**Web API Unit Tests**

**Controller class overview**

I’m going to perform unit tests on Hurling Web API controllers. First I will overview their responsibilities in the application design.

Controllers are responsible for routing http requests to asynchronous methods. For example HTTP GET request for **api/positions/id/1** resource will execute following controller method with number 1 passed as id method parameter *(Figure 1)*



Figure GetPositionById(int id) method

This method and for that matter every controller method will perform the following operations:

* Try to find requested position in repository. Repository is Entity Frameworks interface to actual database.
* If not found it will return HTTP **NotFound** response
* Othervise it will create new DTO object from found position and return HTTP **Ok** response

This is a simplest form of a controller method. Most of the controllers must perform some checking based on Fantasy Hurling game rules. For example if there is a request to DELETE a player from repository, the controller method must first check if the player is not registered in any teams.

Most of the controller methods are capable of creating HTTP responses other than **OK** or **NotFound**, for example above mentioned DELETE request will result in HTTP **Conflict** response with custom text message in response body.

To unit test Web Api controllers I will use Unit Testing Tool available in Visual Studio 2013. It’s similar to Junit in Java ecosystem where I will write annotated test classes and test methods which can be then run from Visual Studio Menu and results are display in a side pane.

First step is to create a Test Project in Hurling Api Solution *(Figure 2)*. This action will generate a test project scaffolding with first test class template.

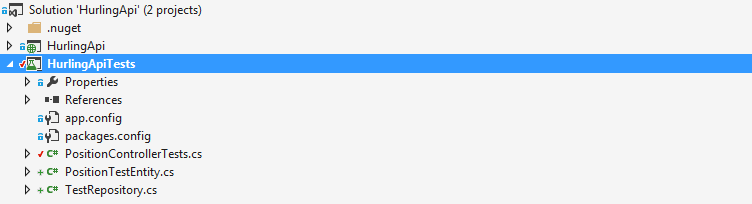


Figure HurlingApiTests Project

A common pattern in unit tests is "arrange-act-assert"(Wasson 2014):

* Arrange: Set up any prerequisites for the test to run.
* Act: Perform the test.
* Assert: Verify that the test succeeded.

Every controller is dependent on a repository for the resources look-up. To mock an actual repository in unit tests I must first add a custom constructor to wire the mock repository to the constructor *(Figure 3).*



Figure PositionController constructors

Now I can create a mock repository for testing purposes. The repository will implement all the interfaces as the real one, but will contain only a small amount of data. In Hurling API implementation I decided to create abstraction for a repository *(Figure 4).*



Figure IRepository interface

This have a number of methods which will return other abstraction for actual database tables called *IEntity* *(Figure 5).*



Figure IEntity interface

*IEntity* interface is an abstraction to actual CRUD (create, read, update, remove) SQL queries. So for example *FindSingleAsync* will return a table row based on given Boolean function. Notice that *IEntity* is a generic interface and forces asynchronous implementation for all the methods. *SaveChangesAsync* method is a proxy method to Entity Framework support for Database transactions. In case of updating, removing and inserting all changes must be saved or rolled back in a case of an error.

As mentioned above for testing purposes I will implement mock *IRepository* for overall testing project with mock *IEntities* for each controller. Let’s do mock repository first *(Figure 6)*.



Figure TestRepository implementation

Let’s say the first controller to unit test is *PositionController*. *IEntity* Positions() called *PositionTestEntity* must be created first. Rest of the *TestRepository* methods are left unimplemented for now. In *PositionTestEntity* I will implement each asynchronous method as testing progresses. First step is to test controller method routed to **api/positions.** This resource is routed to *PositionsController.GetPositions()* *(Figure 7)*



Figure GetPositions() method

*GetPositions()* method is calling Positions().*GetAllAsync()* of wired repository. That’s the first method I want to implement in *PositionTestEntity* mock repository entity *(Figure 8).*



Figure GetAllAsync() method

*GetAllAsync()* method will simply return a list of two positions asynchronously. At this point I finally ready to write first unit test on *PositionsController. (Figure 9)*



Figure GetPositions\_ShouldReturnAllPositions() unit test method

After running the test Visual Studio displayed the result *(Figure 10).*

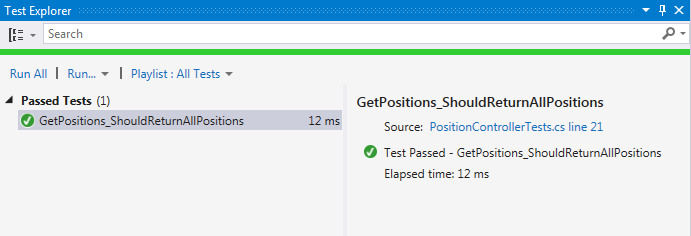


Figure unit test results

First unit test passed. *PossitionsController.GetPossitions()* returned the same resources that *TestRepository.Positions().GetAllAsync()* method. We cannot assume yet that the controller method is implemented without a bug. According to (Wasson 2014) these are the things I must test every controller method for:

* The action returns the correct type of response.
* Invalid parameters return the correct error response.
* The action calls the correct method on the repository or service layer.
* If the response includes a domain model, verify the model type.

**Controller unit testing workflow overview.**

In the section above I outlined my approach for unit testing of Hurling API controllers. I showcased the partial test applied to one method from one controller. In this section I will summarize what have to be done to properly test controllers. Each of the steps below must be applied to all controller methods for all controllers.

1. Create a controller custom constructor to be wired with any repository abstraction.
2. Create implementation for a method in *IEntity* mock implementation.
3. Wire it up in *IRepository* implementation.
4. Wire up a controller with mock *IRepository* implementation in test class.
5. In test class create test methods to test various corner cases on a controller method.

**References**

Mike Wasson. (2014). *Unit Testing Controllers in ASP.NET Web API 2.*Available: http://www.asp.net/web-api/overview/testing-and-debugging/unit-testing-controllers-in-web-api. Last accessed 22th April 2015.