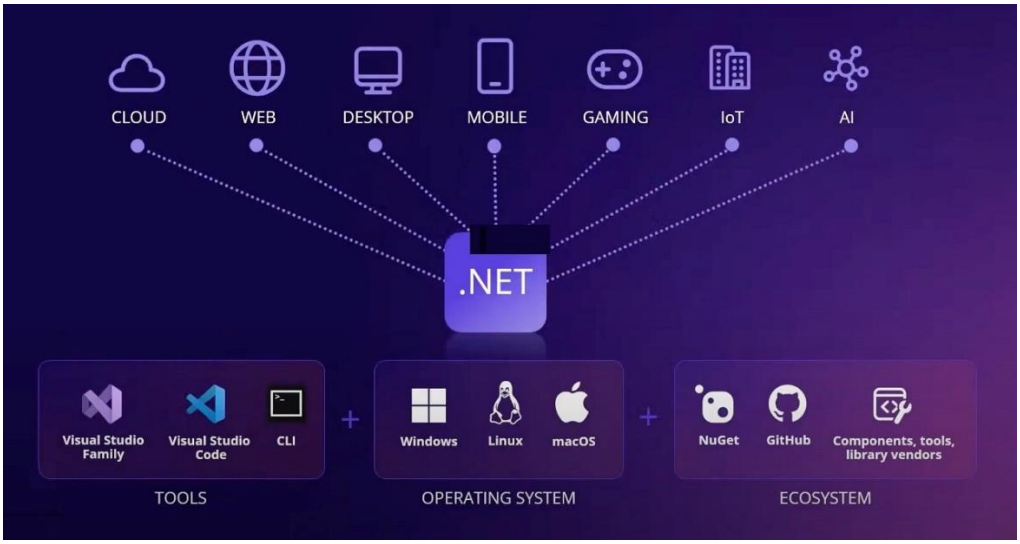
Understanding The Technology Behind the Scenes

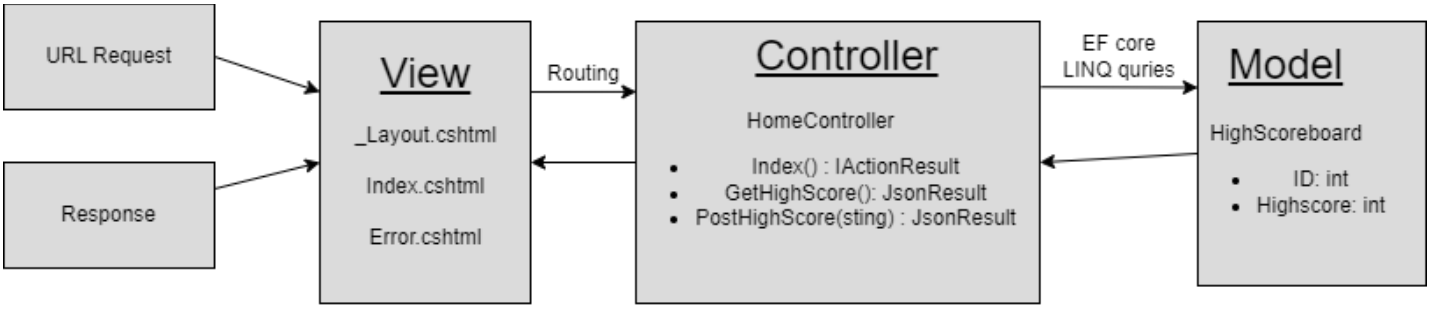
Tech Stack

We spent little bit of our research time on researching different technologies to use for our web application. At the end it came down to two tech stacks. Two options were using MVC design pattern web application with Microsoft .NET Core or using the .NET Core for the API calls and host the complete front-end on angular. From our Co-ops, all of our team member were familiar with MVC design pattern and .NET core so instead of beginning from the scratch with Angular front-end we decided to work with razor page on MVC design pattern with .NET 6.



Here is a high overview of the technology used in this project. We used the Web component of .NET 6. With .NET you can build many different types of applications and it is cross-platform compatible. That came very useful when we deployed our application on Azure since there, we hosted our application on Linux operating system. The tool we used to do our development was Visual Studio Community version which is great for developing applications. Visual Studio has GitHub integration which made the work much smoother with version controlling. We also used the NuGet packages for Jquery and bootstrap. Web application on .Net comes with cshtml which is basically HTML with C stands for CSharp so we were able to inject CSharp code into the html. It also supports JavaScript and other scripting tool. .Net has Entity Framework core where we were able to build SQL table with CSharp classes and migrate that into the SQL and .Net would automatically convert the code into SQL and apply it to our database using the connection string provided on the appsetting.json file. We were able to use LINQ queries to query the database. LINQ queries are basically the CSharp code that queries the database table. Entity Framework was great since it got rid of redundant work of creating database when working together with our teammates and pushing database changes on cloud. Instead of manually changing the schema of the database for each developer and on Azure all we had to do was change it into our CSharp code and it would automatically apply it all across all systems when developers use EF Core command update database. For Azure, it would automatically apply the database schema changes every time the web application is published. So, there were so many tools we used during the process of building our web application so .Net and Microsoft was the best tech stack for us.

MVC Design Pattern For our project



Here is the high-level overview of how front-end and back-end communicates on our page. I will explain the details of these here. URL Request and Response is the browser side.

Homepage:

When users enter our site on the browser, they send a GET request to get the Index.cshtml. We have three views. _Layout.cshtml is basically the layout of the page which has header and other basic HMTL that we could reuse for other pages like Error.cshtml. Error.cshtml is there to server if the server responded with 404 not found and other HTML errors. The main page is Index.cshtml. Since our application is a single page application, we have all of the main front-end code on Index.cshtml. Our _layout.cshtml is a shared component so it contains all of our scripts and stylesheet references and also the navigation bar. On the body section of the _layout.cshtml we render the Index.cshtml. When user sends a GET request to view our page the controller will handle the routing and with the Index() action it will serve the index page.

```
0 references
public IActionResult Index()
{
    return View();
}
```

Here is what the method looks like. Since our view is named Index.cshtml and the method is also named Index we don't need to provide the name of the View when we return the method.

GET High Score request:

Since we only have three endpoints for the project, we decided to use a home controller for everything so that project is less messy. When the home page is loaded, the JavaScript code that handles the game logic will automatically send a HTTP GET request to show on the page.

```
site.js*
Miscellaneous                                   getHighestScor
355  function callback(response)...
358  function getHighestScore() {
359      var geturl = document.getElementById('GetUrl').value;
360      $.ajax({
361          type: "GET",
362          url: geturl,
363          dataType: "json",
364          success: function (result) {
365              callback(result);
366          },
367          error: function (req, status, error) {
368              console.log(status)
369          }
370      });
371  }
```

```
<input type="hidden" id="GetUrl" name="custId" value='@Url.Action("GetHighScore")'>
```

This is the Ajax call that happens when the Index page is loaded. We used hidden input type to get the url to make it dynamic. Since out page is cshtml we use the CSharp code that populates the input with the url for Ajax call.

```
[HttpGet]
0 references
public JsonResult GetHighScore()
{
    var highScore = _db.HighscoreTable.Select(x => x.HighScore);
    if (highScore.Count() < 1)
    {
        return new JsonResult(Ok(0));
    }
    var maxhighScore = highScore.Max();
    return new JsonResult(Ok(maxhighScore));
}
```

As soon as page loaded, the JavaScript makes the Ajax call to this endpoint on the controller. Here I am using the LINQ query to query the table and return max value from the table input. This returns the Json result with value and status ok.

```
Level: 1  Current Score: 0  Highest Score: 1500
```

After we got the response, it updated the front-end here.

HTTP POST follows same pattern as this, the difference with the POST method is that the Ajax call will check if the player has set a new high score or not. If a player has set a new high score, then it will make the post call otherwise it won't send anything to controller.

```
function postHighestScore() {
    var posturl = document.getElementById('PostUrl').value;
    $.ajax({
        type: "POST",
        url: posturl,
        dataType: "json",
        data: highscoreObj,
        success: function (result) {
        },
        error: function (req, status, error) {
            console.log(status)
        }
    });
}
```

```
[HttpPost]
0 references
public JsonResult PostHighScore(string highscore)
{
    int test = (int)Convert.ToInt64(highscore);
    var objForDB = new HighScoreboard
    {
        HighScore = test
    };
    _db.HighscoreTable.Add(objForDB);
    _db.SaveChanges();
    return new JsonResult(Ok());
}
```

```
<input type="hidden" id="PostUrl" name="custId" value='@Url.Action("PostHighScore")'>
```

Here is the component of the HTTP POST. Same idea as HTTP Get but HTTP get will update the view where HTTP Post will update the database when meets certain criteria.

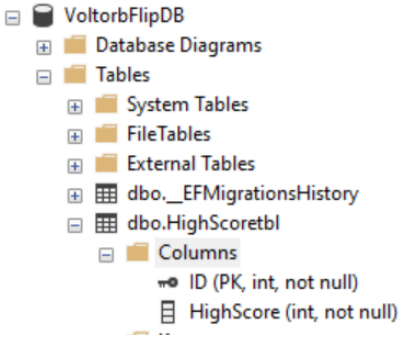
Model:

The model is an exact copy of our database table. We created the model with CSharp classes and using the data annotations we are able to manipulate things like difference on variable names and what we see on SQL database.

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Voltorb_Flip_Web_App.Models
{
    [Table("HighScoretbl")]
    2 references
    public class HighScoreboard
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        0 references
        public int ID { get; set; }

        [Column("HighScore")]
        2 references
        public int HighScore { get; set; }
    }
}
```



Here you can see side-by-side view of the model and the table on SQL server management studio. As you can see the Table("HighScoretbl") annotation matches the table name on the SQL server. And also, the column name for HighScore even though the variable name is different. We could also specify the data type of each column, but entity framework is able to translate that to appropriate value. We use this CSharp class model and use it with LINQ queries to communicate our backend with the controller.

Conclusion

In conclusion, the MVC design pattern works very well, and it worked great for our project. MVC stands for Model, View, and Controller. Model is our database model which has all the details of our database and using that our controller can communicate with the database side. View is the front-end. View is the pages that user can see when they request the page. Controller handles all the logic such as serving the pages when user request to updating and getting data from database.