# Checkout Backend

## Overview

This is a NodeJS application for calculating the total price of a provided shopping basket after calculating any applicable discounts.

## Quickstart

Before first run, run the `npm install` command. Run the application using `npm run start`. Run tests using `npm run test”. The server will start on localhost, using port 8080 is no PORT environment variable is set, or the PORT variable if it is.

## Data Storage

### Database Type

A SQLite database is used for storing the data, and by default is stored in the project route as db.sqlite. No authentication is utilised on the database due to limitations of the SQLite format. The better-sqlite3 library is in use for this.

### Database Structure

A screenshot of a computer

Description automatically generated

* The Product table contains the unit code and individual price
* The Offers table links products to offers using item code
* The Offers table contains the discount price and the number of the specified item required to activate the discount
* Item Code and Discount Quantity serve as a composite key in the offers table

### Database Setup

All database functionality is inside helpers/database.js

A setup function is called on application start to ensure the database is setup correctly.

 // Create the product table  
    db.exec(  
      "CREATE TABLE **IF** **NOT** EXISTS product (item\_code TEXT PRIMARY KEY **NOT** NULL, unit\_price INTEGER **NOT** NULL)"  
    );

 // Create the offers table  
    db.exec(  
      "CREATE TABLE **IF** **NOT** EXISTS offers (item\_code TEXT **NOT** NULL, discount\_quantity INTEGER **NOT** NULL, discount\_price INTEGER **NOT** NULL, PRIMARY KEY (item\_code, discount\_quantity))"  
    );

The product and offers tables are created, and the columns added to the tables.

  // **Insert** **initial** **data** **into** the product **table**  
    db.exec(  
      "INSERT INTO product (item\_code, unit\_price) VALUES ('A', 50), ('B', 35), ('C', 25), ('D', 12) ON CONFLICT(item\_code) DO NOTHING;"  
    );  
  
   // **Insert** **initial** **data** **into** the offers **table**  
    db.exec(  
      "INSERT INTO offers (item\_code, discount\_quantity, discount\_price) VALUES ('A', 3, 140), ('B', 2, 60) ON CONFLICT(item\_code, discount\_quantity) DO UPDATE SET discount\_price = excluded.discount\_price;"  
    );

The test data is then inserted into the database.

This ensures that the database is always running and accessible with the correct test data upon start.

## Routing

The app is a NodeJS application and uses Express for routing. The entry point is the index.js file, where the express application is instantiated, the json body parser is enabled, and the error handling is implemented.

A blue and white text

Description automatically generated

Figure 1. App is created



Figure 2. Body parser is added to the app to enable JSON request bodies

A computer screen shot of a computer

Description automatically generated

Figure 3. Checkout routes are added to the app

A computer code with colorful text

Description automatically generated

Figure 4. Error handling is implemented across the all routes

### Routes Folder

The routes are stored in separate files in the routes folder. This is done to make extending the application simpler, and to keep the code more maintainable.

### Checkout Functions

The /checkout POST route is contained within the checkout.js route file

This file contains several helper functions that handle the checkout process

#### Body Validation

A computer screen shot of text

Description automatically generated

This ensures that the request body is either an object, or an array of objects, with each object containing an item code, and an item quantity, where the item quantity is a number. If the body is valid, an array or objects is returned, else false is returned.

#### GetItemDetails

A screen shot of a computer code

Description automatically generated

This takes an array of item codes and queries the database for the corresponding items.

A left join is used to include information about discounts for items that have discounts available. Prepared statements are used to sanitise user-provided values.

#### Calculating Discounts

A screen shot of a computer code

Description automatically generated

This function takes information about a single requested item, and returns the total for that item, taking into account any discounts.

The function checks if there is a discount quantity and price available, and if there is not just returns the quantity multiplied by the price for a single unit.

Otherwise, the number of items that qualify for a discount is calculated, and this discounted amount is calculated and added to the total price. The remaining full price items are then calculated, with this total added to the total price.

#### /checkout POST endpoint

This endpoint is called for calculating the total for a provided basket.

A screen shot of a computer

Description automatically generated

The body is validated, with a 400 response returned if validation fails.

A screen shot of a computer code

Description automatically generated

The database is queried for the items that the user has requested. The number of returned results is checked against the number of requested items, and if fewer items are returned than requested, a 400 response is returned alerting the user to the fact an unknown item code was provided.

A screen shot of a computer code

Description automatically generated

The quantities of each item requested by the user are extracted into their own map, which is then combined with the data retrieved from the database, so that information about an item, its discounts, pricing, and the quantity requested are all stored in a single object.

A screen shot of a computer code

Description automatically generated

The total price for the individual requested items are calculated and stored in an array

A blue and white text on a dark background

Description automatically generated

The total of all items is then calculated



The subtotals and overall totals are then returned to the client

Any errors are caught and forwarded to the error handler.

## Testing

Jest is used for handling all tests, and can be run using the ‘npm test’ command.

The testing code is available in the \_test\_ folder.

A screen shot of a computer

Description automatically generatedWhen testing the server is started within the testing context, rather than in the index.js, as this allows closing the server at the end of testing.

### Example Test Case:

This is an example test used to ensure that it correctly calculates and returns values when multiple discounted items are in the basket

A screen shot of a computer program

Description automatically generated

The supertest library is used for creating the test requests.

## OpenAPI Spec

An OpenAPI spec is available in the openapi.yaml file.

## Logging

Logging is handled using the Winston logging library.

A screen shot of a computer program

Description automatically generated

The logger is setup to log both to console, and to separate error and combined log files, in the /logs directory.

A computer screen shot of text

Description automatically generated

An abstraction layer is provided to ensure that it is easy to change the underlying logging code when necessary.