Caché Mocking Framework - Demonstration Project

This is a Demonstration Project folder for Caché Mocking framework. This project contains everything necessary to try every feature of this Caché Mocking FW on Docker project with IRIS dataplatform, which allows the same functionality as Caché. For importing framework to Caché see separated repository with only ObjectScript classes and XML importing file. The template for distribution is also on separated repository.

Caché Mocking framework is framework (package) for the Caché intended for mocking simple objects or even complex APIs. After the predefining the mocks and its method, the mock can be called from Caché/IRIS, via Caché/IRIS terminal, through REST API in the range of Caché web server or from anywhere through REST API using Docker technology. The framework is suitable to use in integration, client-server applications or in unit tests.

Instructions for demonstration the features

The suggestion is to work with this project folder in VS Code. There are settings for VS Code which allows to compile ObjectScript classes while the container with IRIS is running.

To start the Docker container with dataplatform IRIS with Caché Mocking framework, put sequence of Docker commands to the bash terminal in project folder location:

```
$ docker-compose build
$ docker-compose up -d
```

You can try to log in to IRIS Managment Portal to easy discover if the container is running:

```
http://localhost:9092/csp/sys/UtilHome.csp?$NAMESPACE=MOCKFW
```

Now you are free to try whatever you want. For inspiration use documentation below.

There is also some demonstration prepared in this project folder **Demonstration** which carries you through the framework features. See sub-folders for more instructions.

Whenever you want to stop container, put Docker commands:

```
$ docker-compose down
```

Features of Caché Mocking Framework

The main class from which can be controlled the mock is **MockFW.MockManager**.

MockFW.MockManager

It allows: **CreateMock()** -- create mock (generate class definition)

- *nameOfTheMock* As %String

```
MOCKFW>do ##class(MockFW.MockManager).CreateMock("MyMock")
```

After creating mock, it is generated the class of the mock in 'MockFW.Mocks' repositary. Now you can add some mocked methods to mock and also get the response which you set up before.

SaveMethod() -- save mocked method to the mock

- *nameOfTheMock* As %String
- *methodName* As %String
- *params* As %String or object
- *response* As %String or object
- *restMethod* As %String ("GET" | "POST" | "PUT" | "DELETE") = ""
- *returnCode* As %Integer = 200
- *delay* As %Integer in seconds = 0
- *force* As %Integer (1 | 0) = 0 -> 1 to force overwrite the same records

```
MOCKFW>do ##class((MockFW.MockManager)).SaveMethod("MyMock", "Method", "
{""name"":""John""}", "return", "POST", 204, 5, 1)
```

Saving the method to the mock can be done also by direcly calling the class of the mock, see bellow.

SaveMethodsFromCSV() -- save multiple mocks and their mocked methods from .csv file

- columns in CSV are the same as parameters for SaveMethod()
- *filePath* As %String -> relative or absolute path to the file

```
MOCKFW>do
##class(MockFW.MockManager).SaveMethodsFromCSV("C:\Users\user\Desktop\mockData.cs
```

GenerateDocumentation() -- generate documentation for certain mock in XML format

- *nameOfTheMock* As %String
- *dirPath* As %String -> directory, where the documentation will be generated
- *inContainer* As %Integer (1 | 0) = 0 -> if inContainer is 1, dirPath is ignored and the file is generated to the Export folder in Docker project folder

```
MOCKFW>do ##class(MockFW.MockManager).GenerateDocumentation("MyMock",
"C:\Users\user\Desktop\")
```

ExportMock() -- export mock for Docker usage

- *nameOfTheMock* As %String
- *dirPath* As %String -> directory, where the files *dataGlobal.gof* and *mockClass.xml* will be generated
- *inContainer* As %Integer (1 | 0) = 0 -> if inContainer is 1, dirPath is ignored and the file is generated to the Export folder in Docker project folder

```
MOCKFW>do ##class(MockFW.MockManager).ExportMock("MyMock",
"C:\Users\user\Desktop")
```

This needs to be done before distribution the mock. See **Instructions for FW user to distribute mock via Docker**

GetAllMocks() -- return all mocks

```
MOCKFW>do ##class(MockFW.MockManager).GetAllMocks()
```

GetAllMethods() -- return all methods of certain mock

- *nameOfTheMock* As %String

```
MOCKFW>do ##class(MockFW.MockManager).GetAllMethods("MyMock")
```

DeleteMethod() -- delete specific method from certain mock

- *nameOfTheMock* As %String
- *methodName* As %String

```
MOCKFW>do ##class(MockFW.MockManager).DeleteMethod("MyMock", "Method")
```

DeleteMethod() -- delete specific method from certain mock with selected parameters

- *nameOfTheMock* As %String
- *methodName* As %String
- *params* As %String or object
- *restMethod* As %String ("GET" | "POST" | "PUT" | "DELETE") = ""

```
MOCKFW>do ##class(MockFW.MockManager).DeleteMethodWithParameter("MyMock",
"Method")
```

DeleteMock() -- delete certain mock (data global and class definition)

- *nameOfTheMock* As %String

```
MOCKFW>do ##class(MockFW.MockManager).DeleteMock("MyMock")
```

CleanAll() -- delete all stored framework data

```
MOCKFW>do ##class(MockFW.MockManager).CleanAll()
```

MockFW.Mocks.*NameOfTheMock*

The generated class definition can be called in order to get predefined response. Also, it can be saved the mock method directly with this class.

SaveMethod() -- save mocked method to the class mock

- *methodName* As %String
- *params* As %String or object
- *response* As %String or object
- *restMethod* As %String ("GET" | "POST" | "PUT" | "DELETE") = ""
- *returnCode* As %Integer = 200
- *delay* As %Integer in seconds = 0
- *force* As %Integer (1 | 0) = 0 -> 1 to force overwrite the same records

```
MOCKFW>do ##class(MockFW.Mocks.MyMock).SaveMethod("Method", "", "return",
"GET", 404)
```

DispatchClassMethod() - call certain method on specific mock class

- *params* As %String or object

```
MOCKFW>do ##class(MockFW.Mocks.MyMock).Method("{""name"":""John""}")
```

DispatchMethod() - call certain method on specific **instance** of mock class

- *params* As %String or object

```
MOCKFW>set mock = ##class(MockFW.Mocks.MyMock).%New()
MOCKFW>do mock.Method("param")
```

Working with the obtained mock (Docker project folder)

As a mock user in Docker you have two options how you can obtain the mock:

1. Obtain the mock via compressed folder Setup:
    - unwrap the project folder
    - open the project directory in terminal
    - build docker image and run the container

      ```
      $ docker-compose build
      $ docker-compose up -d
      ```

      Let the Docker start the app. By default, the container runs on port 9092 (can be changed in docker-compose.yml):
    - use credential to log in -> username: mockuser, password:12345
    - call the mock

      ```
      http://localhost:9092/api/mocks/MyMock/MethodUrl
      ```

    - launch Management Portal

      ```
      http://localhost:9092/csp/sys/UtilHome.csp
      ```

      The mock can be called from a terminal. To launch the IRIS terminal:

      ```
      $ docker exec -it onlymock iris session IRIS
      $ USER>zn "MOCKFW"
      ...
      MOCKFW>h  #to exit from terminal
      ```

      If you wish to add some changes to the mock, you can, just keep in mind that you have to export the edited mock before the container stop to relaunch the container with changes. Look for the ExportMock() function and copy exported file from folder Export to folder src/ImportData:
    - **ExportMock()** -- export mock for Docker usage
        - *nameOfTheMock* As %String
        - *dirPath* As %String -> directory, where the files *dataGlobal.gof* and *mockClass.xml* will be generated
        - *inContainer* As %Integer (1 | 0) = 0 -> if inContainer is 1, dirPath is ignored and the file is generated to the Export folder in Docker project folder

          ```
          MOCKFW>do ##class(MockFW.MockManager).ExportMock("MyMock", "", 1)
          ```

2. Download docker image from DockerHub
    - you just need to run the Docker command with concrete image, which you received:

      ```
      $ docker run --name onlymock -d --publish 9091:51773 --publish
      9092:52773 mattuz/mockingfw:0.2
      ```

      The app listen on port 9092 (or whatever is in command above), call the mock for example via Postman app.
    - use credential to log in -> username: mockuser, password:12345
    - call the mock

      ```
      http://localhost:9092/api/mocks/MyMock/MethodUrl
      ```

      But beware, this approach does not allow to save changes to the mock for later usage. Also, this requires almost double downloaded amount of the data.

Look at the dockbook documentation to see all mocked methods in the mock!

Instructions for FW user to distribute mock via Docker

1. To distribute mock via Docker, first it is neccessary to prepare template directory of the project from the git.

   ```
   $ git clone https://github.com/xtuzil/CacheMockingFW-DockerIRIS-template-for-
   distribution  # or pull
   ```

2. Export mock data from Caché:
    - *nameOfTheMock* As %String
    - *dirPath* As %String -> directory, where the files *dataGlobal.gof* and *mockClass.xml* will be generated
    - *inContainer* As %Integer (1 | 0) = 0 -> if inContainer is 1, dirPath is ignored and the file is generated to the Export folder in Docker project folder

      ```
      MOCKFW>do ##class(MockFW.MockManager).ExportMock("MyMock",
      "C:\Users\user\Desktop")
      ```

3. Copy exported file *dataGlobal.gof* and optionally *mockClass.xml* (necessary only for calling the mock from IRIS terminal) to the project folder **src/ImportData**

Then, there are two option to distribute the mock:

a) Send straightaway the directory to mock user

4. Send the compressed directory of the mock project to the user. Then the user has to build image from project folder.

   ```
     e.g. $ zip -r MyMock.zip MockFW
   ```

- this approach **allows** user to call the mock from IRIS terminal an also to edit the distributed mock

b) Build the container and push it to Docker hub. The user will launch the mock with one command.

- This needs to have account at https://hub.docker.com and to create repositary there.

4. Build the image

```
$ docker-compose build
```

5. Then rename the image (tag the image) by finding the container ID or name (using **docker ps**).

```
$ docker tag mock1 myrepository/imagename:version
```

6. Now, push the image to the registry using the image ID.

```
$ docker push myrepository/imagename:version
```

7. Send the name of tag to the user. He can run the container only by one docker command

- this approach allow user to call the mock from IRIS terminal an also to edit the distributed mock but **does not allow** to save changes to the mock for later usage!

@Matěj Tužil 2020