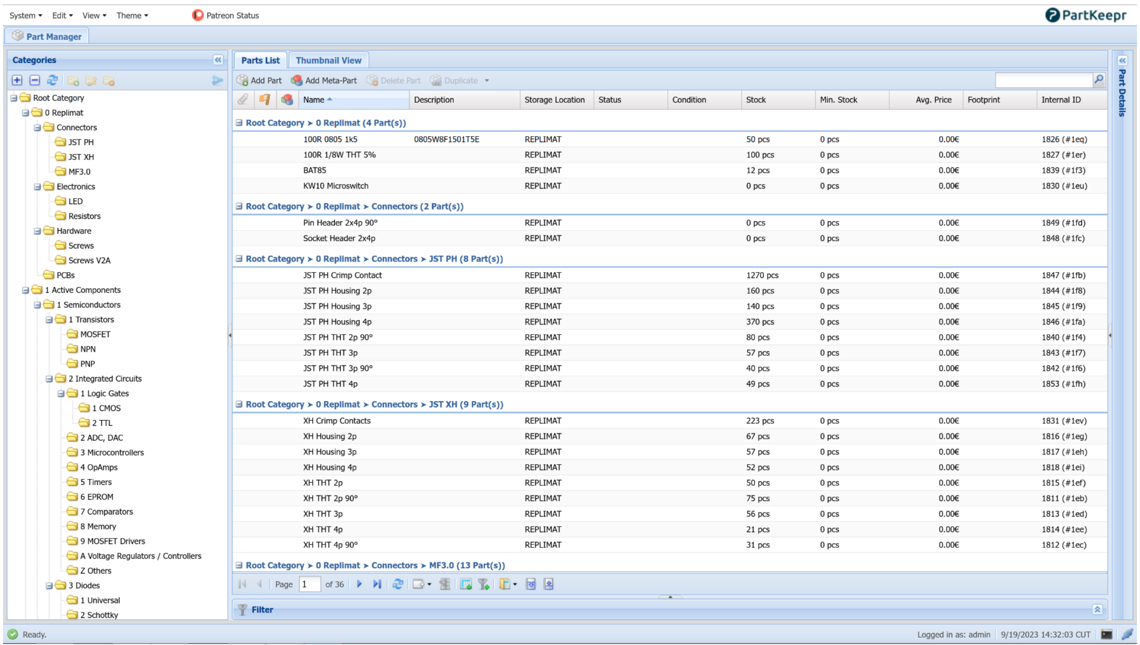
Parts applicable to my solution

- Web-based application
  The application will be web-based.
- Modern, Relatively simple user interface
  InvenTree offers a relatively simple and intuitive user interface.

Parts not applicable to my solution

- Stock control and part tracking specific features
  I am looking to implement a system that is capable of being far more generalized than just part tracking, although the system will have features for library book tracking.

PartKeepr `https://partkeepr.org/`



Overview

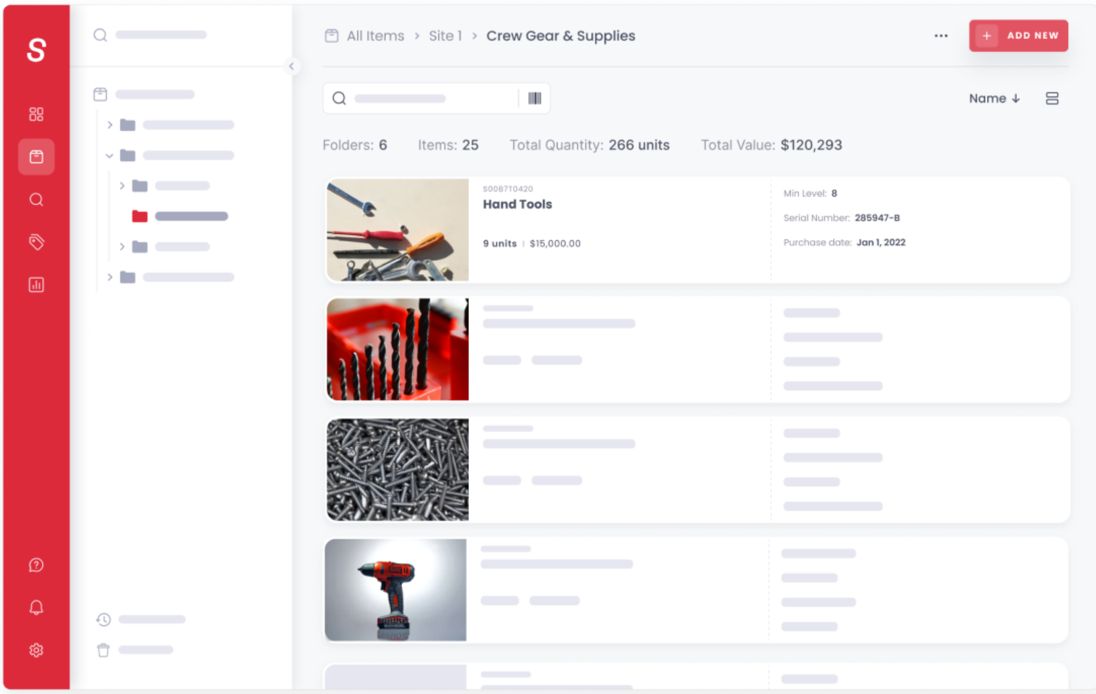
PartKeepr is an open-source inventory management system with a focus on electronic components. It is designed around four main principles:

- Fast Part Searching
- Ability to add complete part database
- Keeping track of stock
- Ease of use

Parts applicable to my solution

Like PartKeepr, I hope to implement a web-based interface. However, I am using a different approach as my solution will not be tailored specifically to electronic components.

Sortly `https://www.sortly.com/solutions/inventory-management-software/`



Overview

Sortly is a proprietary cloud-based inventory management system with a focus on small businesses and inviduals.

It has two plans available, an always free plan with limited functionality and a paid plan will a more complete feature-set.

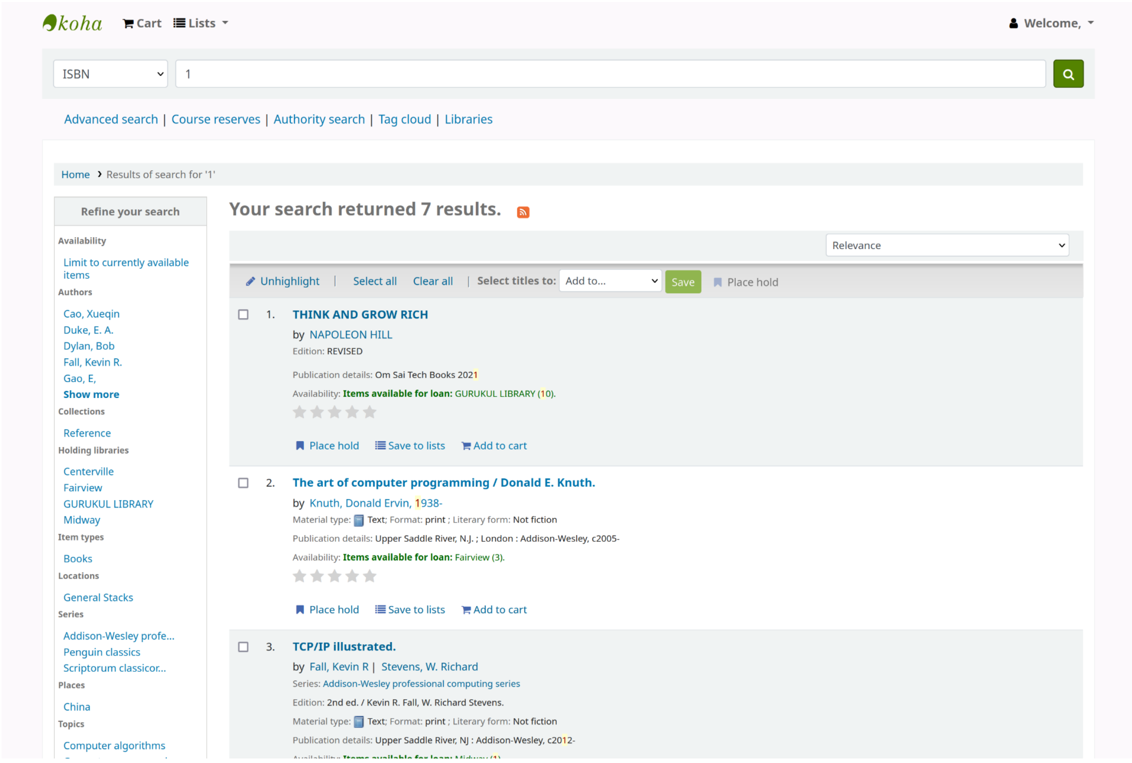
Parts applicable to my solution

I hope to implement the following features from Sortly:

- Web based interface
  - Allows for easy access.
- Barcode support
  - Allows end users to print off QR codes to stick to items
  - Which can be scanned in-app to easily perform actions on the item.
- Real-time reporting insights
  - Allows for added insight into usage patterns for particular units.

Koha

`https://koha-community.org/`



Overview

Parts applicable to my solution

1.3.2 Features to be incorporated into solution

1.3.3 Limitations of the solution

1.3.4 Feedback from stakeholders

1.4 Requirements

1.4.1 Stakeholder requirements

1.4.2 Software and hardware requirements

System Requirements

Hardware	Justification
Laptop/Desktop Keyboard and Pointing Device (eg. Mouse)	For desktop or laptop computer users, a suitable input device is required in order to interact with the software. A pointing device (a mouse) is necessary in order to interact with the user interface, to perform actions such as clicking buttons, icons, and opening menus. A keyboard will be used to manually input data into the system.
Tablet Device Touchscreen	For tablet users, it would be impractical to expect the user to have access to a keyboard and or pointing device. Therefore, we must design the system to accept inputs from a touchscreen. This will be easier to use and more intuitive for tablet users. The touchscreen will be used to input data into the system and to interact with the user interface.
Dual-Core Processor (x86, ARM, RISCV architectures)	A modern processor with sufficient resources to run an up-to-date web browser such as Chrome, Edge or Firefox is required in order to access the web-based interface.
2GB of RAM	Sufficient RAM is required to run the web browser, which can be a memory intensive task.
Monitor	To display the user interface.
Network Interface Card (NIC)	A Network Interface Card, or NIC, is required for the computer to be connected to a network, such as the Internet. This is required as the web interface will be hosted on a domain and server that is external to the user, that is to say, not on their local network.
Optional: *Wireless Network Adapter*	*A Wireless Network Adapter is an optional requirement, it will allow the user to connect to a wireless network in order to access the network or Internet so that they can access the external user interface.*
Optional: *Camera*	*A Camera is an optional requirement; devices with cameras will be able to scan barcodes or QR codes corresponding to inventory items and easily perform actions on them.*

Software Requirements

Talk about why I don’t need much software since dependencies hosted on the server.

Software	Justification
Operating System *(Windows, MacOS, Linux, ChromeOS, iOS, iPadOS, Android)*	An operating system is required in order to run the web browser necessary to access the interface.
A web browser *(eg. Chrome, Firefox, Microsoft Edge, Safari)*	A web browser is necessary to access the interface as it will be primarily a web application.

1.4.3 Success requirements

The overall objectives for the system.
To measure the overall effectiveness of the system, targets must be set before writing the program. These targets will help in the evaluation stage to determine weather our objectives have been met. These objectives will be **SMART**, i.e:

- **Specific**
  What objective needs to be accomplished?
- **Measurable**
  How can we quantify this objective?
  How will the success of this objective be measured? (quantitatively or qualitatively)
- **Achievable**
  Is this objective achievable and realistic? If so, how to you plan to achieve them?
- **Relevant**
  How does this objective benefit the end-users of this application as a whole?
  Why has this goal been set?
- **Timely**
  Can this objective be completed within an appropriate time frame?
  At what stage in the software development lifecycle will you start implementing this goal?
  In which order will any sub-objectives be completed?

The Project's SMART Objectives

1. **To produce a solution for cataloguing a school library and recording users and books borrowed**
   At the end of the project, I will evaluate against my success criteria and determine weather this objective has been met. On the software side, I will be using React, Expo and PostgreSQL. This objective will be the main objective for this project. This objective must be completed by March 2024.

2. **To produce a solution including a database that can store details of books, borrowers, loans and returns**

3. **To produce an intuitive and easy to use solution**
   I will evaluate my success on this objective by having a new user without any prior training or advice use the system and try to carry out a number of tasks without any assistance. If the user is able to successfully complete the tasks I will consider the system to be intuitive and easy to use and therefore this objective satisfied. To achieve this I will design my system to have a consistent layout based on **Material Design 2**, (`https://m2.material.io/`) the design language used by Google products and many apps running on the Android operating system. I will also use language that is a) appropriate for the situation the product will be deployed in (with young children) and b) easy to understand (so that children can interact with the system) I will also use meaningful error messages so that the user has a clear understanding of the problem that has occurred. This objective will benefit the end user as a intuitive and easy to use solution is critical to the usefullness of the project. If the end product is not easy to use, it is less likey to be used and accepted by my stakeholders. This objective will be worked on during the development process, and so will

be completed by the time development concludes. I will mock-up a version of the user interface in the design stage and will continuously iterate on the user interface during development.

4. **To produce a solution that features a fully searchable catalogue**
5. **To produce a solution that features reporting for overdue and/or lost books**
6. **To produce a solution that includes a curated "suggested reading list" for each borrower**
7. **To produce a solution containing a user interface that can be accessed via a mobile device**

2 Design

2.1 User Interface Design

2.1.1 Usability Features

2.1.2 Feedback from stakeholder

2.2 Modular breakdown

2.3 Algorithms

2.4 Data Dictionary

2.5 Inputs and outputs

2.6 Validation

2.7 Testing

2.7.1 Methods

2.7.2 Test Plan

3 Implementation

3.1 First Iteration — Initial Backend and Database

3.1.1 Introduction

In this sprint I will work on the backend service. This service will provide an interface for the frontend to talk to the database via an API (Application Programming Interface). I am writing the backend in Go, which is a performant, statically typed high level language designed by Google. Go is frequently used for backend development thanks to it's excellent performance and built in memory safety. I am also going to use GraphQL as the query language used by the frontend to interact with the backend.

GraphQL is an open-source query and manipulation language designed for use in APIs. The backend will serve as an API which will interface with my database. I choose to use GraphQL as it is better suited for larger, more complex data sources, and supports querying for multiple different types of data at once, unlike REST. It is also something I was interested in learning more about as I have not designed a system using it before.

James: Should this intro be in the design area instead?
TODO: explain what a graphql mutation is (it's a function)

3.1.2 User account creation

The first feature I decided to work on was user account creation. This would involve asking the user for an email address, name and password, before validating it and inserting it into the database. In addition, at a later stage, validation must be performed in order to ensure that:

- The user email is not already in use
- The generated user ID is unique and not already in use

For this early stage of development, I decided to use an SQLite database to make things easier. I hope that I can easily switch this to PostgreSQL (as specified in my design doc) later on in the development process.
I have created a GraphQL mutation called `createUser`. When it is called, the GraphQL library calls the `CreateUser` function, passing any input data from the query.
The database connection is available at "`r.db`".

My first version of this function was as follows:

```
// CreateUser is the resolver for the createUser field.
func (r *mutationResolver) CreateUser(ctx context.Context, input model.NewUser) (*model.User,
    error) {
    // Create the user struct
    user := structs.User{FirstName: input.FirstName, LastName: input.LastName, Email: input.Email}

    // Generate a user ID
    user.ID = uuid.New()

    // Create the database entry
    r.db.Create(&user)

    // Return the created user data, converting it to a GraphQL model.
    return &model.User{
        ID: user.ID.String(),
```

```
        FirstName: user.FirstName,
        LastName: user.LastName,
        Email: user.Email
    }, nil // return nil in the error field
```

The first thing I noticed after implementing this function was that it was tedious to convert back and forth between `structs.User` (the database object) and `model.User` (the GraphQL object). I decided to merge these into a single object. This was done with the following lines in my GraphQL library's configuration file:

```
models:

[..]

    # Custom models
    Checkout:
      model: github.com/jcxldn/fosscat/backend/structs.Checkout
    Entity:
      model: github.com/jcxldn/fosscat/backend/structs.Entity
    Item:
      model: github.com/jcxldn/fosscat/backend/structs.Item
    User:
      model: github.com/jcxldn/fosscat/backend/structs.User
```

This instructs the GraphQL library to use the structs I defined for the database as if they were GraphQL models.

Unique ID generation

I decided to use UUIDs (Universal Unique Identifiers) as IDs for all of the objects in my database. (eg. Users, Items) As seen above, I initially choose to simply call `uuid.New()` to generate a new random UUID. However, I would soon realise that it would be beneficial to perform **validation** in order to ensure that the UUIDs were actually unique, ie. that they were not being used by another object of the same type. For example, I wouldn't want two users to have the same User ID.

I decided to use a **for loop** to continuously generate UUIDs to be used as a potential User ID. I then perform a database lookup to ensure that the UUID is not already in use.
This can be done with the following code:

```
func (r *mutationResolver) CreateUser(ctx context.Context, input model.NewUser) (*structs.User,
     error) {

[..]

    isFreeUuid := false
    for !isFreeUuid {
        // Generate a UUID for the user id.
        user.ID = uuid.New()
        // Check that the UUID has not been used already
        // If true, it will break out of this for loop and continue.
        isFreeUuid = util.IsUuidFree[structs.User](r.db, user.ID)
    }
```

In order to achieve this and reduce code duplication across different functions,
I created a "IsUuidFree" utility function. Here is the code:

```

    func IsUuidFree[T any](db *gorm.DB, id uuid.UUID) bool {
        obj := new(T)
```

```
        err := db.Model(obj).Select("id").Where("id == ?", id.String()).First(&obj).Error
        if errors.Is(err, gorm.ErrRecordNotFound) {
            // Record not found, so user id is free
            return true
        } else {
            // Record was returned successfully, therefore the user exists
            return false
        }
    }
```

JAMES: this initially was different but I changed it to make it simpler. Should I include the old version as well?

This function makes use of **generics**. As per the Go docs:

> *With generics, you can declare and use functions or types that are written to work with any of a set of types provided by calling code.*

To simplify, generics mean that I can pass any struct (**T**) to the function. For example, if I call the function with:

```
util.IsUuidFree[structs.User](r.db, user.ID)
```

Then T is set to the type `structs.User`.

GORM (my database library) works by defining a struct to query for which corresponds to a table in the database (in this case the `Users` struct corresponds to the `users` table). We can then perform SQL actions on this table, such as Select.

Therefore, the GORM db call from above:

```
db.Model(obj).Select("id").Where("id = ?", id.String()).First(&obj)
```

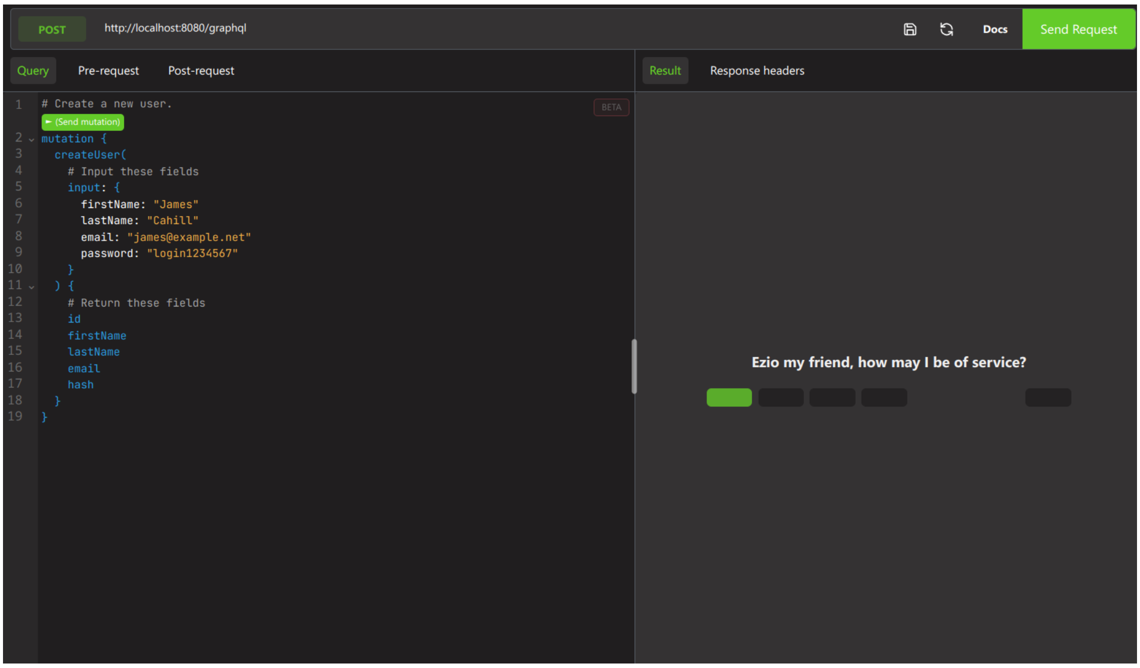
is the equivalent of:

```
SELECT id FROM users WHERE id == ? LIMIT 1 VALUES ("value of id.String()")
```

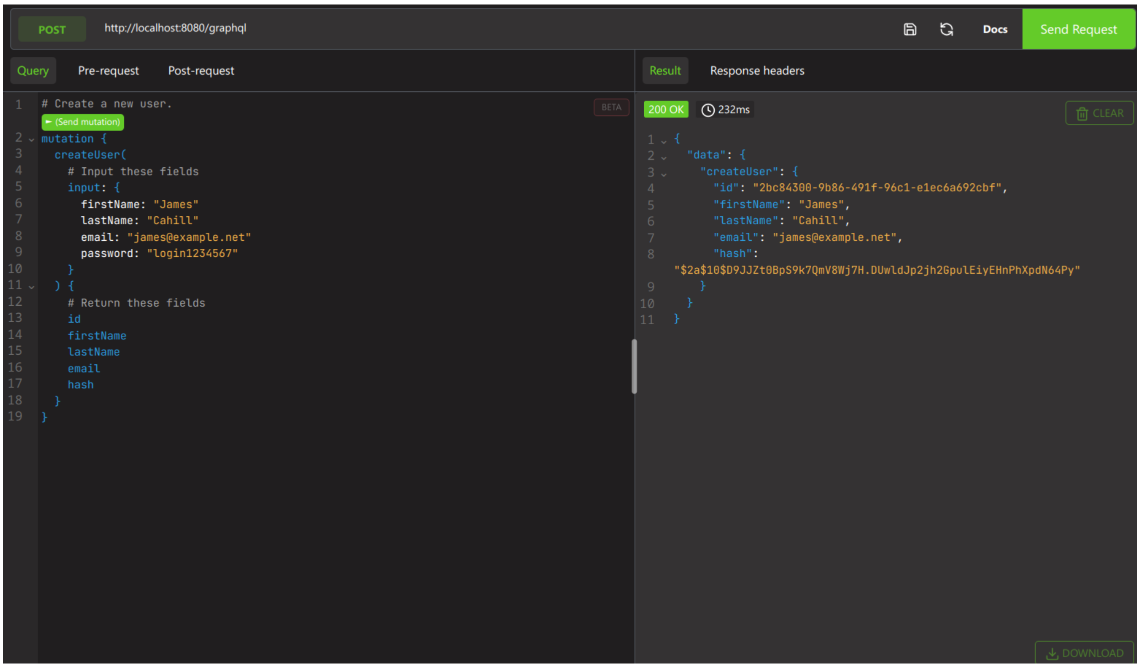
Testing the user account creation flow

Now that I have implemented user account creation, I need to test it to verify that my solution works as expected. I have added a GraphQL query to list all users, which I will use in conjunction with the `createUser` mutation.

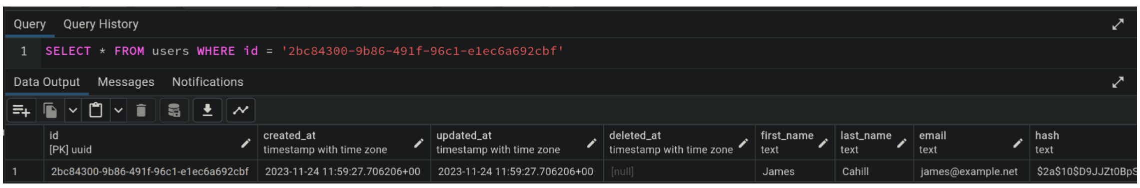
I am using a piece of software called **Altair**, which is an interactive way to make GraphQL queries. I start by creating my GraphQL query which includes the `createUser` mutation, making sure to set all required fields:



Running the query results in the following response:



Let’s check the database for our new user to ensure it was added successfully:



We can see that the user is created successfully, added to the database, and the specified fields (line 12 onwards in the query) are returned to the client.

3.1.3 Problems encountered when moving to PostgreSQL

At this stage, with most of the core functionality implemented, I decided to switch back to PostgreSQL. However, when doing this I encountered two problems:

Problem 1 - Entity relation errors

When creating the database in PostgreSQL, I encountered the following error, displayed in the backend logs.

```
/backend/database.go:35 ERROR: relation "entities" does not exist (SQLSTATE 42P01)

[17.308ms] [rows:0] CREATE TABLE "checkouts" ("id" text,"created_at" timestamptz,"updated_at"
    timestamptz,"deleted_at" timestamptz,"take_date" timestamptz,"return_date"
    timestamptz,PRIMARY KEY ("id"),CONSTRAINT "fk_entities_checkouts" FOREIGN KEY ("id")
    REFERENCES "entities"("id"))
```

This error could be traced to the following line in my code, where the database is "migrated" by GORM, which means that it attempts to create the necessary tables and columns to match the structs I have defined.

```
// "Migrate" the schema
// This will create tables, keys, columns, etc.
// See https://gorm.io/docs/migration.html
// Note that we need to pass each struct in our schema.
db.AutoMigrate(&structs.Checkout{}, &structs.Entity{}, &structs.Item{}, &structs.User{})
```

Could do: For example struct A creates this table? show it off?

This error prevented me from progressing with the backend implementation, as the program would error out during table creation (should I remove this line?)

After some investigation, I found out that this error occurs when tables that have dependencies on each other are created at the same time (in the same `AutoMigrate` call). The fix was to create the dependent table first followed by the table that depended on it, the code for which can be seen below:

```
// "Migrate" the schema
// [..]
db.AutoMigrate(&structs.Checkout{})
// Item has a dependency on Entity, so do them in the correct order
// to avoid "relation does not exist" error during table creation.
db.AutoMigrate(&structs.Entity{})
db.AutoMigrate(&structs.Item{})
db.AutoMigrate(&structs.User{})
```

After making this change, I decided to validate and test it before moving on.

I decided to test this change by first deleting all the database tables and then starting the backend, which should create (or "migrate") all of the tables.

Firstly I will connect to the database and delete the tables. I have attached the console output and have annotated what I am doing to make it easier to understand.

```
// Run 'psql' to connect to the database
[james@linux cs-coursework]$ psql -h localhost -U fosscat -W
// Enter the password (it is not displayed)
Password:
psql (15.4, server 16.1 (Debian 16.1-1.pgdg120+1))
```

```
WARNING: psql major version 15, server major version 16.
        Some psql features might not work.
Type "help" for help.

// List all tables, we can see that they exist
fosscat=# \dt
         List of relations
 Schema | Name | Type | Owner
--------+-----------+-------+---------
 public | checkouts | table | fosscat
 public | entities | table | fosscat
 public | items | table | fosscat
 public | users | table | fosscat
(4 rows)

// Delete the schema containing all of the tables
fosscat=# DROP SCHEMA public CASCADE;
NOTICE: drop cascades to 4 other objects
DETAIL: drop cascades to table users
drop cascades to table checkouts
drop cascades to table entities
drop cascades to table items
DROP SCHEMA
// Re-create the schema
fosscat=# CREATE SCHEMA public;
CREATE SCHEMA
// Set default permissions on schema
fosscat=# GRANT ALL ON SCHEMA public TO public;
GRANT
// List all tables, we can see that there aren't any
fosscat=# \dt
Did not find any relations.
// Quit
fosscat-# \q
```

As can be seen above, the database now contains no tables. Next, let's start the backend, where GORM should recreate the database tables. Below is the startup log:

```
[james@linux cs-coursework]$ GIN_MODE=release ./start-backend.sh
2023/12/05 14:23:08 [database] connected. migrating...
2023/12/05 14:23:08 [database] migrated, done.
2023/12/05 14:23:08 [resolver] db not set, setting.
```

You can see that no errors were produced in the console, indicating that all of the necessary tables and columns were created successfully. To check this, let's connect to the database again and list the tables:

```
// Run 'psql' to connect to the database
[james@linux cs-coursework]$ psql -h localhost -U fosscat -W
// Enter the password (it is not displayed)
Password:
psql (15.4, server 16.1 (Debian 16.1-1.pgdg120+1))
WARNING: psql major version 15, server major version 16.
        Some psql features might not work.
Type "help" for help.

// List all tables, we can see that they exist
fosscat=# \dt
         List of relations
 Schema | Name | Type | Owner
--------+-----------+-------+---------
 public | checkouts | table | fosscat
 public | entities | table | fosscat
 public | items | table | fosscat
 public | users | table | fosscat
```

```
(4 rows)

// Quit
fosscat=# \q
```

We can see that the tables were created successfully.

Problem 2 - Operator does not exist

However, when attempting to create a new user in the database I encountered another error:

```
./backend/util/user.go:14 ERROR: operator does not exist: text == unknown (SQLSTATE 42883)

[0.369ms] [rows:0] SELECT "id" FROM "users" WHERE id == '8596a222-930e-42f0-841e-9d95993668a4'
    AND "users"."deleted_at" IS NULL ORDER BY "users"."id" LIMIT 1
```

This error states that it cannot compare `id` with the given UUID (`'8596a222-930e-42f0-841e-9d95993668a4'`) because the operator `==` does not exist. This error points to my `IsUuidFree` function, which I talked about in the **User account creation** section of this iteration.

After some research, I found that the error occurs because the operator `==` does not exist in PostgreSQL. `==` is commonly used in programming languages to perform a deep comparison of two objects, and I assumed that the same would be true for PostgreSQL. Interestingly, the issue did not manifest itself until after the switch to PostgreSQL, meaning that SQLite handles `==` as I expected. The fix was simple; Change `'=='` to `'='`.

This meant that the line

```
err := db.Model(obj).Select("id").Where("id == ?", id.String()).First(&obj).Error
```

became

```
err := db.Model(obj).Select("id").Where("id = ?", id.String()).First(&obj).Error
```

Problem 3 - "... violates foreign key constraint"

When trying to create a new user, the following error message would appear:

```
/backend/graph/resolver/mutation.go:90 ERROR: insert or update on table "users" violates foreign
    key constraint "fk_checkouts_user" (SQLSTATE 23503)
[2.691ms] [rows:0] INSERT INTO "users" ("id", "created_at", "updated_at", "deleted_at",
    "first_name", "last_name", "email", "hash") VALUES [..]
```

This error occurred when trying to create a user. Upon reading into it the error occurred because of how I defined my foreign keys for GORM. For example, I had the following struct definition for `Checkout`:

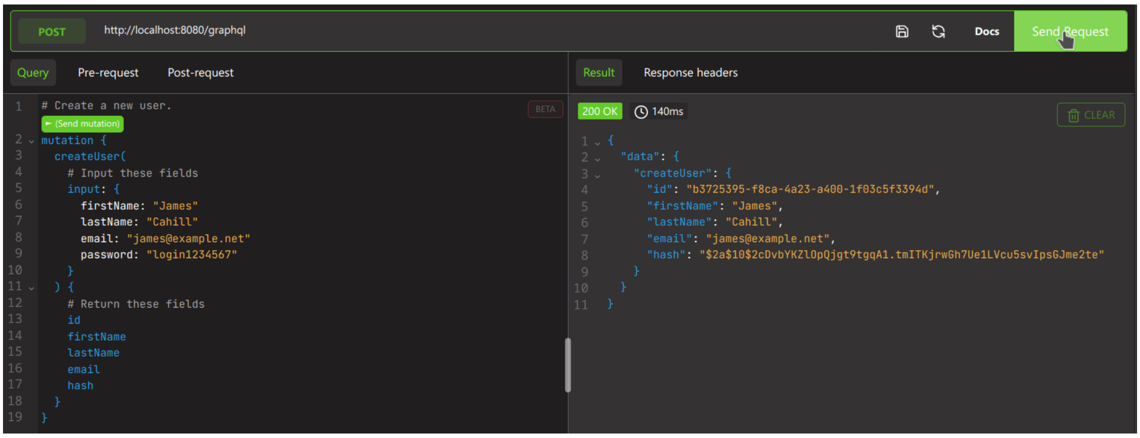
```
type Checkout struct {
    gorm.Model
    ID uuid.UUID
    User User `gorm:"foreignKey:ID"`
    TakeDate time.Time
    ReturnDate time.Time
}
```

Upon reading the GORM documentation, I realised this should actually be:

```
// Checkout belongs to a User, User.ID (UserID) is the foreign key
type Checkout struct {
    gorm.Model
    ID uuid.UUID
    User User
```

```
    UserID uuid.UUID
    TakeDate time.Time
    ReturnDate time.Time
}
```

After applying this change, the program worked as expected. I tested everything by attempting to create a new user in Altair:



This query returned successfully and the user was created without any errors.

3.1.4 Adding foreign keys for lists

3.1.5 Issues with nested queries (queries using multiple tables)

3.1.6 Adding the remaining queries

CreateCheckout validation

3.1.7 Testing

Test Plan

My plan for testing this iteration was to create unit tests for my project. Unit tests are an automated set of tests designed to ensure that the tested application works correctly.

After some research, I decided to use the "testify" testing framework, which is a toolkit with assertions and mocking support that works well with standard go functions. I also decided that it would be beneficial to have a testing database, so I decided to create a script that spins up an ephemeral (short-lasting, only lasts for the lifetime of the test run) database container using the *testcontainers* package.

A snippet of this script is included below:

```go
// SetupSuite is called before any tests run
func (s *DatabaseTestSuite) SetupSuite() {
    s.dbCtx = context.Background()
    // Create a container request for a container running the "postgres" image
    req := testcontainers.ContainerRequest{
        Image: "postgres", // container image to use
        ExposedPorts: []string{"5432"}, // ports to expose to host
        WaitingFor: wait.ForLog("database system is ready to accept
            connections").WithOccurrence(2), // trigger to define when container has started up
        Env: map[string]string{ // environment variables
            "POSTGRES_DB": "fosscat",
            "POSTGRES_USER": "fosscat",
            "POSTGRES_PASSWORD": "fosscat",
```

```go
        },
    }

    // Note that we do not define any persistent storage so the database will start from scratch
         every time it is created.

    // Create the container **and** wait for it to start up
    dbContainer, err := testcontainers.GenericContainer(s.dbCtx,
         testcontainers.GenericContainerRequest{
        ContainerRequest: req, // specify the container request
        Started: true, // automatically start once created
    })

    s.dbContainer = dbContainer

    if err != nil {
        panic(err)
    }

    // Determine the IP address of the container
    ctrIp, _ := dbContainer.ContainerIP(s.dbCtx)

    // Log that the database is now available
    log.Default().Printf("[test/common/database]: ephemeral db available on %s:5432", ctrIp)

    // Define connection details for the database
    dsn := fmt.Sprintf(
        "host=%s user=fosscat password=fosscat dbname=fosscat port=5432", ctrIp,
    )

    // Attempt to connect to the db
    db, err2 := gorm.Open(postgres.Open(dsn), &gorm.Config{})

    if err2 != nil {
        panic(err)
    }

    // Call migrate function (defined in backend/database/database.go) to create tables
    database.Migrate(db)

    // make the GORM instance available to tests
    s.DB = db

}
```

Test Results

Errors encountered during testing

Summary

Fixing error XYZ

3.1.8 Evaluation

In this iteration, I have successfully written code that can create a database, create the necessary tables to store data, and exposes functions to create and store data in the database.

Using GraphQL, user accounts can be created, where a unused unique identifier is found and used, as well as the creation of a salted password hash to securely store the user's password. In addition, checkouts, entities and items can be created each with their unique identifiers. Where necessary, such as with an checkout, we can use relationships to tie the checkout to a specified user object residing in the database.

The next iteration will focus on adding security to the backend. For example, I will need to add authenticated routes, where users can only view the contents of the route if they are logged in, as it will contain sensitive data.

4 Testing

5 Evaluation