Computer Science Programming Report 3rd Semester

Alexander Zahariev
Arnas Sokolovas
Jindrich Jehlicka
Marius Constantin Untaru

2016

The following report of Programming consists of two parts. First parts presents the idea of the project with the following problem statement.

Second part of the report covers some aspects of the product design and focuses on implementation of the system.

Contents

Chapter 1
Introduction

1.1 Approach 1: Environmental, more general auction - bidding system

Nowadays many household items are considered useless by their owners for different reasons. Therefore they are thrown away or replaced by new ones even though the items themselves are still in a good condition and might be useful for someone else. It happens that not all of the useless items are thrown away, some of them are being sold by the owners on different online platforms. Even though this way some of them end up not being sold because the price asked by the owner, whether it is a fixed or a negotiable price, doesn't fit the market price and the interest of the buyers might be lost.

This event might have a bad impact upon the environment as some items might not be recyclable or the recycling process is not very efficient and therefore resulting in a waste of resources.

1.2 Approach 2: Moving sale windows application for students

Often student experience troubles selling their old furniture or home appliances when moving out. A lot of old furniture and other stuff ends up outside next to the trash container, trashing the space and giving troubles for landlords or people who have to take care of the neighbourhood. A similar case occurs when the students are moving into new place and have troubles affording new beds, lamps, tables and everything else they need for daily study life. A lot of new students shelter furniture found in streets, unfortunately the conditions of them are highly changed or damaged due to the weather and humidity. An application such as a moving sale windows app, could be a great tool for both cases. For the students who are moving out, they would have a simple solution where they can place their items and furniture for sale instead of throwing out it into the streets, this way maybe earning some money. The other side of the case, the students who are moving in, they would not pick up the damaged furniture from the streets, instead they would negotiate and get stuff they like for the fair price

1.3 Problem Statement

How can a digital auction system be developed in the C# programming language with the help of the Agile software development process?

Chapter 2

Architecture and Design

2.1 Goal

The aim of the project was to create a web service that links the SQL database to a number of remote clients. The web service was decided to be implemented using ASP.NET WebAPI 2 to create an easy to use modular RESTful (Representational state transfer) API. That allows for the flexibility to choose any type of HTTP client. The choice was made and it was decided to have a web and a desktop client. The web application – a full-stack SPA (Single-Page Application) JavaScript application using NodeJS for the backend and AngularJS the frontend logic. Bootstrap with Material theme was used for the UI framework. The template engine was Jade (Pug). Everything was automated using Grunt scripts and Yeoman was used for generating the initial seed. The desktop application – a UWP (Universal Windows Platform) application which needs minor changes in order to support all Microsoft platforms in the Universal device family (Windows, Windows Phone, Xbox, etc.).

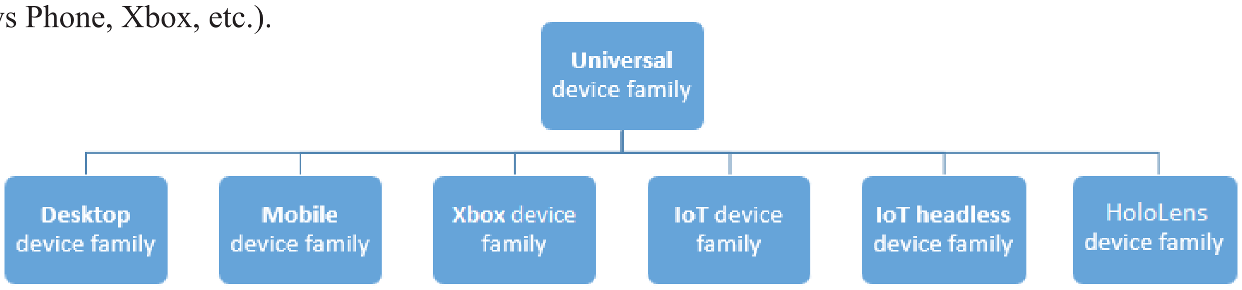


Figure 2.1: Device family tree [1]

2.2 Structure

The figure below(Fig 2.2, p.6) represents how the project is working. The web service (RESTful API) works as a middle point between the database and the client. The clients use simple HTTP requests for all the communications. They send these HTTP requests to the API and the API executes the necessary CRUD (Create, read, update and delete) operations on the database. After that, the API sends the response of the operations to the client in the data format that the client desires. The client receives the information and visualizes it. This way, the 3 layers are completely detached from each other, giving the benefit of having a program that is easier to maintain, test and expand.

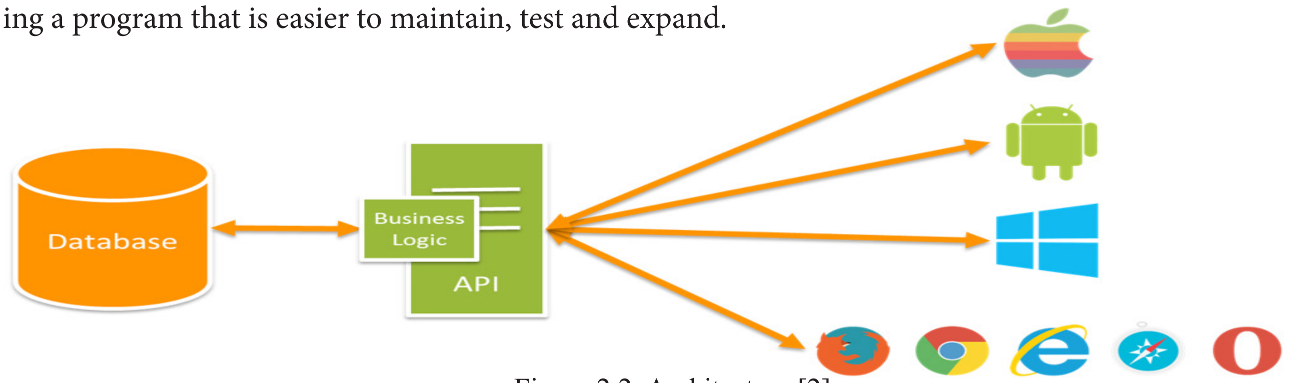


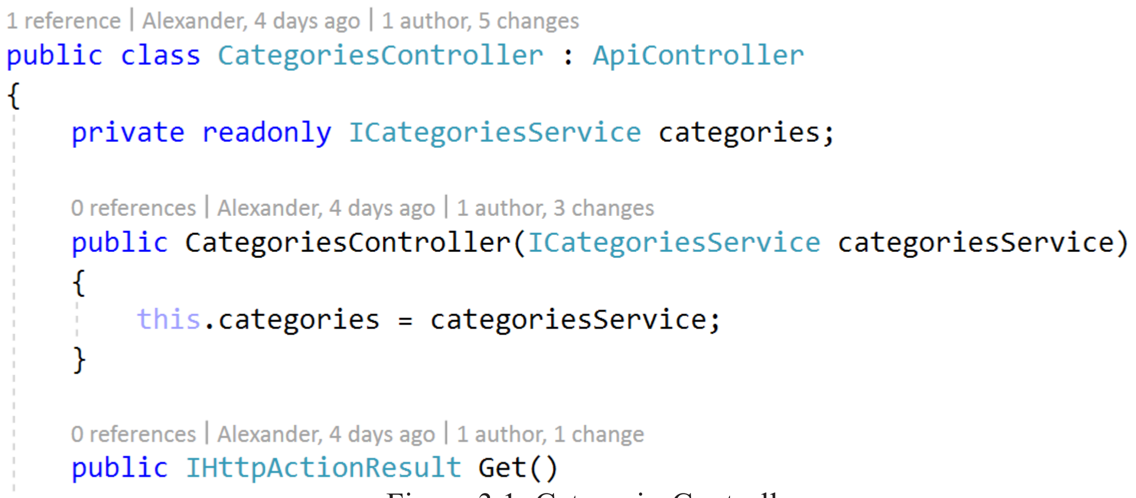
Figure 2.2: Architecture [2]

Chapter 3

Web service

3.1 Architecture

The goal of the project was to write high quality code that is easy to extend in the future but also easy to test and debug. One of the main objectives was to split the project in as many independent parts as possible. The core functionality of the API is in the AuctionSystem.Api project. It contains all the controllers and all the models that are related to the basic functionality of the RESTful API – response and request models. It also holds all configuration files that are necessary for the project to run properly. However, it does not contain a lot of the logic inside. It mainly invokes functions for the AuctionSystem.Services project which do all the database related work invoking functions in the AuctionSystem.Data project. All that is possible by using model definitions in the AuctionSystem.Data.Models project. There is also a project containing all the common variables shared by all projects, AuctionSystem.Common, and one holding the tests, AuctionSystem.Tests. By doing this, each project is independent from the others and the code is very easy to read. and understand.

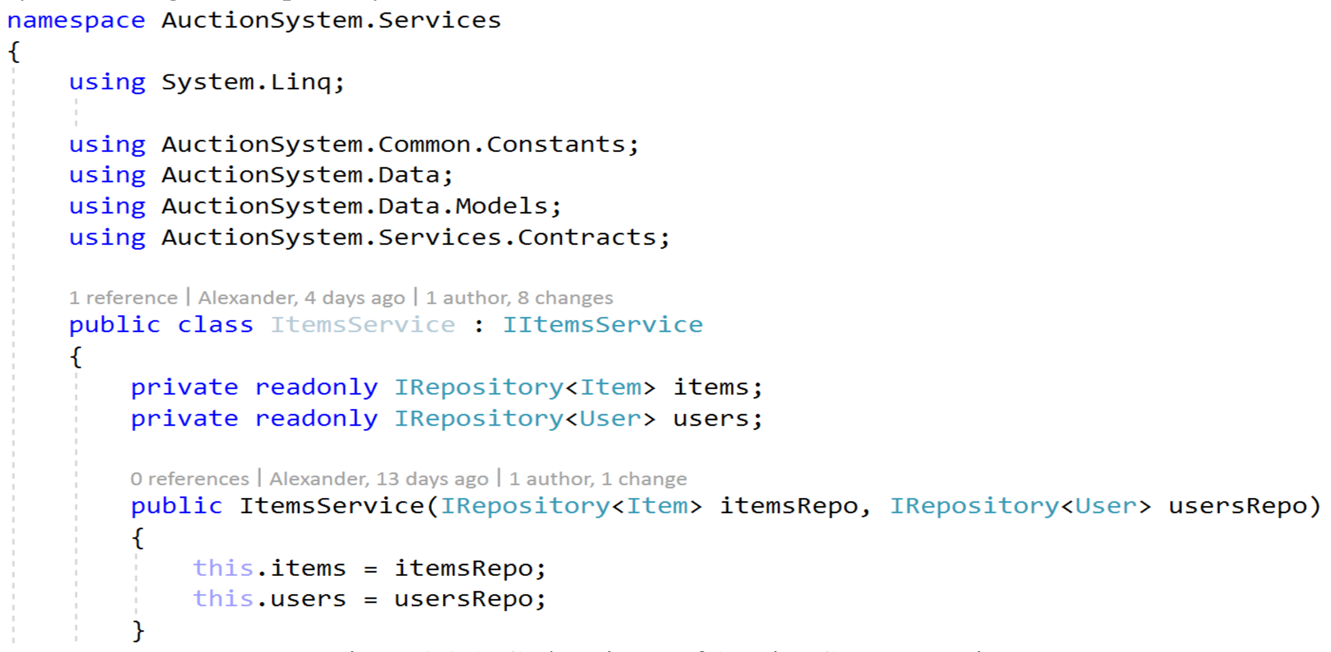

```
1 reference | Alexander, 4 days ago | 1 author, 5 changes
public class CategoriesController : ApiController
{
    private readonly ICategoriesService categories;

    0 references | Alexander, 4 days ago | 1 author, 3 changes
    public CategoriesController(ICategoriesService categoriesService)
    {
        this.categories = categoriesService;
    }

    0 references | Alexander, 4 days ago | 1 author, 1 change
    public IHttpActionResult Get()
```
Figure 3.1: CategoriesController

As seen in the figure above(Fig. 2.1 p. 6), this is part of the CategoriesController and it is responsible for getting all the categories from the database. Having services to do the core functionality enables the controllers to stay under 10 lines of code and remain very easy to understand.

3.2 Data

3.2.1 Entity Framework

AuctionSystem.Data is the database layer of the application. The database was created using code first approach with entity framework 6.1.3. this project is responsible for connecting our programming logic to the database using the Entity Framework. The project was implemented using Repository pattern. This allows us to change our database in the future if necessary by implementing the IRepository interface. For the current use case the program uses Entity Framework so the EfGenericRepository implements the IRepository interface. That way it is possible to use all the Entity Framework functionalities in different projects by initializing the IRepository interface.



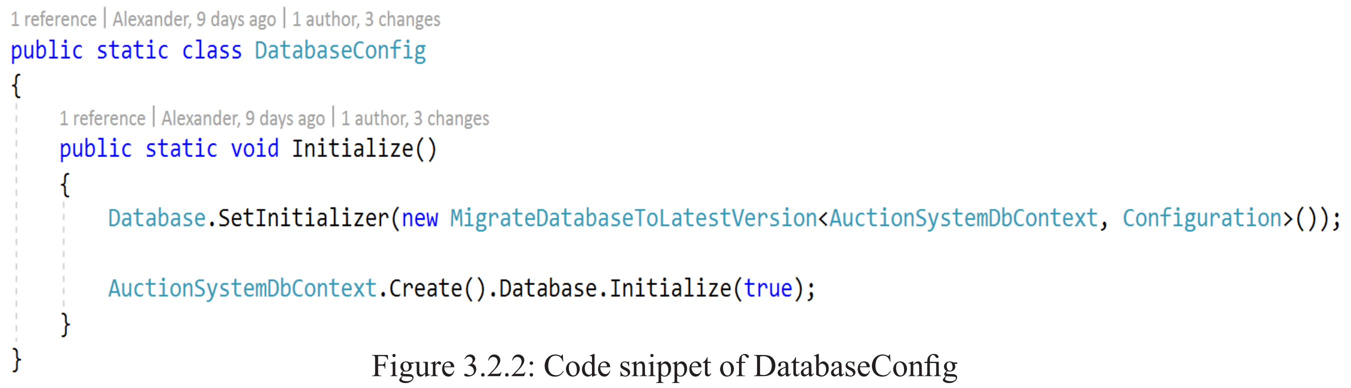
```
namespace AuctionSystem.Services
{
    using System.Linq;

    using AuctionSystem.Common.Constants;
    using AuctionSystem.Data;
    using AuctionSystem.Data.Models;
    using AuctionSystem.Services.Contracts;

    1 reference | Alexander, 4 days ago | 1 author, 8 changes
    public class ItemsService : IItemsService
    {
        private readonly IRepository<Item> items;
        private readonly IRepository<User> users;

        0 references | Alexander, 13 days ago | 1 author, 1 change
        public ItemsService(IRepository<Item> itemsRepo, IRepository<User> usersRepo)
        {
            this.items = itemsRepo;
            this.users = usersRepo;
        }
```

Figure 3.2.1: Code snippet of Auction System services

As seen in the figure 3.2.1 above, the AuctionSystem.Services project does not have information, if the IRepository is implemented by class using Entity Framework or not. It will work regardless of the implementation.

3.2.2 Configuration

The project also holds the IAuctionDbContext which is implemented by AuctionSystemDbContext class. It has a method Create() which is later called from DatabaseConfig from App_Start in AuctionSystem.Api in order to initialize a new DbContext and create the database. See figure below:



```
1 reference | Alexander, 9 days ago | 1 author, 3 changes
public static class DatabaseConfig
{
    1 reference | Alexander, 9 days ago | 1 author, 3 changes
    public static void Initialize()
    {
        Database.SetInitializer(new MigrateDatabaseToLatestVersion<AuctionSystemDbContext, Configuration>());

        AuctionSystemDbContext.Create().Database.Initialize(true);
    }
}
```

Figure 3.2.2: Code snippet of DatabaseConfig

The primary class that is responsible for interacting with data as objects is System.Data.Entity.DbContext (often referred to as context). The context class manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database. (reference)

3.2.3 Models

AuctionSystem.Data.Models contains all the models for the MVC (Model – View – Controller) architecture of the project. It is referenced from all other projects except AuctionSystem.Common. Based on these models the AuctionSystem.Data builds the database.
The database was completely generated by the models in AuctionSystem.Data.Models. The only thing that was added in addition was a database trigger to change the values of Expired column in Items table.

```
CREATE TRIGGER ItemExpiration
  ON dbo.Items
  FOR INSERT, UPDATE
  AS
  BEGIN
    UPDATE dbo.Items
    SET dbo.Items.Expired = 1
    FROM dbo.Items
    WHERE EndDate < getdate()
  END
```

Figure 3.2.3: Code snippet of Database

3.2.4 Database

Enabling database migrations allowed for extreme flexibility building the project and changing the database accordingly. Below is a picture of the current state of the database. And because of the migrations the database can be changed any time by editing the models.

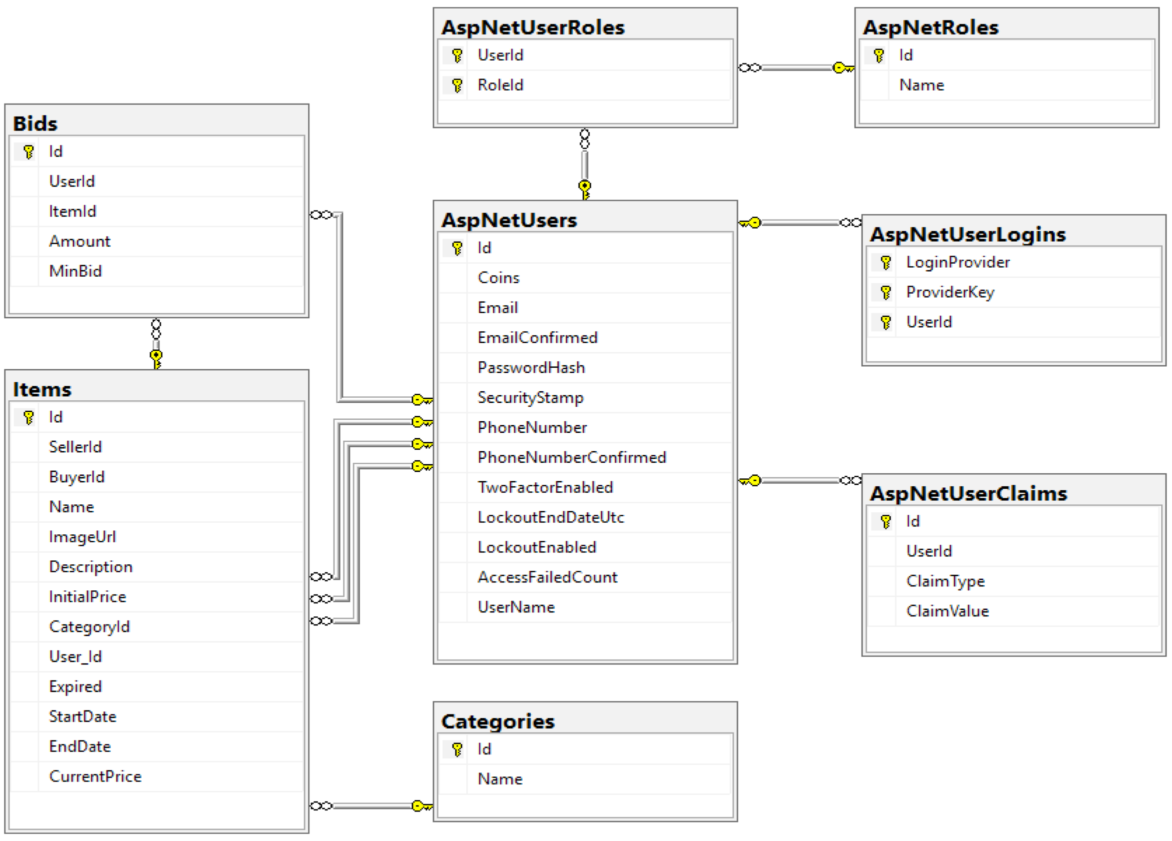


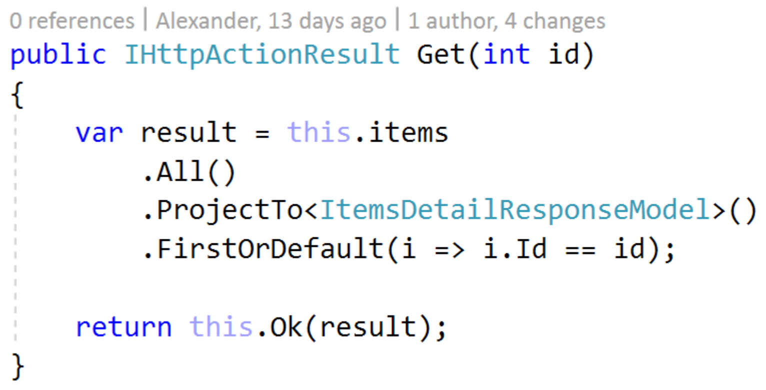
Figure 3.2.4: Database diagram

3.3 Web API

3.3.1 Controllers

AuctionSystem.Api is the ASP.NET WebAPI 2 application itself. It has references to all other projects in the solution. The models folder holds response and request models that the Web API uses to serializable the C# classes. The controllers folder holds all controller classes which inherit from and extend ApiController from System.Web.Http. Each API method that handles a user request returns IHttpActionResult. The execution flow of the controllers is very straightforward.

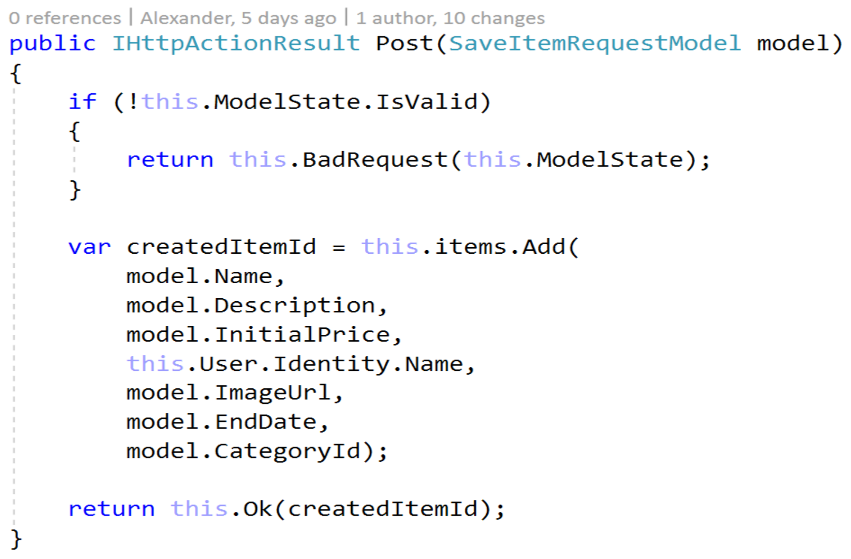
- If it is a GET method, it (the method) receives all relevant information through the corresponding class in AuctionSYstem.Services project and filters it using LINQ if necessary. After that the information get parsed to a response model and made into a list using AutoMapper. An example from ItemsController GET method for getting item from the database by id, can be seen below:



```
0 references | Alexander, 13 days ago | 1 author, 4 changes
public IHttpActionResult Get(int id)
{
    var result = this.items
        .All()
        .ProjectTo<ItemsDetailResponseModel>()
        .FirstOrDefault(i => i.Id == id);

    return this.Ok(result);
}
```

Figure 3.3.1: ItemsController GET

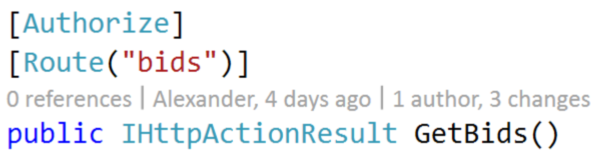
- If it is a POST method, it receives as argument a request model which creates a guideline to what the request should look like in order to be processed by the Web API. After that the user request is validated. If the user request does not match the requirement set by the request model, the API returns BadRequest() and information about the mistake. The client receives HTTP response code 400 (Bad Request). An example of a POST method from ItemsController using SaveItemRequestModel can be seen below:



```
0 references | Alexander, 5 days ago | 1 author, 10 changes
public IHttpActionResult Post(SaveItemRequestModel model)
{
    if (!this.ModelState.IsValid)
    {
        return this.BadRequest(this.ModelState);
    }

    var createdItemId = this.items.Add(
        model.Name,
        model.Description,
        model.InitialPrice,
        this.User.Identity.Name,
        model.ImageUrl,
        model.EndDate,
        model.CategoryId);

    return this.Ok(createdItemId);
}
```

Figure 3.3.1*: ItemsController POST

Some routes of the application require for the user to be authenticated. If a request is made without user authentication the server will respond with status code 401 (Unauthorized). An example from User-Controller where the previous bids can only be seen if the user is signed in, is presentet below:



```
[Authorize]
[Route("bids")]
0 references | Alexander, 4 days ago | 1 author, 3 changes
public IHttpActionResult GetBids()
```

Figure 3.3.1**: Authorization

In case everything is meeting the criteria set by the Web API, it returns status code 200 (OK) or 201 (Created).

3.3.2 Models

The folder models in AuctionSystem.Api does not contain the main models for the application that are used for creating the database. It keeps only the models that are required for the execution of the api. In the folder there are subfolders for every visualizable entity of the application:

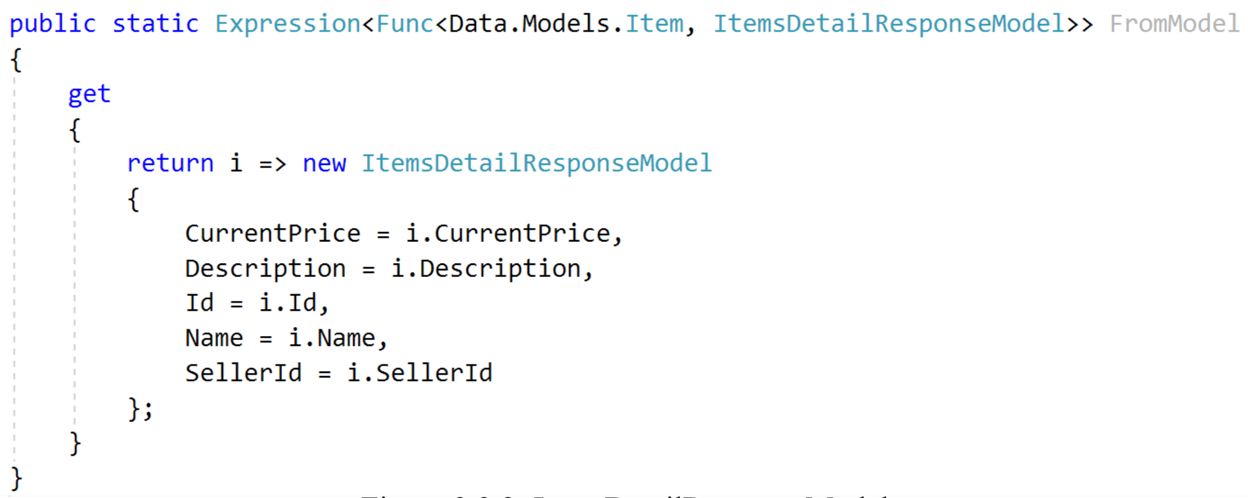
- *Account* – for every account related models - such as registering, authenticating users or visualizing user info.
- *Bid* – containing response and request model for the bid controller
- *Category* – response and request model for the category class
- *Item* – response and request model

Request models

Request models are the models that the API uses in order to bind the client's request to the database entity framework models during a POST request. The user request is being binded to the request model. After that, if the request model is validated successfully, a method from AuctionSystem.Services is called in order to execute the desired function.

Response models

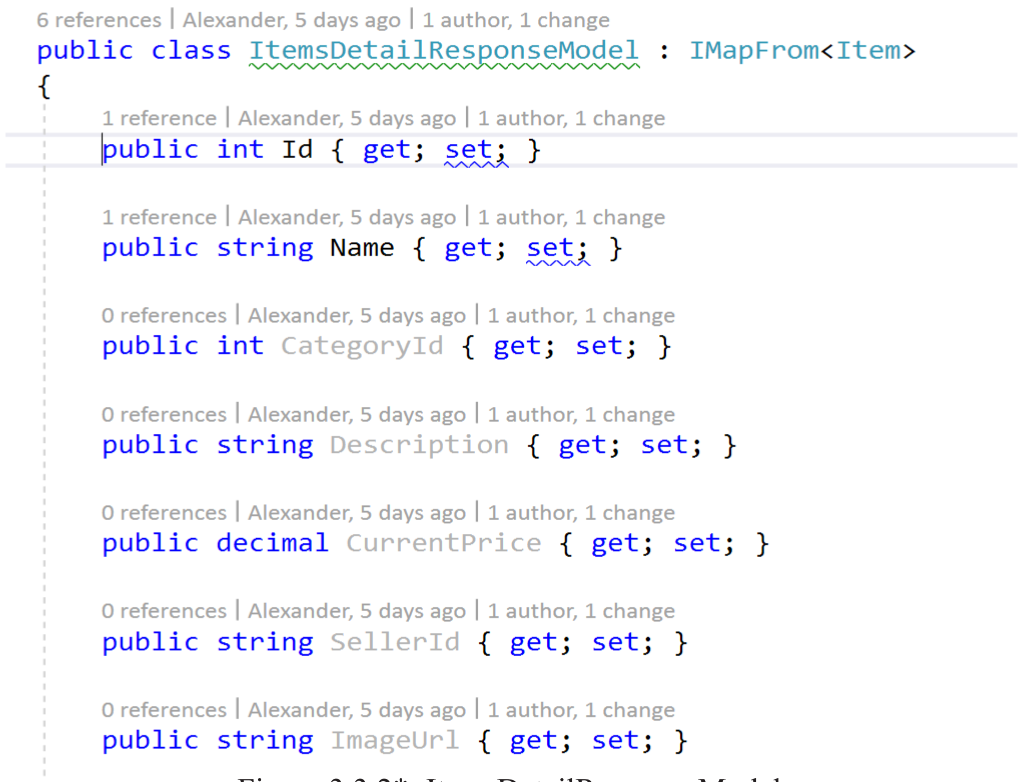
Response models are the models that the API uses to serialize the SQL classes coming the database thought Entity Framework to the desired of the client format (JSON, XML, etc). Using AutoMapper it was avoided duplicating code when creating the models. Bellow you can see photo of the code from the state where the project was not using AutoMapper.



```
public static Expression<Func<Data.Models.Item, ItemsDetailResponseModel>> FromModel
{
    get
    {
        return i => new ItemsDetailResponseModel
        {
            CurrentPrice = i.CurrentPrice,
            Description = i.Description,
            Id = i.Id,
            Name = i.Name,
            SellerId = i.SellerId
        };
    }
}
```

Figure 3.3.2: ItemsDetailResponseModel

Because of it was known that the project will have a lot of response models inside, it was decided that the best solution is to use AutoMapper. By implementing the response models classes using that framework it was possible to simplify them to a normal-looking models as it can be seen in the picture below:



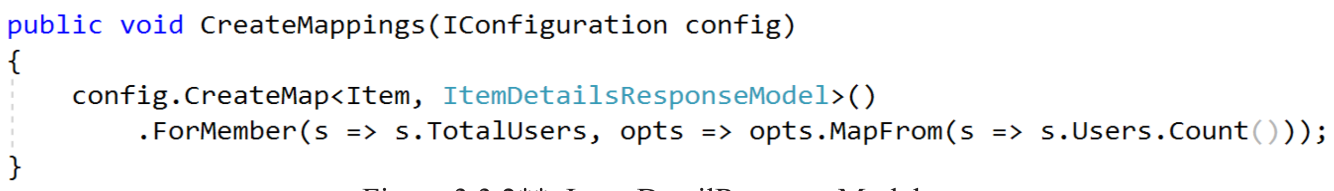
```
6 references | Alexander, 5 days ago | 1 author, 1 change
public class ItemsDetailResponseModel : IMapFrom<Item>
{
    1 reference | Alexander, 5 days ago | 1 author, 1 change
    public int Id { get; set; }

    1 reference | Alexander, 5 days ago | 1 author, 1 change
    public string Name { get; set; }

    0 references | Alexander, 5 days ago | 1 author, 1 change
    public int CategoryId { get; set; }

    0 references | Alexander, 5 days ago | 1 author, 1 change
    public string Description { get; set; }

    0 references | Alexander, 5 days ago | 1 author, 1 change
    public decimal CurrentPrice { get; set; }

    0 references | Alexander, 5 days ago | 1 author, 1 change
    public string SellerId { get; set; }

    0 references | Alexander, 5 days ago | 1 author, 1 change
    public string ImageUrl { get; set; }
```

Figure 3.3.2*: ItemsDetailResponseModel

Because of it was known that the project will have a lot of response models inside, it was decided that the best solution is to use AutoMapper. By implementing the response models classes using that framework it was possible to simplify them to a normal-looking models as it can be seen in the picture below:

- *IMapFrom<TModel>* - points that the class is a response model and takes all the properties which are matching their names with properties from TModel.

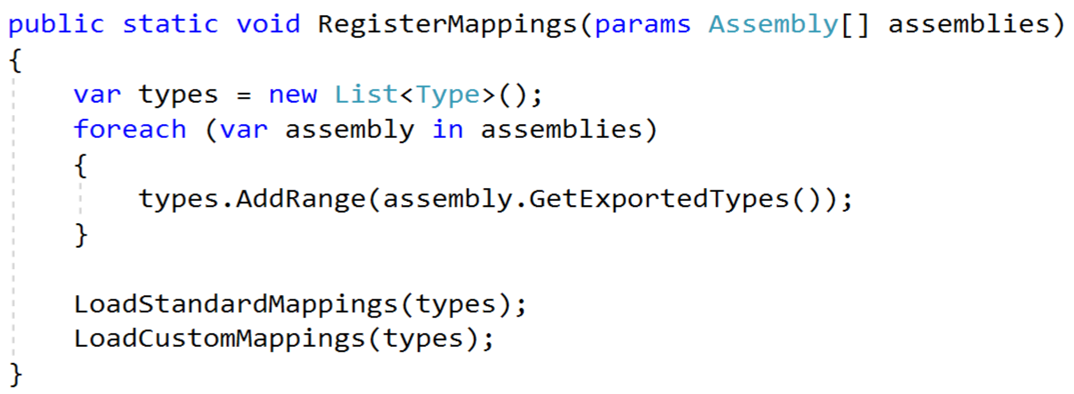
- *IHaveCustomMappings<>* - contain CreateMappings method. This is used when the response model has a property with a different name from the TModel and cannot be matched automatically. LINQ expression is used to map the additional properties as in the following picture



```
public void CreateMappings(IConfiguration config)
{
    config.CreateMap<Item, ItemDetailsResponseModel>()
        .ForMember(s => s.TotalUsers, opts => opts.MapFrom(s => s.Users.Count()));
}
```

Figure 3.3.2**: ItemsDetailResponseModel

The mapping is made possible by using Reflection in the AutoMapperConfig file. Before implementing AutoMapper with Reflection in was necessary to invoke Mapper.CreateMap() before the beginning of each controller. Now it is enough to use the method only once – when implementing the model. On the following page, a picture can be seen, representing this.

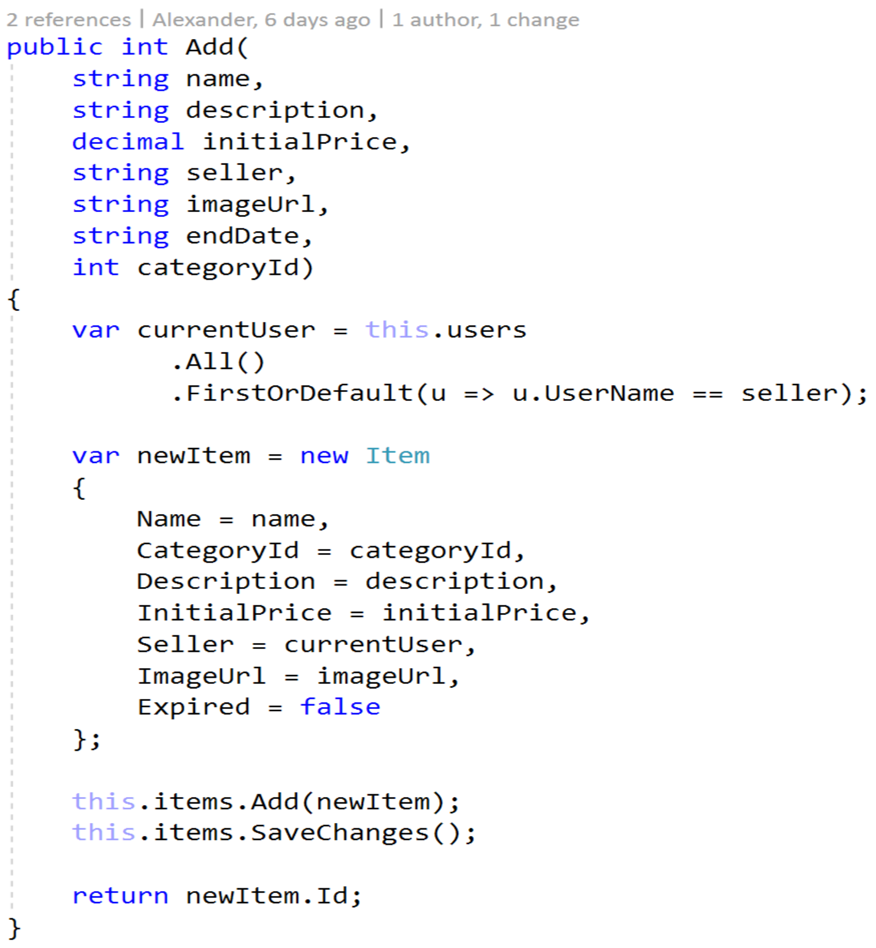


```
public static void RegisterMappings(params Assembly[] assemblies)
{
    var types = new List<Type>();
    foreach (var assembly in assemblies)
    {
        types.AddRange(assembly.GetExportedTypes());
    }

    LoadStandardMappings(types);
    LoadCustomMappings(types);
}
```

Figure 3.3.2***: Reflection in Automapperconfig

3.3.3 Services

All services are located in a C# Class Library named AuctionSystem.Services. The services can handle a lot of logic such as caching objects, background tasks, identity tasks and much more. In this case they are used to simplify the controllers. The services act like a bridge between the database and the controllers. They get all the needed information from the database and send it to the controller which is responsible for filtering and resending it to the clients. That way the controllers are separate from the database as they only instantiate a service's interface in their body. On the other hand, the services contain reference to the data layer using the IRepository interface. Because of the implementation of a repository pattern they are not restricted to using only SQL database with Entity Framework and will work with every database that implements the IRepository interface.



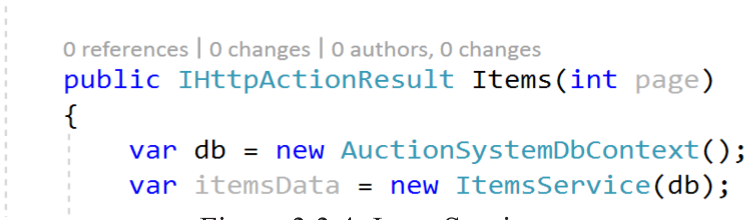
```
2 references | Alexander, 6 days ago | 1 author, 1 change
public int Add(
    string name,
    string description,
    decimal initialPrice,
    string seller,
    string imageUrl,
    string endDate,
    int categoryId)
{
    var currentUser = this.users
          .All()
          .FirstOrDefault(u => u.UserName == seller);

    var newItem = new Item
    {
        Name = name,
        CategoryId = categoryId,
        Description = description,
        InitialPrice = initialPrice,
        Seller = currentUser,
        ImageUrl = imageUrl,
        Expired = false
    };

    this.items.Add(newItem);
    this.items.SaveChanges();

    return newItem.Id;
}
```

Figure 3.3.3: ItemsService

All services are located in a C# Class Library named AuctionSystem.Services. The services can handle a lot of logic such as caching objects, background tasks, identity tasks and much more. In this case they are used to simplify the controllers. The services act like a bridge between the database and the controllers. They get all the needed information from the database and send it to the controller which is responsible for filtering and resending it to the clients. That way the controllers are separate from the database as they only instantiate a service's interface in their body. On the other hand, the services contain reference to the data layer using the IRepository interface. Because of the implementation of a repository pattern they are not restricted to using only SQL database with Entity Framework and will work with every database that implements the IRepository interface.

3.3.4 Dependency Inversion

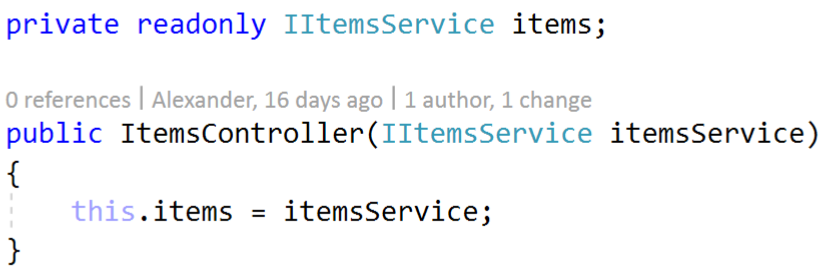
Instantiating the thought out the actions is a bad idea because it creates a new DbContext for each request. Example:



```
0 references | 0 changes | 0 authors, 0 changes
public IHttpActionResult Items(int page)
{
    var db = new AuctionSystemDbContext();
    var itemsData = new ItemsService(db);
```

Figure 3.3.4: ItemsService

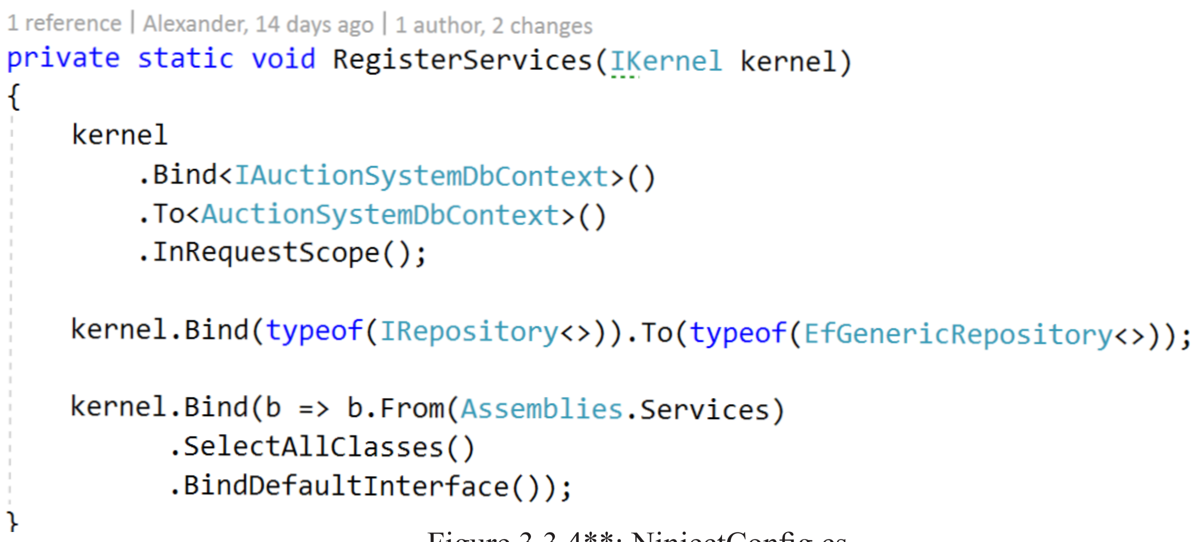
Much better solution would be to put them inside the constructor as interfaces and "inject" them from outside. Example:



```
private readonly IItemsService items;

0 references | Alexander, 16 days ago | 1 author, 1 change
public ItemsController(IItemsService itemsService)
{
    this.items = itemsService;
}
```

Figure 3.3.4*: ItemsService

However, in order for this to work it is important to have "something" to "inject" the dependencies. Installing an Inversion of Control Container (IoC) solves this problem. In this project it was used Ninject, as it can be seen in the following picture:



```
1 reference | Alexander, 14 days ago | 1 author, 2 changes
private static void RegisterServices(IKernel kernel)
{
    kernel
        .Bind<IAuctionSystemDbContext>()
        .To<AuctionSystemDbContext>()
        .InRequestScope();

    kernel.Bind(typeof(IRepository<>)).To(typeof(EfGenericRepository<>));

    kernel.Bind(b => b.From(Assemblies.Services)
          .SelectAllClasses()
          .BindDefaultInterface());
}
```

Figure 3.3.4**: NinjectConfig.cs

The RegisterServices method inside NinjectConfig which is located in App_Start in AuctionSystem.Api project is responsible for binding all interfaces to their implementations.

Chapter 4
Web client

4.1 Structure

The root of the SPA was created by installing generator-angular-jade-stylus with yeoman. The main folder of the project holds the "app" folder which contains all views and controllers. There is also the package.json file which holds information about all node modules on which the project depends. In bower.json are described all client-side dependencies of the project such as UI frameworks and JavaScript libraries. Gruntfile.js holds all scripts for automating the development process such as recompiling the files on change and refreshing the web page, installing all modules, building minified production version, etc.
The "app" folder is the core of the application. Inside "views" are all Jade files which are used to generate the HTML which is visualized by the browser. In "scripts" are all AngularJS modules. They are the logic on which the HTML files are built and visualized. Folder styles keeps all Stylus files which are later compiled into CSS files using Grunt.

4.2 NodeJS

In order to build the application, it is required to have NodeJS installed on the computer. Once NodeJS is installed on the local machine the following commands have to be executed in the terminal at the root of the project:

- *npm install* – downloads all node dependencies described in package.json using NPM (Node Package Manager);
- *bower install* – downloads all client-side libraries;
- *grunt serve* – runs code style check, compiles all files and starts the project on https://localhost:9000/
- */optional/ grunt build* – runs all unit tests and if they pass successfully, it builds deployment version of the project; minifies all files and merges them in one; compresses all images.

4.3 AngularJS

AngularJS is a MV* framework used to create JavaScript Single-Page applications. The file app.js is the starting point of the application. It sets the main dependencies of the project using dependency injection and configures all the routes with their views and corresponding controllers. The figure in the next page will shows a picture of the code.

```
var app = angular
    .module('auctionApp', ['ngCookies', 'ngResource', 'ngSanitize', 'ngRoute'])
    .config(function ($routeProvider) {
      $routeProvider
        .when('/', {
          templateUrl: 'views/main.html',
          controller: 'MainCtrl'
        })
        .when('/login', {
          templateUrl: 'views/login.html',
          controller: 'LoginCtrl'
        })
```

Figure 4.3: app.js

4.4 Controllers

The controllers in the application are built on a similar fashion as the controllers in the API. The consume the information from the services. In this situation they are receiving the data from the services as JSON and are saving it in the $scope. Later the view visualizes on user's screen the information from the $scope.

```
'use strict';

app.controller('ItemCtrl', function ($scope, $routeParams, ItemsResource) {

    $scope.getItem = function(id) {
        ItemsResource.getById(id).success(function(item){
            $scope.item = item;
        })
    }
```

Figure 4.4: ItemCtrl

4.5 Services

The main responsibility of the services in the project is to send request to the web service and handle its responses.

- If it is a POST request in the service is created a function that takes a variable holding all the information that is going to be send to the server. After the information is received by the server it is parsed to the correct RequestModel and validated. Then the service receives the server's response code and returns it for the controller to use. As in the code snippet below:

```
create: function(item) {
    return $http.post(itemsApi, item, {
        headers: authorization.getAuthorizationHeader()
    })
},
```

Figure 4.5: POST request

- If it is a GET request in the service is created a function that takes all necessary variables for the request (if any) as parameters. Then it sends the request to the server, which get the required information from the database, parses it to the correct ResponseModel and returns it as JSON to the service.

```
getById: function(id) {
    return $http.get(itemsApi + '/' + id);
},
```

Figure 4.5*: GET request

4.6 Authentication

The WebAPI uses token-based OAuth authentication and because of some of the functionality of the API require authentication it was first in the implementation schedule.
For registering a user the API is expecting a POST request at api/account/register with body containing email, password and confirmPassword in JSON format. This is implemented in auth factory.

```
angular.module('auctionApp')
.factory('auth',
    function($http, $q, identity, authorization, UsersResource, baseUrl) {
        var usersApi = baseUrl + '/api/account'

        return {
            signup: function(user) {
                var deferred = $q.defer();
                var user = new UsersResource.register(user);
                user.$save().then(function() {
                    deferred.resolve();
                }, function(response) {
                    deferred.reject(response);
                });
                return deferred.promise;
            },
```

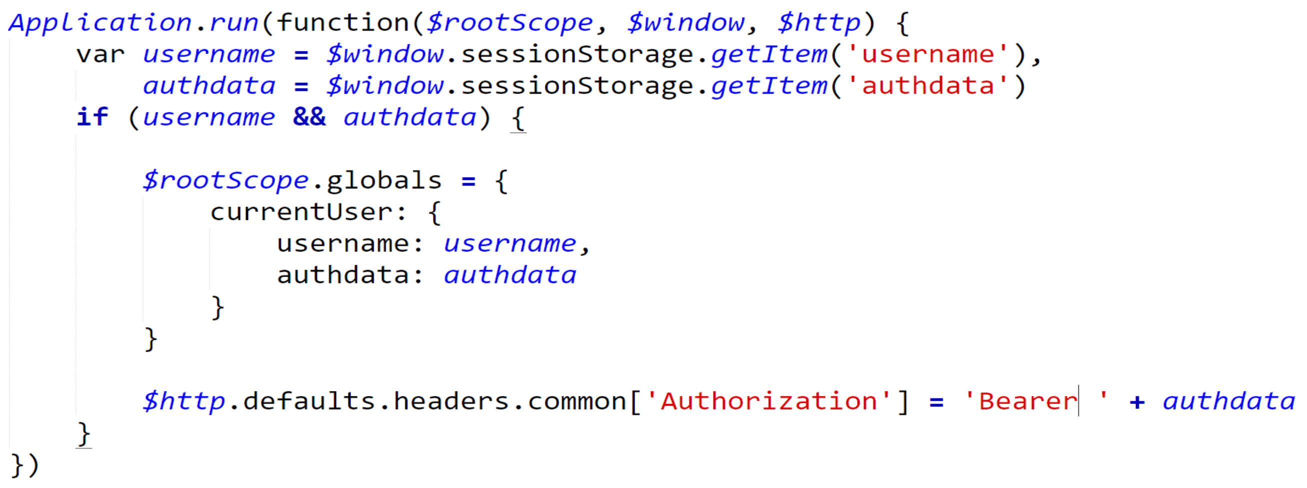
Figure 4.6: SignUp user

For logging in the API expects POST request at /token containing username, password and grant_ type in x-www-form-urlencoded format. As a response the client receives a unique token that is user for the authentication.

```
login: function(user) {
    var deferred = $q.defer();
    user['grant_type'] = 'password';
    $http.post(baseUrl + '/token',
     'username=' + user.username +
      '&password=' + user.password +
       '&grant_type=password', {
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        }
    }).success(function(response) {
        if (response["access_token"]) {
            identity.currentUser = response;
            deferred.resolve(true);
        } else {
            deferred.resolve(false);
        }
    });
    return deferred.promise;
},
```

Figure 4.6*: Log In

After the token is received it is stored in the sessionStorage of the used HTML5 browser. Now every request that requires authentication from the user should contain header Authorization with value Bearer and the token. Example of how the authentication data is stored to sessionStorage and added to the HTTP header can be seen below:



```
Application.run(function($rootScope, $window, $http) {
    var username = $window.sessionStorage.getItem('username'),
        authdata = $window.sessionStorage.getItem('authdata')
    if (username && authdata) {

        $rootScope.globals = {
            currentUser: {
                username: username,
                authdata: authdata
            }
        }

        $http.defaults.headers.common['Authorization'] = 'Bearer ' + authdata
    }
})
```

Figure 4.6**: Credentials

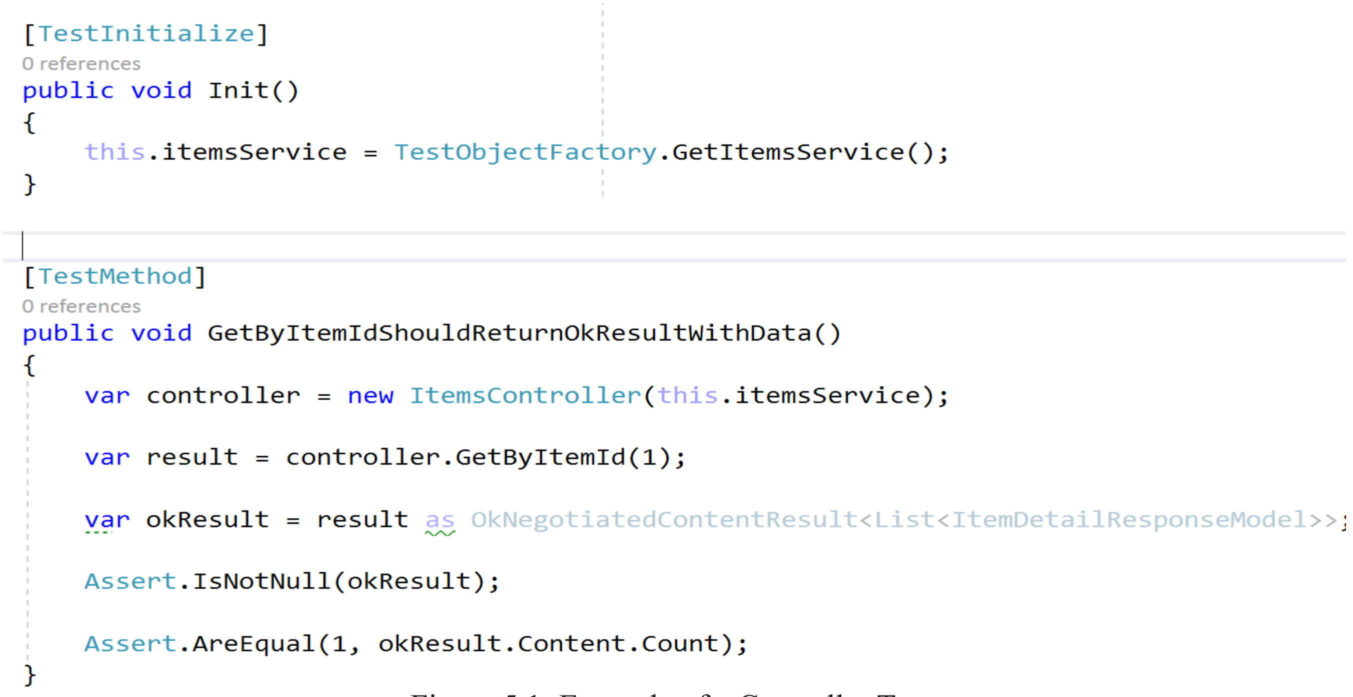
Chapter 5

Testing and Refactoring

5.1 Testing

Testing is necessary to spot all the mistakes and errors that were made during the planning and the development. Another reason why testing is important is to make sure the application does not result in any failure, because it might be expensive to fix it in the future. Testing is essential in order to deliver high quality product, which requires low maintenance, to the customer.[3]
Unit testing was a crucial part in implementing the API. Before every commit to GitHub all tests were ran to make sure that the new functionality did not have any bugs. The Unit test were implemented using TestObjectFactory. Angular was tested with framework called Karma.

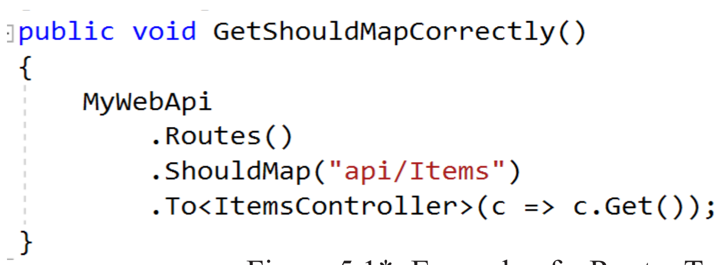
Here are presented some of the tests that were made:



```
[TestInitialize]
0 references
public void Init()
{
    this.itemsService = TestObjectFactory.GetItemsService();
}
```

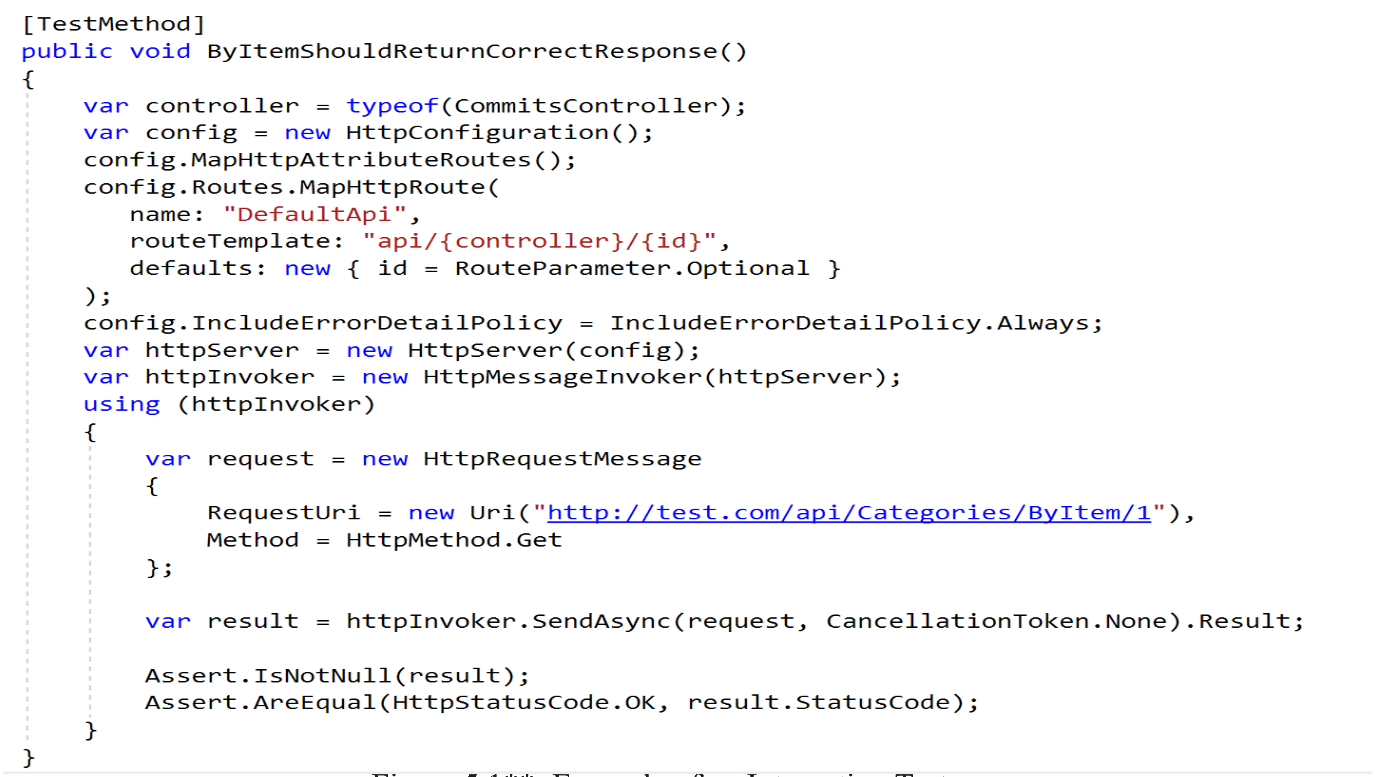
```
[TestMethod]
0 references
public void GetByItemIdShouldReturnOkResultWithData()
{
    var controller = new ItemsController(this.itemsService);

    var result = controller.GetByItemId(1);

    var okResult = result as OkNegotiatedContentResult<List<ItemDetailResponseModel>>;

    Assert.IsNotNull(okResult);

    Assert.AreEqual(1, okResult.Content.Count);
}
```

Figure 5.1: Example of a Controller Test



```
public void GetShouldMapCorrectly()
{
    MyWebApi
        .Routes()
        .ShouldMap("api/Items")
        .To<ItemsController>(c => c.Get());
}
```

Figure 5.1*: Example of a Router Test

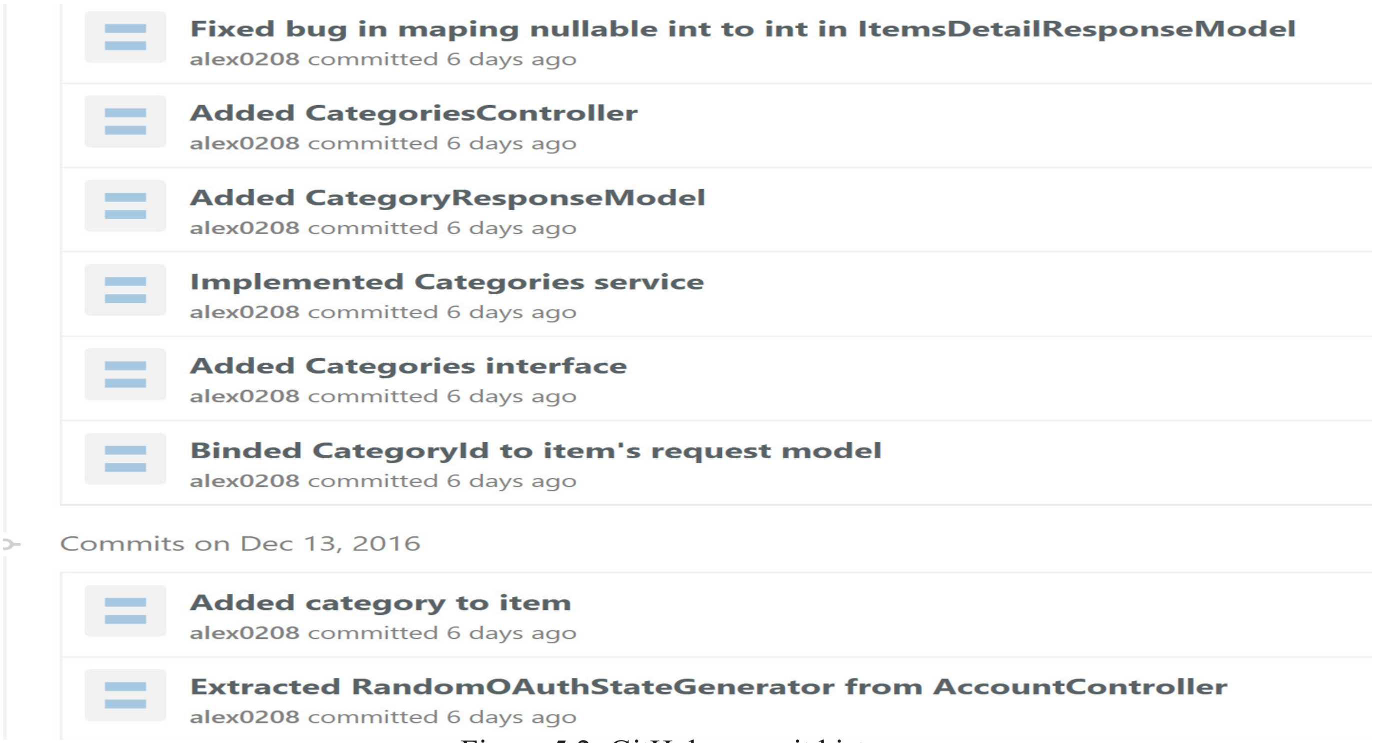


```
[TestMethod]
public void ByItemShouldReturnCorrectResponse()
{
    var controller = typeof(CommitsController);
    var config = new HttpConfiguration();
    config.MapHttpAttributeRoutes();
    config.Routes.MapHttpRoute(
       name: "DefaultApi",
       routeTemplate: "api/{controller}/{id}",
       defaults: new { id = RouteParameter.Optional }
    );
    config.IncludeErrorDetailPolicy = IncludeErrorDetailPolicy.Always;
    var httpServer = new HttpServer(config);
    var httpInvoker = new HttpMessageInvoker(httpServer);
    using (httpInvoker)
    {
        var request = new HttpRequestMessage
        {
            RequestUri = new Uri("http://test.com/api/Categories/ByItem/1"),
            Method = HttpMethod.Get
        };

        var result = httpInvoker.SendAsync(request, CancellationToken.None).Result;

        Assert.IsNotNull(result);
        Assert.AreEqual(HttpStatusCode.OK, result.StatusCode);
    }
}
```

Figure 5.1**: Example of an Integration Test

5.2 Workflow and Style

The project is following a very strict style and quality guides. For the Web Service there was a StyleCop check ran on each file and all styling mistakes were fixed. For the Web Client there was a EsLint check ran on every save by Grunt. The project would not compile if there were any styling errors. Before making each commit the new code had to pass the unit tests which were already implemented. All of the code was kept in remote repository in GitHub as a private project as the group decided not to have an open source project. Below is a screenshot of the project's commit history.



Fixed bug in maping nullable int to int in ItemsDetailResponseModel
alex0208 committed 6 days ago

Added CategoriesController
alex0208 committed 6 days ago

Added CategoryResponseModel
alex0208 committed 6 days ago

Implemented Categories service
alex0208 committed 6 days ago

Added Categories interface
alex0208 committed 6 days ago

Binded CategoryId to item's request model
alex0208 committed 6 days ago

Commits on Dec 13, 2016

Added category to item
alex0208 committed 6 days ago

Extracted RandomOAuthStateGenerator from AccountController
alex0208 committed 6 days ago

Figure 5.2: GitHub commit history

Chapter 6
Conclusion

All in all, this was the biggest and the most complex project for the group so far. It required a lot of planning and understanding how multiple systems work.
Because of how different the modules were from each other, they required completely new methodologies and strategies for implementation. From programming in C#.NET – an object-oriented programming language using strong typing, to the complete opposite JavaScript – a dynamic, untyped functional language. From programming in big and heavy, fully-featured IDE like Visual Studio to a minimalistic light-weight text editor like Sublime Text. From Pascal case to Camel case naming style.
The project was very beneficial for the group's members, giving an insight of programming in two completely different environments.

References

- [1] - Msdn.microsoft.com. (2016). Intro to the Universal Windows Platform. [online] Available at: https://msdn.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide [Accessed 20 Dec. 2016].
- [2] - Martin Kearn. (2015). Introduction to REST and .net Web API. [online] Available at: https://blogs.msdn.microsoft.com/martinkearn/2015/01/05/introduction-to-rest-and-net-web-api/ [Accessed 20 Dec. 2016].
- [3] - Istqbexamcertification.com. (n.d.). Why is software testing necessary?. [online] Available at: http://istqbexamcertification.com/why-is-testing-necessary/ [Accessed 20 Dec. 2016].