HTEC Champions League API

Documentation

Technical overview

* Given task was implemented as a microservice using Web API project type.
* .Net Core 2.1 and ASP.Net Core 2.1 frameworks were used.
* IDE that was used was Visual Studio 2017 CE.
* As data storage ‘In Memory’ storage type was selected, Entity Framework Core 2.1 was used for this purpose.
* For Unit Testing, xUnit runner was used and Moq library for creation of mock objects.
* For automatic JSON to Object Model conversion and vice-versa, the default Newtonsoft JSON library was used.
* To run both the service, and unit tests for it, it is necessary to install Microsoft .Net Core SDK 2.1.
* Postman app was used to manually test the implemented API against various input data.
* Microservice was implemented as a single project with models, DB context and business logic all-in-one.

API

REST API was implemented with six different endpoints:

* api/results
* GET – Returns current state of Groups. Specific Groups can be requested by sending a ‘groups’ query parameter. Parameter is a single comma delimited string, i.e. “A,B,C”
* PUT – Edits scores for existing Matches. Arbitrary number of score updates can be passed with the request within the requests body. One score update is defined with ‘matchId’ and ‘updatedScore’ parameters. Request return Ok response in case all the updates are successfully processed. Otherwise Bad Request response is sent.
* POST – Adds arbitrary number of Matches to the storage and returns updated state of Groups. For the sake of data consistency, all Matches must be valid, and follow the Champions League rules\* otherwise no match will be saved within the storage. List of Matches is sent within the body of the request. In case at least one of the sent Matches is not valid or Match cannot be added to the group, Bad Request is returned. Otherwise Ok response is returned, with updated state of all the groups.
* api/matches
* GET – Returns set of Matches that correspond to the given filter parameter. Specific filters are defined through query parameters:
* from – Date from when returned matches should date
* to – Date up to when returned matches should date
* group – Name of the Group matches should belong to
* team – Name of the team whose matches should be returned

None of the parameters is mandatory. If no parameters are sent, all matches are returned. Filter parameters that are sent are logically ‘AND’-ed.

* api/results/id
* GET – Returns desired instance of the Match if found in storage. Return Not Found result in case wrong id is sent.
* api/results/delete/id
* DELETE – Deletes Match entry with given Id from the storage if such entry could be found. If successfully deleted, returns deleted Match as response, if not, return Not Found result

\* - Basic rules and validation that are implemented:

* One group can consist of 4 teams.
* One ‘Match day’ can have only two Matches with all different teams

All response from the API a JSON based messages. Also, all parameters that are pulled from the body of the request are expected to be in the JSON format.

Further development:

* **Componentize** - Although currently a ‘small’ project it has potential to grow. Therefore, a good idea would be to create several projects out of it: one representing the service itself, with all the business logic, other one with common interfaces and model classes and potentially a third one with implementation of data storage, so different implementation could be used more easily.
* **Async Requests** - Make all the endpoints asynchronous. Current example is simple with no much data within the storage. Additionally, the storage is ‘In Memory’ so everything runs with satisfactory performance. In the future, with real DB and much more data in it, requests and read and writes to the storage could slow down. Asynchronous methods could help prevent usage of too many request threads while data is being written or read.
* **API Help page -** Automatize creation of API documentation by using Swagger with Swashbuckle NuGet package.
* **Validations** - Implement additional rules and validations. Current implementation consists of only basic set of validation which should suffice to support basic data consistency and validity.
* **Error Messaging** - Better error messages. Current implementation uses Argument Exceptions with different text messages depicting what went wrong. Possibly, users of the API could benefit from more different and more informant messages.
* **Testing** - Add integration or end-to-end test. Automatized tests of the whole service and its API is always a good idea :)