ENGAGE CHALLENGE DOCUMENTATION

AUTHOR: Diana Sormani.

INTRODUCTION

This document describes the design, implementation and setup details for the developing task as described by in Engage's GitHub.

The application implements the requested stories 1 to 4, plus a few improvements. Also includes the skeleton of an authentication and authorization system using Basic Authentication, setup and self-signed certificate for HTTPS, and CORS settings to enable the web application to consume the services from another domain.

Different testing strategies were performed and different kinds of documentation are provided.

DESIGN

The whole project as presented has a front end and a back end. In this section, only the back end design will be detailed.

The back end was designed in three layers:

1. The **api** layer which defines the classes that implements the REST APIs published for external use, as well as the required resources or representation classes for exposing this APIs, and the data objects exchanged with clients
   Specifically:
   - The `ExpenseResource` class implements the REST resource for expenses creation and listing.
   - The `ExpenseJSON` class defines the exact structure for the expenses information that the expense resource will produce and also retrieve.
   - The `ApiResult` generic class is the actual result that the resources will return. This class defines whether the transaction was performed successfully, and will include a list of errors or the desired objects.
   - The `ErrorResult` class will contain a specific error code and description produced by the transaction should it failed for some reason. This object will also be included in the `ApiResult`.
   - The `ConverterResource` class implements the rest resource for converting a given amount from euros to United Kingdom pounds.
   - The `VatResource` class implements the rest resource for vat percentage retrieval.

2. The **service** layer which implements the actual logic for the REST APIs to work.

- The class `ExpenseService` implements the `ExpenseResource` logic.
- The class `CurrencyService` implements the `ConverterResource` logic.
- Also, some exception classes were created to handle specific types of errors that can be the transactions results, such as: `InvalidInputException`, `SetupException`, `ObjectNotFoundException`.

3. The **data access** layer which implements the SQL queries required by the rest services, and also the classes required for the data representation.
   - The `ExpenseDAO` interface is created to interact with the DBMS. It provides the required SQL queries to create and retrieve records.
   - The `ExpenseDaoBean` class contains all the information related to the entity “Expense": both the obtained by the front end application as well as the information calculated locally.
   - The `ExpenseMapper` class implements the mapping between the `ExpenseDaoBean` class and the ResultSet from the database.

IMPLEMENTATION

➢ TECHNOLOGIES AND TOOLS USED IN THE SOLUTION

- The back end system was developed with: Dropwizard 1.2.3 (Jersey, Jackson, Jetty, JDBI), Java 9 and Maven.
- The front end: AngularJS, CSS and gulp.
- For version control: GitHub.
- For testing: Postman, JUnit, Mockito.
- For the REST API documentation: Swagger.
- For IDE: IntelliJ.
- RDBMS: MySQL 5.5

➢ REST APIs

The solution provides the following REST APIs:

The VAT API

The class `VatResource` implements a REST endpoint that returns the system Valued Added Tax value. By default, this value is “20”.

This resource was implemented so both the back end and front end can access the VAT value, and so that its value is centralized (as opposed to hard coded all over the system’s code). This value is stored in a class created to store default system parameters: `GeneralSettings`.

It can be accessed via a GET request. For example: https://localhost:8443/vat

The currency API

The class `ExpenseResource` implements a REST endpoint that returns the system conversion rate from EUR to GBP.

This service in turn consumes another REST API to obtain the conversion rate from euros (EUR) to United Kingdom pounds (GBP). The rest endpoint that provides the rate is http://apilayer.net/api/live.

Should the rate provider fail, the currency API will capture the exception and log an error.

It can be accessed via a GET request. For example:
https://localhost:8443/converter/800

The expenses API

The class `ConverterResource` implements a rest endpoint that provides three services.

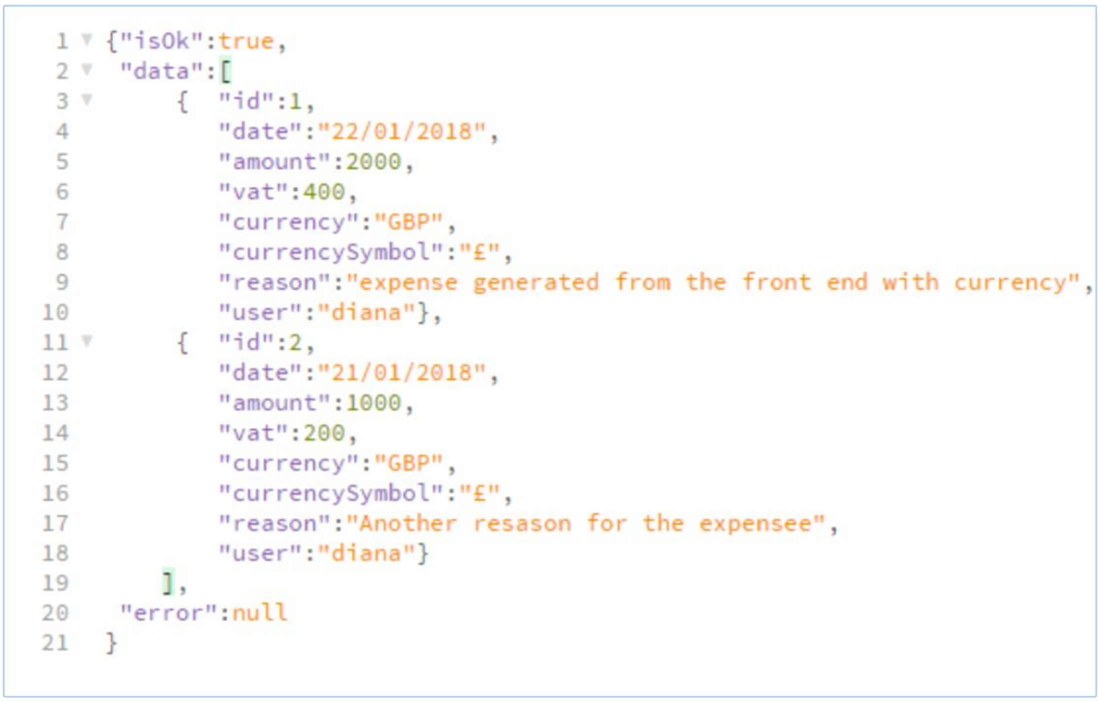
All services provide responses in JSON. More specifically, all services return an object of type `ApiResult` which contains:

- An isOK flag indicating whether the requested transaction was successful
- An `ErrorResult` object which will only be instantiated if an error occurred. This object will indicate an error code, and a human legible description of the problem.
- A `data` member which will contain the information requested.

This `ApiResult` object was created in order to provide a more detailed feedback regarding possible errors which does not match exactly with the HTTP standard error codes. The data member will be returned when the HTTP request has a status 200.

The available methods are:

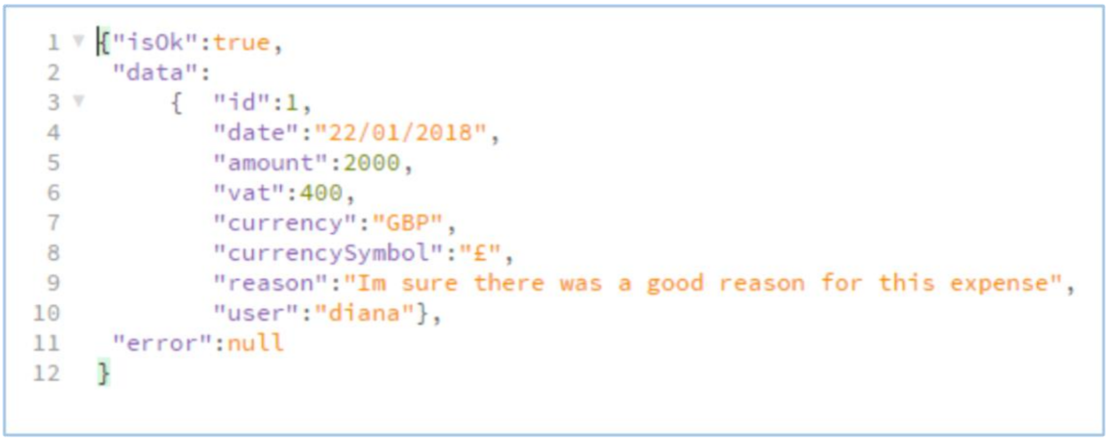
1. Get a list of all the users' expenses: this method will display a list of all the expenses associated with the logged user. The user is extracted from the Basic Authentication HTTP headers.
   This method has no inputs. It can be accessed via an HTTP GET request like:
   https://localhost:8443/expenses/
   - ✓ If the user has no expenses, then the method will respond with an `ApiResult` object containing a flag indicating a successful transaction and an empty `data` member. The HTTP response will also have a status 200.
   - ✓ If the user has associated expenses, then the method will respond with an `ApiResult` object containing a flag indicating a successful transaction and a `data` member object containing a list of expenses (represented as `ExpenseJSON` objects). The HTTP response will also have a status 200. An example of the response would be:



```
{"isOk":true,
 "data":[
     {  "id":1,
        "date":"22/01/2018",
        "amount":2000,
        "vat":400,
        "currency":"GBP",
        "currencySymbol":"£",
        "reason":"expense generated from the front end with currency",
        "user":"diana"},
     {  "id":2,
        "date":"21/01/2018",
        "amount":1000,
        "vat":200,
        "currency":"GBP",
        "currencySymbol":"£",
        "reason":"Another resason for the expensee",
        "user":"diana"}
    ],
 "error":null
}
```

- ✓ If an execution error is detected, then the `ApiResult` will display the code and description in the `error` member.
- ✓ Note: because this structure is slightly different from the structure initially expected by the web app (which didn't accept errors), the web app was updated to work with this structure and display the errors, if any.

2. Get an expense by Id: this method will display an `ExpenseJSON` object with the expense information identified by the parameter {id} sent by the url.
   This method can be accessed via an HTTP GET request like:
   https://localhost:8443/expenses/{id}
   - ✓ If the parameter id is valid (which means that there is an expense identified by that parameter) then the method will respond with an `ApiResult` object containing a flag indicating a successful transaction and a `data` member object containing an `ExpenseJSON` object with all the information. The HTTP response will also have a status 200.
     An example of the response would be:



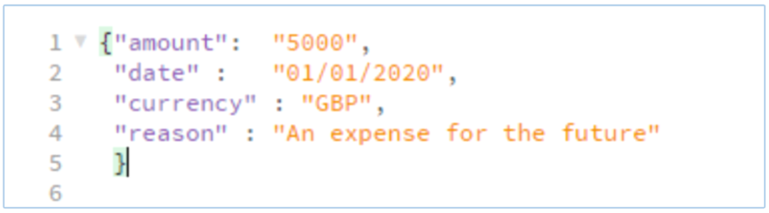
```
{"isOk":true,
 "data":
     {  "id":1,
        "date":"22/01/2018",
        "amount":2000,
        "vat":400,
        "currency":"GBP",
        "currencySymbol":"£",
        "reason":"Im sure there was a good reason for this expense",
        "user":"diana"},
 "error":null
}
```

- ✓ If the parameter id does not match any expense, then the method will return an `ApiResult` object that will display the `error` member with a code “NOT_FOUND” and a description: “The Object identified with: 1000 was not found".

3. **Create an expense**: this method creates a new expense record with the provided information from an HTTP POST form.
   This method has no URL parameters and can be accessed via an HTTP POST request like: https://localhost:8443/expenses/

   This method validates every input. Specifically it controls:
   a. That the required inputs are not null or empty (date, amount and reason)
   b. That the currency has one of the following values: empty, GBP or EUR
   c. That the amount input is actually a number. If the user enters a character other than a digit, an error will be displayed.
   d. That the amount input is positive.
   e. That the user logged is not null.

   - ✓ If the parameter id is valid (which means that there is an expense identified by that parameter) then the method will respond with an `ApiResult` object containing a flag indicating a successful transaction and a `data` member object containing an `ExpenseJSON` object with all the information. The HTTP response will also have a status 200.
     An example of the form would be:



```
{"amount":  "5000",
 "date" :   "01/01/2020",
 "currency" : "GBP",
 "reason" : "An expense for the future"
 }
```

   - ✓ If an execution error is detected, then the `ApiResult` will display the code and description in the `error` member.

➢ Currency Management

As requested, the user can enter the currency in which to express the expense. The currency options are limited to:

- United Kingdom pounds: GBP
- Euros: EUR
- Empty value

If the user enters the “GBP” as the currency code, then system will not translate the expense’s amount nor the VAT.

If the user does not enter a currency code (leaves an empty value), then the system will assume the default currency for the expense, which is GBP.

If the user enters "EUR" currency code, then the system will consume an external endpoint to obtain the conversion rates and with this values will convert the entered expense's amount from euros to pounds.

One implementation consideration: since the system is currently using a free version of the currency rate endpoint, then it does not return directly the conversion rate from EUR to GBP. Instead, it returns the conversion from EUR to US dollars, and the conversion from US dollars to GBP. This is the input the system has to convert to GBP.

Once the conversion is done, then the amount is recalculated and stored in GBP. Obviously, once the amount is converted, then the currency code will be GBP (even if the original was EUR). The VAT amount is now calculated based on the converted amount.

Another important consideration is that both the expense amount and the VAT amount are stored with the appropriate decimal scale. In order to do so, the Java `Currency` class is used. The system define the proper number of decimals for a given currency code.

Similarly, the currency symbol is obtained from the `Currency` class.

If the user enters the currency codes in lower case or some combination of lower and upper, the system will still interpret the currency code only AFTER translated to upper cases.

If the user enters any other different currency code (valid or not to the existing currency codes in the world), they won't be valid for this system.

The currency code was stored in the database for every record. This may seemed unnecessary for the current requirements, but it seems natural to keep the currency in which the values are expressed. And also, this could change in the future.

Possible improvement:

- It could be useful to store the original currency code for tracking purposes.
- Also, the conversion rates could be useful to be kept too.

Data Model

Based on the requirements, the system has one data table: `expenses`. This table has the following structure:

```
( id                      INT
  expense_date            DATETIME
  expense_amount          DECIMAL(18,4)
  expense_amount_VAT      DECIMAL(18,4)
  amount_currency_ISO     CHAR(3),
  expense_reason          VARCHAR(600) ,
  expense_user            VARCHAR(100)
  row_updated_at          TIMESTAMP )
```

The `id` is the table's primary key and it is autonumbered.

The fields `expense_date, expense_amount` and `expense_reason` are inputs from the front end entry.

The `expense_amount_VAT` is calculated based on the `expense_amount` and the VAT percentage retrieved from the application's general settings. Since currently this is always 20% one might be tempted not to store the VAT amount since can always be calculated as that 20%. However, it was included in the table just in case the VAT percentage could change.

The `expense_user` represent the user associated with the expense (for now, it is assumed as the user logged in the front end, registering the expense). This field is important to be able to perform Story 1: list all the expenses from one particular user.

Finally, the `row_updated_at` field is just for tracking purposes. It stores the date and time in which the record was last modified.

Security

CORS

The `ExpenseResource` supports cross origin resource sharing (**CORS**) to allow the front end to interact securely with the REST endpoint.

Authentication

The expenses services are authenticated via HTTP Basic Auth. For that purpose, the necessary HTTP headers are integrated.

The package `security` is created with `BasicUser`, `UserAuthenticator` and `UserAuthorizer` classes which implement the requirement for the HTTP Basic Auth authentication.

In this version, the only validation provided requires to logged user to enter the password "**secret**" to be allowed access, since the integration of a real user control module is beyond the scope of this task.

HTTPS

The expenses services are exposed through the HTTPS protocol only. For this purpose, a self-signed security certificate was created. This certificate can be used for development and it is version controlled.

Log

The application generates structured and readable logs throughout the execution.

Error management

The application captures and generates exceptions in order to identify where the foreseeable errors occur and also to register the type with detailed information. This errors are handled and whenever possible and necessary, are sent back to the user in a readable text.

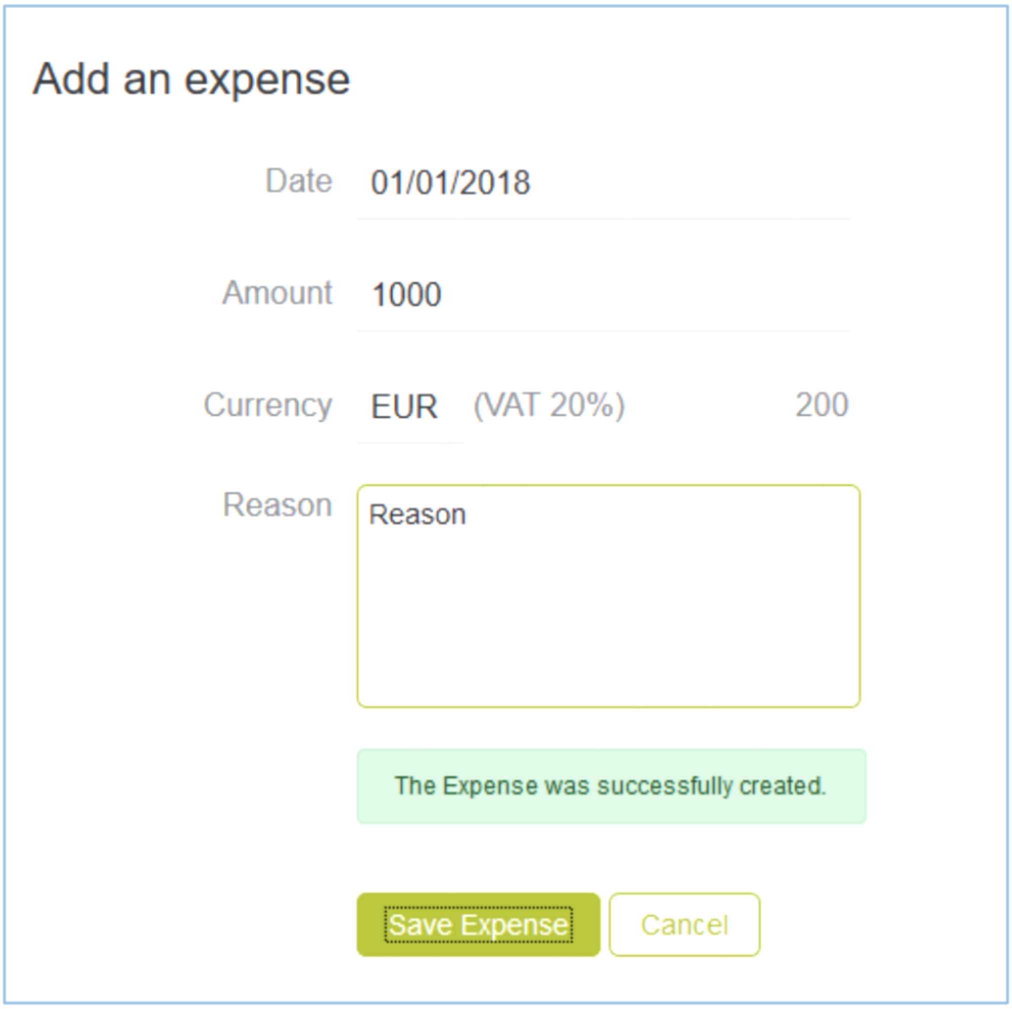
Every time an error is detected, the application also logs the event.

➢ Front End

Form: expenses-content.html

The front end was modified in several aspects in order to implement the requirements.
The form was updated and now shows new information:

1. The percentage that will be used to calculate the VAT.
2. Also the VAT amount must be added in the form for a new expense.
3. The currency in which the expense is expressed.
4. Additionally, the column for the VAT is added to the grid.



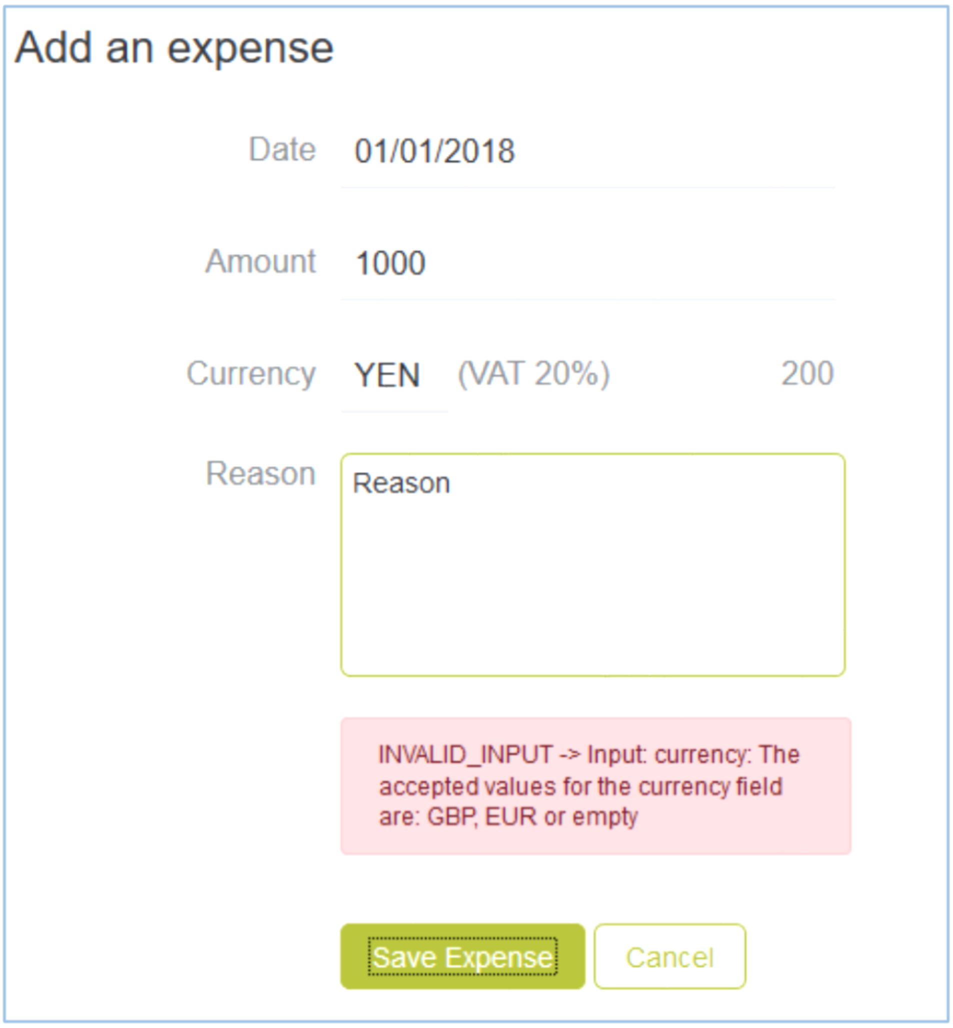
Add an expense

Date	01/01/2018		
Amount	1000		
Currency	EUR	(VAT 20%)	200
Reason	Reason		

The Expense was successfully created.

A new notification was created to indicate whether the expense was actually created or not.

In case of failure, another kind of notification is shown to clearly indicate the reason for failure. In both cases, the notifications disappear after a few seconds (no extra click required).



Add an expense

Date	01/01/2018
Amount	1000
Currency	YEN (VAT 20%) 200
Reason	Reason

INVALID_INPUT -> Input: currency: The accepted values for the currency field are: GBP, EUR or empty

In order to accommodate the new fields without expanding too much the form, and to define the look and feel of the notifications, a new .css file was created: "**alert.css**". This file was included in the "**css.less**" file for import.

Controller: expenses-controller.js

- When the form is loaded, the controller consumes a REST API from the backend that provides the percentage with which the VAT will be calculated. This was implemented this way, so that this default value 20% is not hardcoded all over the application. Ideally it should be obtained from one source, which is the backend. If this should change, then only one place should be changed. This percentage is also displayed.

- Every time the user enters a new amount, the VAT is calculated (or recalculated) and displayed.

- The frontend VAT calculation is only for displaying purposes. The backend always performs its own VAT calculation for storage in the database (and future display).

- Once the form is submitted, the REST API to create a new expense is consumed. If the expense was created, it will be created associated as the user for the expense.

Then an alert is displayed indicating the success of the transaction. Otherwise, another alert message is displayed indicating the reason for failure.

- Once the expense is created it will be displayed in the grid, just as all other expenses that the logged user has created.

- In order to specify the user owner of the expense, the front end should send this user to the back end. Since building of an authentication and authorization module is beyond the scope of this test, then the user at the front end is hardcoded. Only one user is created.

Run method: main.js

- In order to send the logged user to the back end, then the run method from the main.js object is modified: it adds and http header for “Authorization” with the user “diana” and the password “secret” in base64 encoding.

configure.js

- In order to specify where the API end points can be accessed, the `apiroot` is set pointing to the back end: https://localhost:8443/

TESTING

Different strategies of testing have been performed:

➢ SERIALIZATION AND DESERIALIZATION

The class `ExpenseTester` implements tests for serialization and deserialization of the object `ExpenseJSON`, which is the predefined class for exposing the entity “Expense” via the API.

➢ TESTS WITH JUNIT

Tests performed over VAT calculations

The class `ExpenseTestVat` implements unit tests over the VAT value retrieval and the VAT percentage calculation.

1. The first case tests retrieving the VAT value from the general settings of the application.
2. The second case tests calculating the VAT percentage.
3. The third case tests the VAT amount calculation with positive amount.
4. The fourth case tests the VAT amount calculation with a negative amount.
5. The fifth case tests the VAT amount calculation with a zero amount.

All of the above also test the decimal scale in which the vat is specify for a given currency.

Tests performed over Currency management

The class `ExpenseTestCurrency` implements unit tests over the currency management.

1. The first case tests that when creating a new `ExpenseDaoBean` object with the currency code value empty, the system will fill currency code as GBP by default and the currency symbol as *£*.
2. The second case tests that when creating a new `ExpenseDaoBean` object with the currency code value = GBP, the system will fill currency symbol as £ and calculate the vat according to the vat percentage.
3. The third case tests that when creating a new `ExpenseDaoBean` object with the currency code value = EUR, the system will fill currency symbol as and calculate the vat according to the vat percentage (expressed in the new currency).
4. The fourth case tests that when creating a new `ExpenseDaoBean` object with a currency code DIFFERENT than EUR, GBP or empty, the system will throw an exception with code INVALID_CURRENCY and a clear description.

➢ TESTS WITH MOCKITO

The class `ExpenseResourseTest` was created to implement unit testing of the expenses resource. In order to perform unit testing, several mocks were used:

- A mock for the JDBI classes (database connection, handle and DAO)
- A mock for the currency REST API

Tests performed over the /expense resource

1. The first case tests the connection to **/expenses** and checks whether the GET request responds with a 200 status.
2. The second case tests **/expenses/{id}** via a GET request and checks whether it returns an well-formed object with a flag indicating the validity of the response and an "ExpenseJSON" object with ALL the correct information (field by field)
3. The third case tests **/expenses/{id}** with an ID that does not exist (INVALID). It checks that it returns a well-formed object ApiResult<ExpenseJSON> with a flag indicating that the request is ok, but with an empty list.
4. The fourth case tests **/expenses/** via POST for creating an expense with all the correct information.
5. The fifth case tests **/expenses** for a specific user and checks that the number of expenses returned is the correct one, as well as the isOk flag.
6. The sixth case tests **/expenses** for a specific user that has NO expenses and checks the number of expenses returned (should be ZERO) as well as the isOk flag.
7. The seventh case tests **/expenses** via POST for creating an expense with a negative amount.
8. The eighth case tests **/expenses** via POST for creating an expense with a ZERO amount.
9. The ninth tests **/expenses** via POST for creating an expense with an invalid amount which contains characters instead of just digits).

➢ HEALTHCHECK

The class `ExpensesApplicationHealthCheck` class was created to implement system's health checks such as:

- Check the connection to the database.
- Check whether the actual "expense" table exists in the database and all the required attributes also exists.
- Check whether the conversion rate external APIs is working properly.

Should any of the tests fail, then an exception is thrown and handled by the resource. The resource in turn will capture this exceptions and create an appropriate `ApiResult` object indicating the code and description for the failure.

INSTALLATION / EXECUTION

➢ HOW TO INSTALL

The application is available in my GitHub: https://github.com/diasor/backend-coding-challenge

Both the back end and modified front end are included in the same project (although separated by packages).

Once the project is cloned, the back end project should be built using Maven. With the command line in the api-expense directory, run **mvn package**:

```
C:\Users\diana\myGitHub\backend-coding-challenge\api-expense>mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building CodeTest 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
…
```

To start the back end, run the "ExpensesApplication" main method in the resulting fat JAR:

```
C:\Users\diana\myGitHub\backend-coding-challenge\api-expense>java -jar
target\CodeTest-1.0-SNAPSHOT.jar
```

Note: a very extensive list of reflection warnings will pop up, because of the way Swagger scans jars for classes to include in the API doc, combined with the way Dropwizard recommends to deploy the application as a fat JAR. This is a known Swagger issue. The warnings have no negative effect on the application, other than slowing the process initialization a little bit.

To start the front end, run the gulp command as usual.

➢ SQL SCRIPT

As already mentioned, the RDBMS MySQL 5.5 was used. The scripts used to create the database and tables are the following:

CREATE DATABASE **engExpenses**;
USE engExpenses;

```
// creating the table expenses
CREATE TABLE expenses
(
Id INT unsigned NOT NULL AUTO_INCREMENT COMMENT "Unique id for the expenses",
expense_date DATETIME NOT NULL COMMENT "Date and time of the invoice date",
expense_amount DECIMAL(18,4) NOT NULL COMMENT "The amount of the expense",
expense_amount_VAT DECIMAL(18,4) NOT NULL COMMENT "The VAT amount of the
expense",
amount_currency_ISO CHAR(3) NOT NULL COMMENT "Currency in ISO 4217 CHAR CODE",
expense_reason VARCHAR(600) CHARACTER SET utf8mb4 NOT NULL COMMENT "Reason
for the expense",
expense_user VARCHAR(100) NOT NULL COMMENT "User who created the expense",
row_updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    PRIMARY KEY(id)
) DEFAULT CHARSET=latin1 ENGINE=InnoDB;
```

DOCUMENTATION

➢ OBJECT DOCUMENTATION

At the back end, every Java class was documented. Every method (other than getters and setters) were also documented.

At the front end, every modification required for the implementation of the new stories were commented.

The attributes at the database table also have comments.

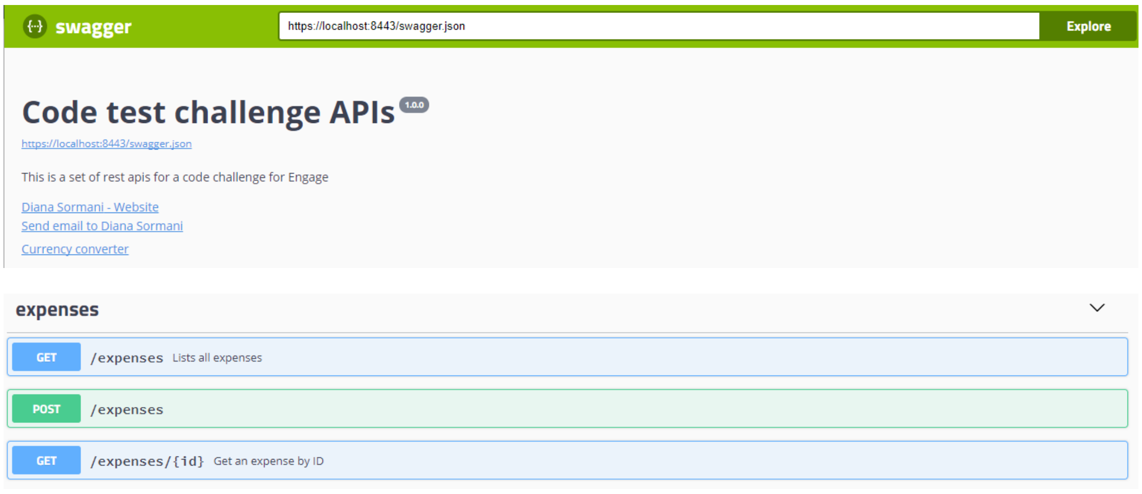
➢ API DOCUMENTATION

The expense REST API was also documented. For this purpose, the Swagger tools were used.

The final API documentation is integrated with the application code itself. The resource classes have annotations that allows the Swagger tool to generate it.

It can be accessed at the application itself by querying https://localhost:8443/swagger.json.

Use this URL with the swagger-ui application (which can be downloaded from Swagger):



Code test challenge APIs 1.0.0

https://localhost:8443/swagger.json

This is a set of rest apis for a code challenge for Engage

Diana Sormani - Website
Send email to Diana Sormani
Currency converter

expenses

GET /expenses Lists all expenses
POST /expenses
GET /expenses/{id} Get an expense by ID

It presents every method available through the resource, its inputs and outputs. As well as its error codes and object models.