**Implementation**

**Methodology**

As explained in design section, we choose to implement this application using Prototyping methodology. This approach allowed us to build a working prototype pretty early in the process. Because our application is a collaboration of many different web technologies, first step was to make sure we can actually make them work together on something remotely resembling fantasy sport game. If that’s the case we would then mutate the working prototype toward the final project objective. During the each iteration we will add a new functionality or improve the existing one. Because this is a mutli-tier web application with a pipeline where data flow from persistent storage through server toward the clients and vice versa, we must simultaneously improve each tier.

**Prototype (Version 1, November 2014)**

1. A simple database with a few mock tables and simple relations between them was created in Microsoft SQL Server Compact on localhost.
2. Connection to database was assured.
3. New .Net Web API 2 with Entity Framework project was created in Visual Studio 2013 with a connection string to mock database.
4. One ORM Entity and Web API Controller with GET, POST, DELETE and UPDATE methods was created.
5. URLs to resources were crafted accordingly to a fantasy game like logic (Figure 1). The server will return JSON representation of requested table rows in the case of GET request or an appropriate http response for POST, DELETE, UPDATE request. 

Figure 1 URLs to resources

1. Application was tested until worked
2. Simple HTML and Javascript client which would consume the API was created.
3. JQuery Ajax function for GET, POST, UPDATE, DELETE requests was crafted, tested and refactored to acceptable form(Figure 2)



Figure 2 JQuery Ajax Request Method

**Prototype Version 1 Summary**

* a “working prototype” of an AJAX application consuming REST like web service resources from the persistent storage

We knew we can do this on localhost. Next step was to make sure we can deploy this on internet.

**Prototype (Version 2, December 2014)**

1. SQL Server was created on Azure Cloud North Europe datacentre and the localhost mock database was migrated there. (Figure 3)

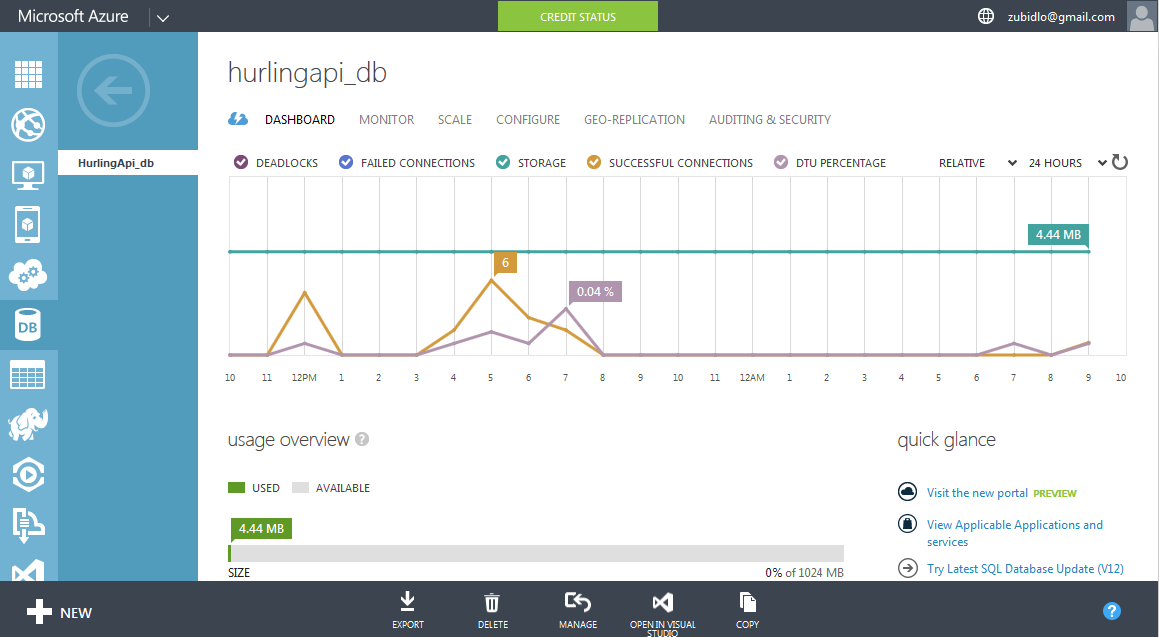


Figure 3 HulingApi SQL server Azure

1. Visual Studio Web API project was configured with a new connection string to the database on the cloud.
2. A web server was created on Azure Cloud North Europe datacentre and Visual Studio was configured to publish Web API there. (Figure 4)

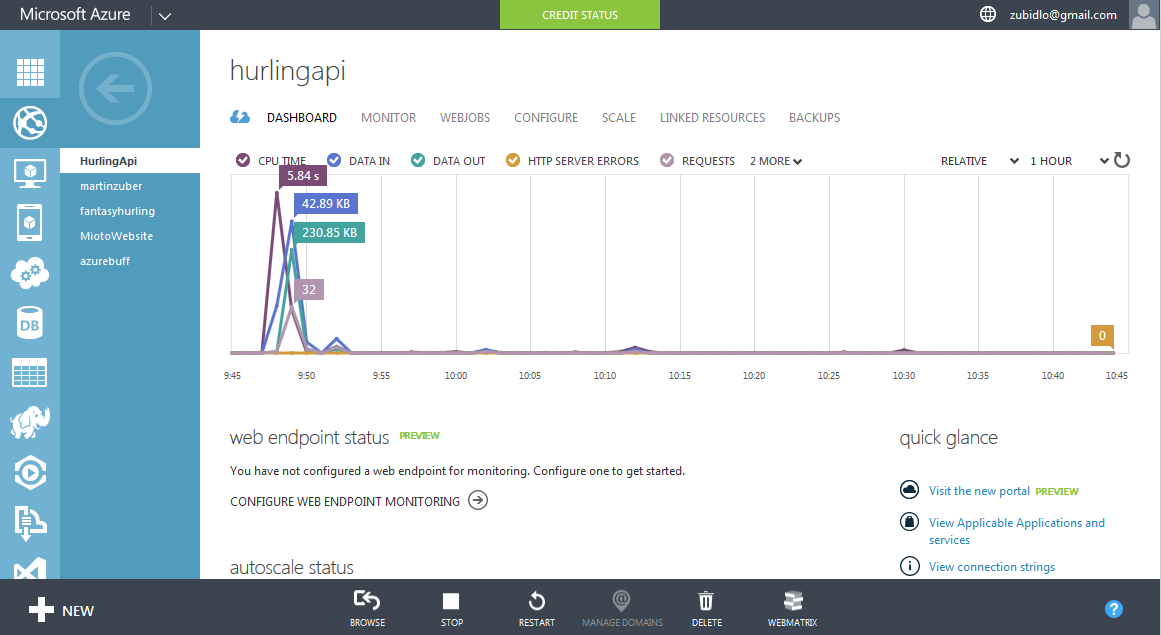


Figure 4 HurlingApi Web server Azure

1. Deployment was tested.
2. Simple web forms and tables were created for our client. At this point we were turning the client into “database admin pages”. These will serve for populating the tables of the final version of fantasy hurling database.
3. At this point we faced first serious problem. Our application turned into Cross-Origin resource sharing (CORS) application. When the client running on localhost (Domain 1) requests the resources from Web API on Azure (Domain 2) the browser will deny the request.
4. After the research on CORS in .Net Web API, the CORS were allowed for each resource route. (Figure 5)



Figure 5 CORS enabled for api/teams route

**Prototype Version 2 Summary**

* REST like web service deployed on the web
* simple admin pages client running on localhost

It was a time to start to turn this into ‘fantasy hurling’ application

**Prototype (Version 3, December 2014)**

1. With fairly good idea how ‘fantasy hurling’ should work, the database relations were crafted. It turned out there must be a many-to-many relationship between team and player entities, which added additional complexity to the project. But overall we design the database schema well and we never needed to change it in the future except for some cosmetic changes like renaming or adding the columns. So at this point we had final version of database deployed on Azure. (Figure 6)

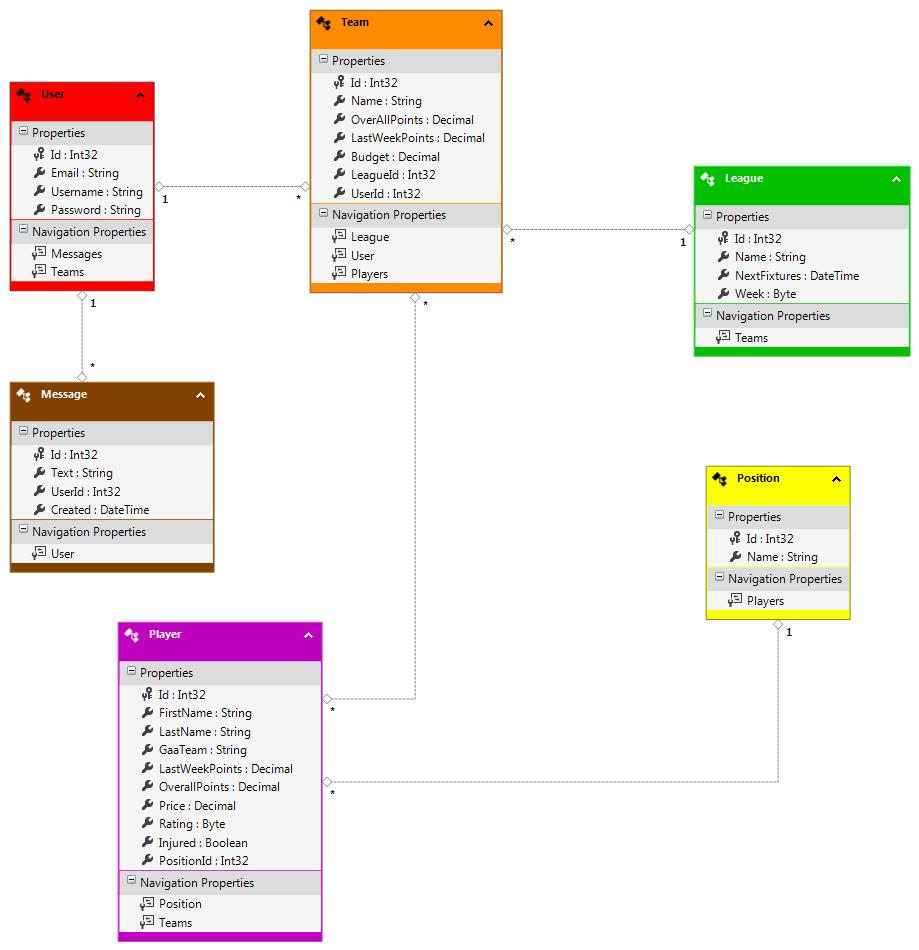


Figure 6 Fantasy Hurling Database Schema

1. Now the development process turned into the circles of
   1. Populate a database table with few rows of data.
   2. Mapping the table columns to plain old c# object POCO object with Entity Framework which is an Object-Relational Mapper (ORM)
   3. Writing the controller which would route the requests to POJO object or collection of POJO objects and return custom crafted http response with JSON representation of POJO object in the http message body.
   4. Creating client ‘data admin page’ for each database table. (Figure 7)

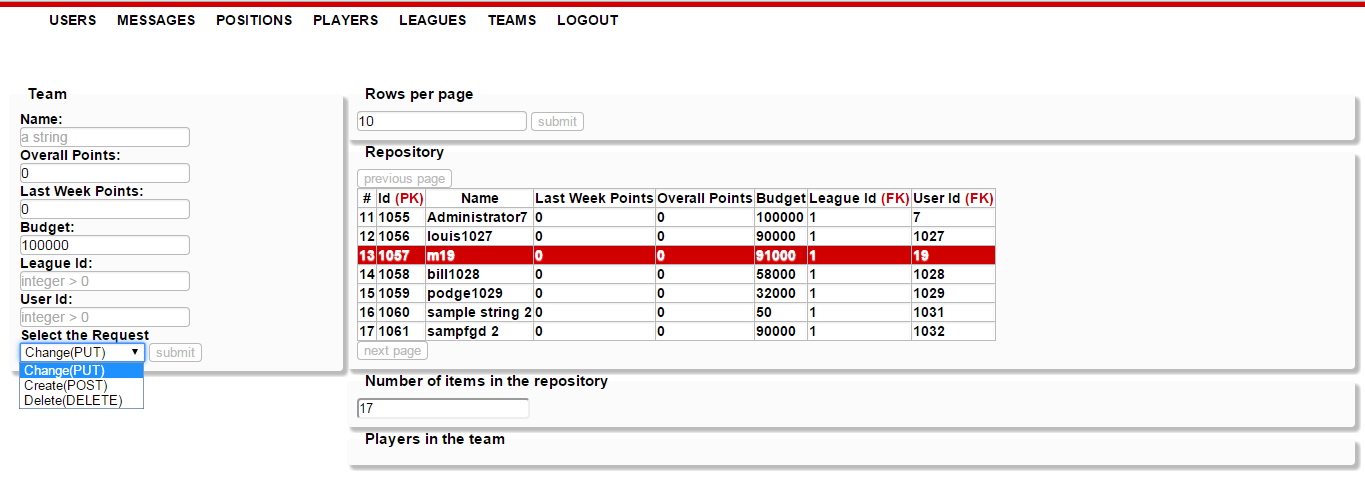


Figure 7 Admin Page for Teams

1. There is 6 tables in our database so I needed to apply steps mentioned above for each and with each new iteration of the process I learned how to do it more efficient and code it more nicely. I faced a number of challenges during this process mainly because each step uses different languages and tools.
   1. SQL insert queries
   2. Entity Framework with C# with design patterns such as Abstract Factory, Repository. Visual Studio 2013
   3. C# Web API 2 Framework with C#, LINQ C# extension, Visual Studio 2013
   4. HTML5 , CSS3, Javascript, JQuery, HTTP protocol
2. It became apparent that I will need to implement some kind of ‘business logic’ to the controllers in later prototype. For example if an administrator using admin pages client would like to delete an player which is registered in one or more teams the Entity Framework will throw exception with 3 page long stack trace. This exception will be then inserted into http response body and send back to client. So I will need to check for all such a cases and create custom http responses like *“you cannot delete this player because he is registered with one or more teams”*
3. Other issue was ‘cyclic entity references’ when Entity framework will parse never ending data to JSON. For example a team has references to players and those have references to teams they play for and those have references to same players again and so on. This issue was handled by one more layer of object mapping where POCO entities are mapped to much simpler Data Transfer Objects (DTOs) and those are parsed to JSON and send to client.

**Prototype Version 3 Summary**

* Database in final form deployed on the web.
* Web API working prototype deployed on the web.
* Data admin pages prototype on localhost.

**Prototype (Version 4, January 2015)**

1. Custom http action results were created need for business logic. (Figure 8)



Figure 8 ConflictActionResult

1. These results will be returned by controllers instead of internal server error 500 with a stack trace. (Figure 9)



Figure 9 Conflict Result in code and action

1. Controller methods were turn into Asynchronous Tasks. This approach will not block the server thread.(Figure 10)



Figure 10 asynchronous method example

1. Web API 2 framework supports Open Data Protocol (OData). A controller methods can return OData query able collections. OData is powerful and well-crafted protocol which adds a lot of functionality for the web service. For example sorting, grouping, conditions, server side paging and much more. (Figure 11)



Figure 11 HurlingApi OData support

1. Database admin pages client was finished with highlight-able, click-able and page-able tables. Simple login dialog was created. Admin pages were deployed on new Azure web server.(Figure 12)



Figure 12 fantasyhurling Web Server dashboard Azure

**Prototype Version 4 Summary**

* Database in final form deployed on the web.
* Asynchronous Web API with OData support prototype deployed on the web.
* Database Admin Pages in final form deployed on the web.

**Prototype (Version 5, January 2015)**

1. With Database Admin pages deployed we started to populate database tables.
2. WebAPI business logic was finished and testing and debugging process on running service started. At this point we had fairly well working Web service, which could be considered as final product when properly debugged. There are some design decision which should be reconsidered: I decided to tread following as additional functionalities and will be implemented only if time allows.
   1. The business logic should be pushed one level lower to the repository dbContext. That way the controllers would be decoupled and could be tested more easily with help of Dependency Injection pattern.
   2. Instead of manually mapping relation Entities to DTOs, an auto-mapper tool could be used.
   3. There is a question of authentication. Web API 2 supports various methods over HTTP and HTTPS, OAuth2 protocol included. After some initial research I decided to leave the Web service without the authentication, because it would introduce the complexity we are as a team not equipped to handle just yet. The web page clients will have their own authentication mechanism implemented using HTLM5 session storage.
3. Web API have a default home page client with automatically generated help page. I decided to add ‘API Test’ client to it, which can be imported from NuGet package manager in Visual Studio 2013. This client will allow the front end developers to see how craft the requests toward the API. (Figure 13)

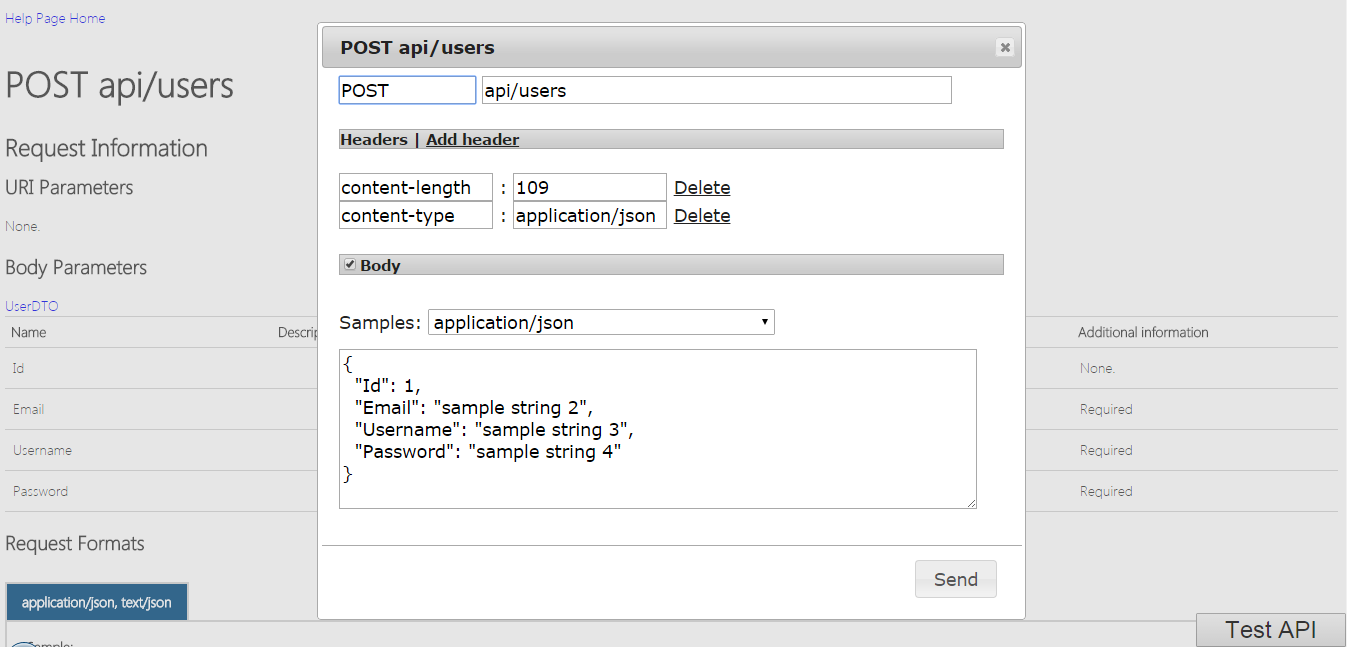


Figure 13 Test API client in action

**Prototype Version 5 Summary**

* Database in final form deployed on the web.
* Web API in final form deployed on the web.(1)
* Database Admin Pages in final form deployed on the web.(2)

At this point we are ready for development of Fantasy Hurling Client. It will be deployed on same server as admin pages(3).

**References**

1 <http://hurlingapi.azurewebsites.net/>

2 <http://fantasyhurling.azurewebsites.net/consume_api_examples/data_admin/login.html>

3 http://fantasyhurling.azurewebsites.net