Fetch’s Receipt Processor Challenge

Problem Link:  
<https://github.com/fetch-rewards/receipt-processor-challenge/tree/main>

This Document contains the following:  
  
1. About the Problem

2. Steps to Run

3. Api Endpoints

4. Architecture and Components

5. Room for Improvements

# About the Problem:

The goal is to build a web service that processes receipts and calculates reward points based on specific rules. The service must expose two endpoints:

**POST /receipts/process:** Accepts a receipt JSON and returns a unique ID.

**GET /receipts/{id}/points:** Returns the number of points awarded for the receipt with the given ID.

Points are awarded based on various receipt attributes, such as the retailer name, purchase time, item descriptions, and total amount. Data persistence is not required beyond application runtime — in-memory storage is sufficient.

# Steps to Run:

**Clone the Repo:**  
git clone <https://github.com/darshan601/Fetch_Receipt_processor.git>

Go into the directory

**Build the Docker image**docker-compose build --no-cache

**Start the application**docker-compose up -d

**The application will be available at:**[**http://localhost:18080**](http://localhost:18080)

This will redirect to the Swagger UI, where you can view and interact with all API endpoints directly.

Alternatively, you can also use Postman to test the APIs.

Stop and remove the container:  
**docker-compose down**

This will shut down the service and clean up all running containers.

API Endpoints  
**POST /receipts/process** — Accepts a receipt JSON and returns a unique receipt ID.  
example:  
{

"retailer": "Target",

"purchaseDate": "2022-01-02",

"purchaseTime": "13:13",

"total": "1.25",

"items": [

{"shortDescription": "Pepsi - 12-oz", "price": "1.25"}

]

}

Returns: { "id": "7fb1377b-b223-49d9-a31a-5a02701dd310" }

**GET /receipts/{id}/points** — Returns the number of points awarded for the specified receipt ID.

Returns: { "points": 32 }

# Architecture and Components

A diagram of a process flow

AI-generated content may be incorrect.

Architecture diagram

Components:

1. Receipts Controller: Exposes the API that can receive POST and GET requests.
2. Validation Service: Contains Validation logic to check for the incoming requests.
3. Receipt Service: Converts Request models into Entity models and Entities to Response models.
4. Receipt Storage: Contains Concurrent Dictionary which stores the string Id and receipts objects, along with the Hash Dictionary for Duplicate Checking.
5. Points Service: Contains the logic for point calculation and aggregates different points and outputs final points
6. Channel/Queue: Unbounded Channel that works as a Message Broker, receipt ids are sent into the channel and later processed by background worker.
7. Receipt Processor Background Worker: consumes the ids from the channel and retrieves the receipts and sends it to the Points Services and then updates and stores the receipt back to Receipt Storage.

Room for Improvements:

* I’ve added a background worker that takes the receipt id from the queues, retries the receipt object from the Storage and then calls the Points Service and after the points service returns the points, the workerservice will update the receipt object and store that into the storage.
* For Checking Duplicates, instead of converting the data back into json and then hashing directly, we can check for the individual details such as Retailer’s name, Purchase Date, Purchase Time,Total, along with item Details(stored in alphabetical order) such as Item Description, Item Amount (All Trimmed); and append all together in a string, and then hash them, and store it into the HashDictionary, for checking duplicates.

But that too will have add some processing overload and can impact Api latency.

* I have stress tested the apis and have received following results:  
  **Load Test Summary (5-minute Run)**

Performed a 5-minute load test simulating up to 5000 virtual users (VUs) against the /receipts endpoint using k6

**Metric | Threshold | Result | Status**

http\_req\_duration | p(95) < 1000 ms | p(95) = 1.69s | ❌ Failed

http\_req\_failed | rate < 0.01 | rate = 0.00% | ✅ Passed

**Metric | Value**

Total HTTP Requests | 481,410

Success Rate | 100% (0 failures)

Average Duration | 487.89 ms

Median Duration | 223.41 ms

Max Duration | 7.66 s

90th Percentile (p90) | 1.31 s

95th Percentile (p95) | 1.69 s

While the API handled all requests successfully (0% failure), the **95th percentile latency exceeded the 1s threshold**, indicating some performance degradation under high load.

* I might’ve missed some points, some coding practices here and there, Pls feel free to review this and let me know about the same.