**MythTechTest.API**

Access point:

* API Access Address - <http://localhost:5069>

Architecture:

* Entity Framework Core – Data access with SQL Server (tested with SSMS and SQL Express 2022)
* Repo Pattern – Follows Entity Framework using data migrations, separating data access logic
* DTOs – Separate models for API responses and entities
* Async / wait – Used for non blocking operations

Endpoints:

Endpoints followed spec and only have one access point – to search by available ID. This can be accessed through:

* /api/SportsEvents/{id} – refer to MythTechTest.http to see example request

Ingestion:

Data ingested through adding enpoint to appsettings.json, then called through Program.cs (refer to both files)

Handling:

* JSON Array: To store variable length arrays as JSON, seen in source data through “state”, “related sport events”
* Ingestion contains logic to check if record exists, if does then can perform update rather than include duplicate
* Error handling: Comprehensive error handling with appropriate messages

Design:

* API follows RESTful practice with predictable endpoints accessed through standard URI’s and used nouns
* Swagger Testing: Included swagger for clean UI for testing, can also use HTTP file as I’ve included a sample request
* Returned Data: Looking at the data I chose the most meaningful data points, included information that is relevant to the request and omitted fields that are often null or provide little information without appropriate context
* SQL Access: Included a connection string to my machine, leaving this in to show the proper access string but please replace to see DB configuration.

Testing:

* Due to time constraints I tested mainly in the .http file, I tested positive scenarios, failing, edge cases, different HTTP methods and included a test to see if the API is alive
* Normally I would setup a testing suite, including mock repos to test more thoroughly

Advanced Considerations:

* Caching – Included systems like Redis for popular event reading to reduce performance hit on reads, incase of high input
* Database optimization:
* Indexing – composite indexes on key entities like SportId
* Read replicas – separate out read / write for scaling
* Security:
* Implement token for API access
* Role based authentication – Only admins can perform CRUD operations
* Performance:
* Rate limiting incase of heavy traffic
* Health:
* Implement health checks that show db connectivity and system resources
* Metrics to show request duration, success / failures, db performance