Project Overview

This project is a simple **Receipt Processor** service built in Go. It reads a receipt's data (retailer, items, total, etc.), calculates a **points** value according to specified rules, and returns those points. It also provides an endpoint to retrieve the points for a previously processed receipt.

Requirements

- **Go 1.18+** (tested on Go 1.18 or higher)
- An internet connection for fetching dependencies via Go modules
- A tool to send HTTP requests, such as:
    - **URL** or
    - **Postman** (recommended for testing)

Getting Started

1. **Clone or download** this repository into your local environment.

2. **Initialize the Go module**:

   Bash:

   `go mod init receipt-processor`

   (Replace `receipt-processor` with any module name you prefer.)

3. **Install and tidy up dependencies**:

   Bash:

   `go mod tidy`

Dependencies

- `github.com/gin-gonic/gin` – Gin web framework for HTTP routing
- Standard Go libraries (no additional third-party packages besides Gin)

Running the Service

1. Make sure you're in the project's root directory where go.mod is located.
2. Run:

   bash

   go run main.go

3. The service should be running at **http://localhost:8080**.

Local Go Build

If you want to compile a binary:

1. From the project's root, run:

   bash

   go build -o receipt-processor main.go

2. After building, you can start the service:

   bash

   ./receipt-processor

3. The service will again be available on **http://localhost:8080**.

API Endpoints

POST /api/process-receipt

Description

Accepts a JSON object representing a receipt and returns a unique id, the same receipt data, and the computed points.

Request Body (JSON)

json

```
{
  "retailer": "string",
  "purchaseDate": "YYYY-MM-DD",
  "purchaseTime": "HH:MM",
  "items": [
    {
      "shortDescription": "string",
      "price": "string"
    }
  ],
  "total": "string"
}
```

Example Request

Bash

```
POST http://localhost:8080/api/process-receipt
Content-Type: application/json

{
  "retailer": "GroceryMart123",
  "purchaseDate": "2023-05-12",
  "purchaseTime": "14:30",
  "items": [
    {
      "shortDescription": "Bananas",
      "price": "3.25"
    },
    {
      "shortDescription": "Ice Cream",
      "price": "5.50"
    }
  ],
```

```
  "total": "8.75"
}
```

Example Response

json

```
{
  "id": "0096c2d9-c56a-4d45-9b1c-2e198b4a6078",
  "receipt": {
    "retailer": "GroceryMart123",
    "purchaseDate": "2023-05-12",
    "purchaseTime": "14:30",
    "items": [
      {
        "shortDescription": "Bananas",
        "price": "3.25"
      },
      {
        "shortDescription": "Ice Cream",
        "price": "5.50"
      }
    ],
    "total": "8.75"
  },
  "points": 93
}
```

GET /api/points/:id

Description

Retrieves the points associated with a previously processed receipt. The :id path parameter must match the id value returned in the response from **POST** /api/process-receipt.

Example Request

Bash

```
GET http://localhost:8080/api/points/0096c2d9-c56a-4d45-9b1c-2e198b4a6078
```

Example Response

json

```
{
  "points": 93
}
```

Testing with Postman

1. **Open Postman** and create a new **Collection** or **Request**.
2. **POST**:
    - URL: http://localhost:8080/api/process-receipt
    - **Body**: JSON (raw):

      json

      ```
      {
       "retailer": "GroceryMart123",
       "purchaseDate": "2023-05-12",
       "purchaseTime": "14:30",
       "items": [
         {
          "shortDescription": "Bananas",
          "price": "3.25"
         },
         {
          "shortDescription": "Ice Cream",
          "price": "5.50"
         }
       ],
       "total": "8.75"
      }
      ```

3. **Send** the request. You should receive a JSON response with an id and points.
4. **GET**:
    - URL: http://localhost:8080/api/points/<id_from_post_response>
    - **Send**: You should receive a JSON object with "points": <number>.

Points Calculation Rules

1. **1 point per alphanumeric character in `retailer`**
   - Letters (`A-Z`/`a-z`) and digits (`0-9`) count.
2. **50 points if `total` is a round dollar amount**
   - For example, `total = 100.00` or `50` (no cents).
3. **25 points if `total` is a multiple of 0.25**
   - For example, `1.75`, `2.00`, `2.25`.
4. **5 points for every 2 items**
   - Integer division is used to count pairs.
   - E.g., `3 items = 5 points`, `4 items = 10 points`.
5. **(price * 0.2) rounded up for items with a short Description length multiple of 3**
   - Trim the description before measuring length.
   - Add `ceil(price * 0.2)` points.
6. **6 points if the purchase day is odd**
   - E.g., the 1st, 3rd, 5th, 7th, etc.
7. **10 points if purchase time is between 2:00 PM and 4:00 PM**
   - Strictly after `14:00` and before `16:00`.

Sample Receipts and Expected Points

Example 1

json

```
{
  "retailer": "ABCStore",
  "purchaseDate": "2023-06-03",
  "purchaseTime": "15:10",
  "items": [
    { "shortDescription": "Milk", "price": "2.25" },
    { "shortDescription": "Eggs", "price": "3.75" }
  ],
  "total": "6.00"
}
```

- Retailer "ABCStore" → 8 alphanumeric characters → 8 points
- Total = 6.00 is a round dollar → +50 points
- 6.00 mod 0.25 = 0 → +25 points
- 2 items → (2/2)*5 = 5 points
- "Milk" length = 4 (not multiple of 3), "Eggs" length = 4 (not multiple of 3) → +0

- Purchase day = 3rd (odd) → +6 points
- Purchase time = 15:10 (between 14:00 and 16:00) → +10 points
- **Total** = 8 + 50 + 25 + 5 + 0 + 6 + 10 = **104 points**

Example 2

json

```
{
 "retailer": "X1",
 "purchaseDate": "2023-06-02",
 "purchaseTime": "13:00",
 "items": [
   { "shortDescription": "Toy", "price": "9.99" },
   { "shortDescription": "Ball", "price": "2.00" },
   { "shortDescription": "Car", "price": "5.00" }
 ],
 "total": "16.99"
}
```

- "X1" → 2 alphanumeric characters → +2 points
- 16.99 is not round → +0
- 16.99 mod 0.25 is not 0 → +0
- 3 items → (3/2)*5 = 5 points
- Descriptions:
  - "Toy" length 3 (multiple of 3) → price 9.99 * 0.2 = 1.998 → ceil = 2 points
  - "Ball" length 4 (not multiple of 3) → +0
  - "Car" length 3 (multiple of 3) → price 5.00 * 0.2 = 1.0 → ceil = 1 point
- Purchase day = 2nd (even) → +0
- Purchase time = 13:00 (not between 14:00 & 16:00) → +0
- **Total** = 2 + 0 + 0 + 5 + 2 + 0 + 1 + 0 + 0 = **10 points**

Known Limitations

- **In-memory storage**: The receipt data and points are stored in a Go map. Once the service restarts, all stored receipts and points are lost.
- **UUID generation**: Uses a simplified approach with math/rand. Collisions are unlikely but **not** guaranteed impossible.
- **Error handling**: Date/time parsing errors or malformed prices default to zero values. Additional validation may be required in production.