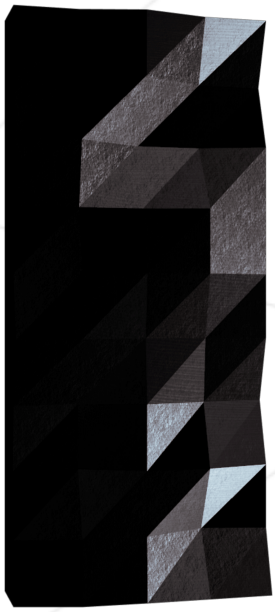


Road to CyberAgent

CA Tech Dojo Go

Summary: このドキュメントは、*Road to CyberAgent @ 42 Tokyo*の*CA Tech Dojo Go*モジュール用の課題である。

ROAD to CyberAgent

42 | 東京

Contents

Chapter I

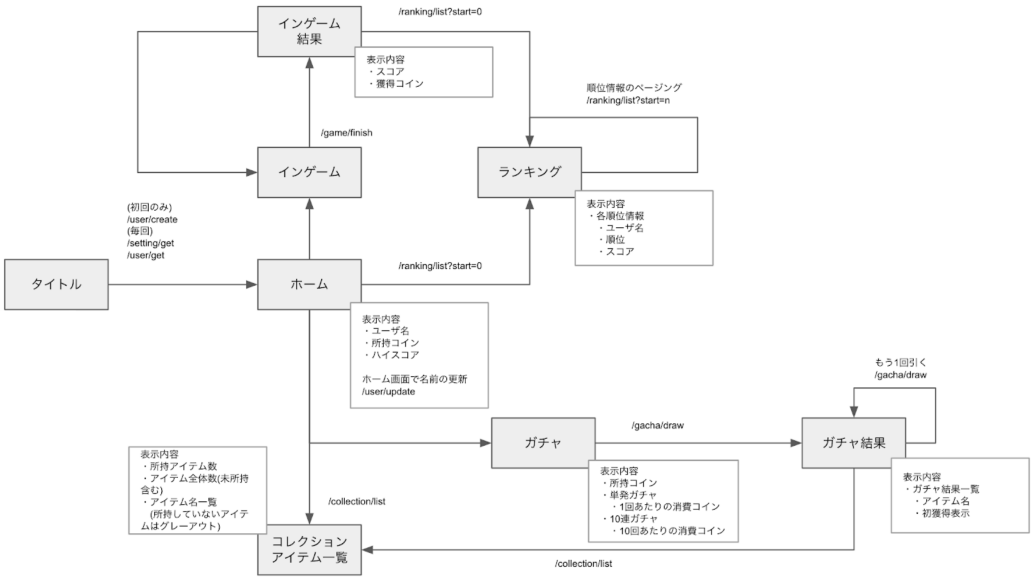
Introduction

Road to CyberAgentではゲームを想定したバックエンドAPIを実装して頂きます。

https://github.com/CyberAgentHack/42tokyo-road-to-dojo-go

想定するゲームは昨今のスマートフォン向けゲームで一般的な
- ランキング
- ガチャ

といった機能を有し、8つのAPIを必要とします。



Chapter II

Instruction

- 課題に関する噂に惑わされないよう気をつけ、信用しないこと。
- この書類は、提出前に変更になる可能性があるため、気をつけること。
- ファイルとディレクトリへの権限があることを、あらかじめ確認すること。
- すべての課題は、提出手順に従い行うこと。
- 課題の確認と評価は、あなたの周りにいる学生により行われる。
- 課題は、簡単なものから徐々に難しくなるように並べられている。
- 課題で指定されていないものは、どんなファイルもディレクトリ内に置かないこと。
- 助けてくれるのは、Google / 人間 / インターネット / ...と呼ばれているものたちである。

Chapter III

Vocabulary

- MySQL
  - DDL(DataDefinitionLanguage):データベースの構造や構成を定義するためのSQL文
  - DML(DataManipulationLanguage):データの管理・操作を定義するためのSQL文
- Data
  - ユーザデータ：ユーザーによって更新が加わるデータ。ユーザーの名前や所持コイン、所持しているコレクションアイテムの情報など。
  - マスターデータ：ユーザーによって更新が加わらないデータ。サービスの運営者などによってのみ変更が加わるデータ。コレクションアイテムの名前、ガチャの内容（排出内容・確率）など。

Chapter IV

Development Environment

開発環境はdocker-composeで構築します。

IV.1 Requirements

- Go
- docker / docker-compose

IV.2 MySQL

MySQLはリレーショナルデータベースの1つです。Road to CyberAgentではMySQLデータベースを主に使用して開発を行います。

IV.2.1 Start

```
$ docker-compose up -d mysql
```

IV.2.2 Re-Initialize

MySQLの初回起動時に db/init/ 以下にあるSQLファイルが実行されます。再度読み込みの際は一度MySQLのvolumeを削除して再度MySQLを起動します。

```
$ docker-compose down
$ docker volume rm 42tokyo-road-to-dojo-go_db-data
$ docker-compose up -d mysql
```

IV.3 phpMyAdmin

phpMyAdminはMySQLの閲覧や操作、管理を行うことができるWebアプリケーションです。開発の際、適宜使用してください。

IV.3.1 Start

```
$ docker-compose up -d phpmyadmin
```

IV.3.2 Browse

http://localhost:4000/

IV.4 Redis

Redisはインメモリデータベースの1つです。必須ではありませんが課題の中でキャッシュやランキングなどの機能でぜひ利用してみましょう。

IV.4.1 Start

```
$ docker-compose up -d redis
```

IV.5 Swagger UI

API使用はSwagger UIを利用して閲覧します。
定義ファイルは api-document.yamlです。

IV.5.1 Start

```
$ docker-compose up -d swagger-ui
```

IV.5.2 Browse

http://localhost:3000/

IV.5.3 API Test

Swagger UIの各API定義の「Try it out」からlocalhost:8080に対して実際にリクエストを行い、APIの動作確認を行うことができます。

IV.6 Shutdown

```
$ docker-compose down
```

IV.7 Supplement

docker-compose upの際、Portの重複が発生すると port is already allocated といったエラーが発生する。 Portの重複が発生した場合はローカル環境で対象のPortを使用しているプロセスがないか確認し停止すること。またはdocker-compose.yamlファイルで定義されているportの設定を変更し回避すること。

Chapter V

Conditions

Road to CyberAgentでは以下の条件で開発を行うこと。

1. Webフレームワークは利用せず、Goの標準パッケージを用いてAPIサーバーを実装すること。

2. データベースへのアクセス処理にgorm や sqlboilerといったO/R Mapperは利用せず、 database/sqlパッケージを利用して自身でSQLを記述し実装を行うこと。

3. その他ライブラリの使用については制限はしません。
   （データベースのDriverやUUIDの生成など）ただし、ゲームロジックに関わる処理は自身で実装するよう努めること。（ランキングやガチャの抽選など）

4. 各APIに記載されている「Check Point」を読み、その内容を考慮した上で実装を行うこと。

5. 開発が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、正常に動作することを確認すること。

Chapter VI

Exercise 01 : テーブル設計

APIの実装に着手する前にこのプロダクトに必要なデータを把握し、データベースのテーブル設計と初期化を行いましょう。

1. API仕様書を読んで必要なデータを把握する。その際、「ユーザーデータ」「マスターデータ」を意識すること。

   Swagger UI: http://localhost:3000

   ```
   $ docker-compose up -d swagger-ui
   ```

2. 必要なデータをテーブル定義に落とし込む。（任意でER図を書いてみましょう）

3. 定義したテーブル定義に基づいてDDLファイルを作成し、db/init/ddl.sql として配置する。

4. コレクションアイテムやガチャの確率定義など、ゲームをプレイするユーザーによって内容が書き換わらないレコードをDMLファイルとして作成し、db/init/dml.sqlとして配置する。

5. MySQLのdocker volumeを一度削除して再起動を行い、3.4.で作成したDLL/DMLファイルを初期化 SQLファイルとして読み込ませる。

Chapter VII

Exercise 02：ユーザー作成APIの実装

以下のエンドポイントを実装しましょう。

- POST /user/create

このステップで新しく行う作業として以下のようなことが挙げられます。

- /pkg/server/handlerの実装
- /pkg/server/server.goへのルーティングの追加
- 後続のAPIの認証で利用するTokenの生成処理の実装（ランダムな文字列やUUIDの生成など簡易的なもので良い）
- MySQLテーブルへのアクセス処理(DAO)の実装

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

Chapter VIII

Exercise 03 : ユーザー情報取得APIの実装

以下のエンドポイントを実装しましょう。

- GET /user/get

このステップで新しく行う作業として以下のようなことが挙げられます。

- リクエストヘッダから認証Tokenを取得し、その認証Tokenをもとにユーザーを一意に特定し取得する認証処理の実装
- 認証で取得したユーザーの情報をcontext.Context を利用してリクエストスコープで格納/取得する処理の実装

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

VIII.1 Check Point

- ユーザーの認証処理がmiddlewareとして実装されているか。 (各APIの実装全てに認証処理が記述されないこと)

  認証のmiddlewareの処理は /pkg/http/middleware/auth.goへ実装をし、以下のように /pkg/server/server.goで各APIへ適用すること。

```
http.HandleFunc("/user/get",
  get(middleware.Authenticate(handler.HandleUserGet())))
http.HandleFunc("/user/update",
  post(middleware.Authenticate(handler.HandleUserUpdate())))
```

Chapter IX

Exercise 04：ユーザー情報更新APIの実装

以下の認証付きエンドポイントを実装しましょう。

- POST /user/update

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

Chapter X

Exercise 05：コレクションアイテム一覧APIの実装

以下のエンドポイントを実装しましょう。

- GET /collection/list

このステップで新しく行う作業として以下のようなことが挙げられます。

- ユーザーの所持コレクションアイテム情報とコレクションアイテムのマスターデータを紐付け加工する処理の実装

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

X.1 Check Point

- n+1問題が発生していないか。
- コレクションアイテムのマスターデータがリクエストの度にMySQLに対しSELECTされていないか。（マスターデータがAPIサーバー起動時にキャッシュされているか）

Chapter XI

Exercise 06：ランキング一覧APIの実装

以下のエンドポイントを実装しましょう。

- GET /ranking/list

このステップで新しく行う作業として以下のようなことが挙げられます。

- MySQLまたはRedisを利用したランキングの実装

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

XI.1 Check Point

- n+1問題が発生していないか。
- リクエストパラメータに不正な値が設定されている場合を考慮できているか。(startパラメータにマイナス値やランキングに登録されているスコア数を超過した値が設定されている場合の考慮)
- 不要なデータの取得を行っていないか。（ユーザーデータの全件取得等）

Chapter XII

Exercise 07：ゲーム終了APIの実装

以下のエンドポイントを実装しましょう。

- POST /game/finish

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

XII.1 Check Point

- リクエストパラメータに不正な値が設定されている場合を考慮できているか。(scoreパラメータにマイナス値が設定されている場合の考慮)

Chapter XIII

Exercise 08 : ガチャ実行APIの実装

以下のエンドポイントを実装しましょう。

- POST /gacha/draw

このステップで新しく行う作業として以下のようなことが挙げられます。

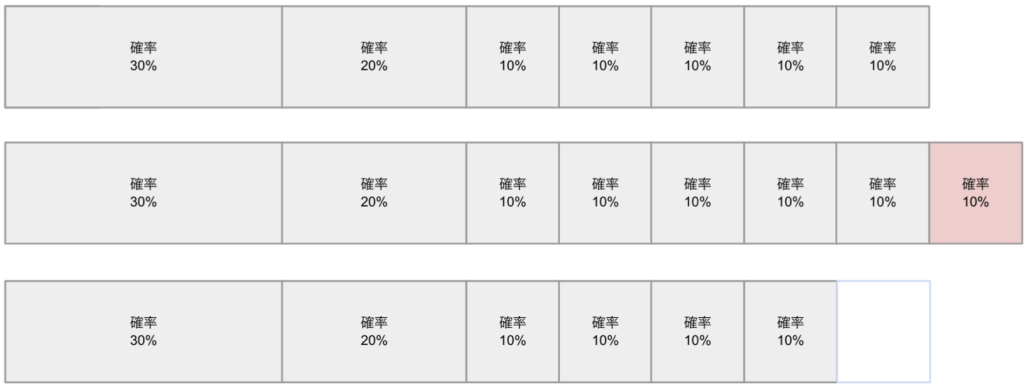
- ガチャの抽選ロジックの実装
- MySQL Transaction処理の実装

実装が完了したらSwagger UIの「Try it out」から実際にリクエストを行い、HTTPステータスコード200で意図したレスポンスが返ってくることを確認すること。

XIII.1 Check Point

- 各コレクションアイテムの排出確率の設定は排出内容のレコード数が変動することを前提に設計をすること。例えば「排出するコレクションアイテムのレコードは必ず100件必要で、その100件のレコード毎に固定の排出確率を設定する」という設計はNG。（レコード数が増減したり、一部のレコードの排出確率の設定を変更することで排出確率の合計が必ず100%にならないという設計はNG）排出内容毎に「重み」を定義し、レコード数が増減しても排出確率の合計が必ず100%になるようにすること。

NG: データに確率自体を定義してしまっている例
データ数の増減で確率の合計が100%であることが保証されなくなってしまう



OK: データに重みを定義している例
データ数が増減しても確率の合計が100%であることが保証されている

- リクエストパラメータに不正な値が設定されている場合を考慮できているか。(timesパラメータにマイナス値やAPIサーバにDBに負担がかかってしまうような大きな値が設定されている場合の考慮)
- n+1問題が発生していないか。
- ガチャ排出確率に関するマスターデータがリクエストのたびにMySQLに対しSELECTされていないか。 (マスターデータがAPIサーバー起動時にキャッシュされているか)
- コレクションアイテムのINSERTクエリが獲得数分後に発行されていないか。(更新負荷がかかる仕組みになってしまうためINSERTクエリはまとめること)
- Transactionを利用して原子性が担保されているか。例えば処理の途中でエラーとなり、「コレクションアイテムが獲得できてコインが消費されていない」といった事象が発生しうる状態はNG。 (処理が途中でエラーになってしまっても中途半端なデータが保存されないこと)

Chapter XIV

Bonus 01: ユニットテストの追加

handlerや各ビジネスロジックににユニットテストを追加してみましょう。

DBへのアクセスを行う処理や、外部のAPIに接続する処理など、インフラに依存するテストコードを書く場合は、処理をモック化することでテストコードの記述を簡単にすることができます。
適度レイヤーを意識したアーキテクチャの考慮も必要となるため、合わせてアプリケーション全体のアーキテクチャについても考えてみましょう。

e.g.)

- レイヤードアーキテクチャ
- クリーンアーキテクチャ

Chapter XV

Bonus 02: ロギング用middlewareの実装

実装のサービスでは様々なログを取得し、データ分析やトラブルシューティングに活用しています。

以下のような内容をそれぞれ別のログファイルとして出力する仕組みをmiddlewareとして実装してみましょう。

- アクセスログ
    - 日時
    - リクエストURL
    - リクエストBody
    - アクセス元IP
    - ユーザーのID
    - HTTP Status Code
- エラーログ
    - 日時
    - リクエストURL
    - リクエストBody
    - ユーザーのID
    - エラー内容