DIGITAL. THINKERS

.NET TEST EXERCISE

OVERVIEW

In this test exercise you need to build a simple web API using a chosen version of .NET.

Your application will simulate a self-checkout machine in a supermarket, which can be restocked, and calculates which bills and coins should it give back as change when used.

The finished project should be uploaded to a repository on GitHub, GitLab, BitBucket or a similar git repository website, and the link sent to us in email.

Please include a very brief guide in the README for building, starting and accessing your application.

The whole exercise should not take more than 3-6 hours to complete.

The application must

- Build and start without errors
- Log basic information to the console
- Be accessible and usable
- Cover the compulsory feature set
- Be committed frequently, with descriptive commit messages, so we understand the development process

It's good if the application

- Has a coherent code style
- Has readable, commented, and easy to understand code
- The algorithms used are neatly written and performant

FEATURES

Compulsory features

These features should be implemented in your application, and should be able to run without error. If you cannot finish with everything, that is no problem, it is better to have some of the features tested and working, than to have all of them, but incomplete.

- The application should have three main endpoints:

`POST /api/v1/Stock`

- The **`Stock`** endpoint should accept a JSON in a POST request, with an object containing the bills and coins to be loaded into the "machine" (HUF).
- The keys should be the numerical values of the bills or coins, the value should be the number of the items inserted. See example object below.
- The app should store the currently available currency in memory.
- The endpoint should return a **`200 OK`** response, with the currently stored items in the response body, or an appropriate error response if an exception occurs.

`GET /api/v1/Stock`

- The endpoint should return a **`200 OK`** response, with the currently stored items in the response body, or an appropriate error response if an exception occurs.

`POST /api/v1/Checkout`

- The endpoint should accept the same object used in the POST request for the **`Stock`** endpoint, but this time the object represents the bills and coins inserted into the machine by the customer during purchase. The JSON should also contain a price field representing the total price of the purchase.
- The purchase, if successful, should update the number of bills and coins stored in the machine accordingly
- The endpoint should return with a
    - **`200 OK`** response, and an object containing the change given back by the machine
    - **`400 Bad Request`** response, with a response body describing the error if the purchase cannot be fulfilled for some reason. Prepare the algorithm for common exceptions and use cases.
- Example usage of the **`Checkout`** endpoint:

```
POST /api/v1/Checkout
{
  "inserted": {
    "1000": 3,
    "500": 1
  },
  "price": 3100
}
```

```
Response: 200 OK
{
  "200": 2,
  "100": 1
}
```

Optional features

If you have some spare time on your hands, or don't mind some extra practice, you may also implement one or more of the following features.

- Use a database to store the currency instead of memory
- Implement a **`/api/v1/BlockedBills`** endpoint. The endpoint should return with an array containing the denominations the machine currently accepts. A bill or coin cannot be accepted if the machine would not be able to give back proper change.
- Prepare the machine for accepting Euros (Checkout endpoint only). Change should always be returned in HUF. The exact details of the implementation are up to you, but please provide a quick run-down of the feature in the README.

Good work, and good luck! :)

The DigitalThinkers Team