# USING ASP.NET CORE WEB API

By

Joshua Salijeni

635307

## Contents

Introduction ........ 2
Client/Server Application ........ 2
Routing Implementation ........ 3
RESTful Request Methods ........ 3
Get Hello ........ 3
Get Sort ........ 4
Creating a User ........ 4
Deleting User ........ 5
Get User ........ 5
Update User Role ........ 6
Protected Get Hello ........ 6
Protected Get SHA1 ........ 7
Protected Get SHA256 ........ 7
Protected Get SHA256 ........ 7
Using User API Key and Middleware ........ 8
Entity Framework ........ 8
Reflection ........ 8
Conclusion ........ 9
References ........ 9

## Introduction

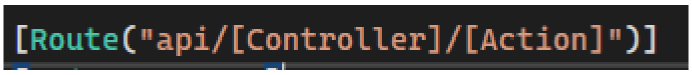
In the rapidly expanding digital landscape, delivering information to users precisely when it's needed is vital. With the increasing reliance on the internet, the transportation of data must be executed with efficiency, speed, and security—key principles that RESTful APIs are designed to uphold. The architectural style known as Representational State Transfer (REST) establishes a collection of principles for creating distributed systems and applications. These principles have shaped the design and evolution of the modern Web. Web services that adhere to the REST style are termed RESTful Web services, and their programmatic interfaces are referred to as REST APIs (Rodríguez, C., et, al, 2016).

### Client/Server Application

An application was built based on REST principles using ASP.NET, the application consisted of a client and a server that communicated and transferred data. The API managed request in an asynchronous manner by handling authentication, validating input data, retrieving, and manipulating data from the database. Because the API was built based on REST principles, it inherited a stateless design. The API was stateless as each time a request was made from the client to server, data was not stored but rather the client provided all the information required to complete the specified tasks. Each request was handled independently, and the server did not retain information about any previous interactions made. If the server was stateful the server could have stored information from previous interactions and request couldn't have been handled independently as data stored from prior interactions would influence the processing of subsequent requests. Red Hat (2023) stated that the main distinction between stateful and stateless systems lies in how they handle user interactions. In a stateful system, the application stores information about the user's current state across multiple requests, while in a stateless system, each request is treated as a separate and independent transaction without retaining any information about previous interactions.

## Routing Implementation

The application was able to carry out task by using routing which had access to various API endpoints, The routing was implemented in each controller. First in the base controller a routing template that defined how routing should be implemented in other controllers was created so that route mapping can consist of a controller name and the action being carried out by the specific controller based on the mapping. Each controller class inherited this implementation and mapped each action it implemented to the corresponding URI based on the defined route template.



```
[Route("api/[Controller]/[Action]")]
```

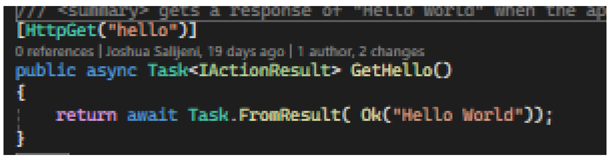
*Figure 1 Base controller routing template.*

This approach allowed for a clear and consistent mapping of URIs to specific actions within the API. By using the [Route] attribute with dynamic segments such as [Controller] and [Action], the API was able to define routes that directly corresponded to the controller and action names, providing a structured and intuitive way to access different functionalities within the application. The routing used in the application is attribute routing, but another approach can be implemented by using route prefix. Rick-Anderson (2022) stated that frequently, the URIs for routes within a controller share a common prefix. To establish a uniform prefix for all the routes within a controller, the [RoutePrefix] attribute can be utilized. This means that routes in the controller will share the same starting part, making it easier to organize and manage API endpoints.

## RESTful Request Methods

The application uses multiple RESTful requests methods to handle and manipulate data, these methods include are Get, Post, and Delete and Put.

### Get Hello

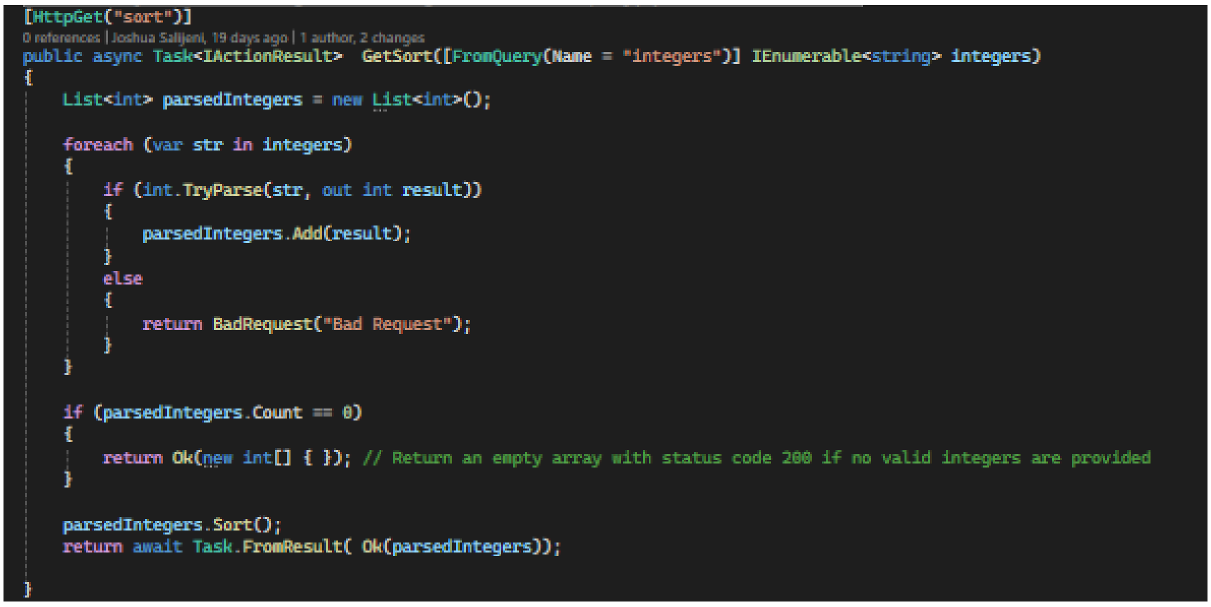


```
/// <summary> gets a response of "Hello World" when the ap
[HttpGet("hello")]
0 references | Joshua Salljeni, 19 days ago | 1 author, 2 changes
public async Task<IActionResult> GetHello()
{
    return await Task.FromResult( Ok("Hello World"));
}
```

*Figure 2 Getting the string with Get method.*

The request method gets a string and returns it to the user when the server is run.

## Get Sort

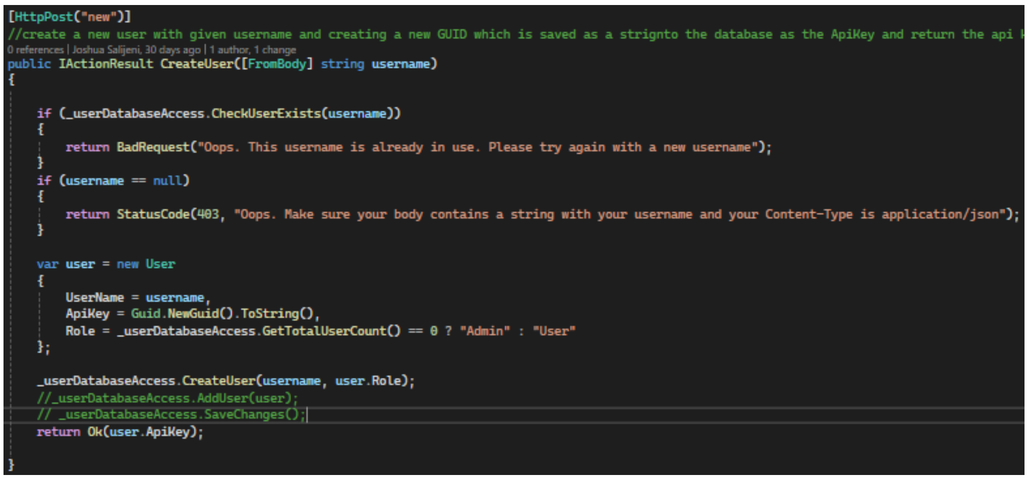


```
[HttpGet("sort")]
0 references | Joshua Salijeni, 19 days ago | 1 author, 2 changes
public async Task<IActionResult>  GetSort([FromQuery(Name = "integers")] IEnumerable<string> integers)
{
    List<int> parsedIntegers = new List<int>();

    foreach (var str in integers)
    {
        if (int.TryParse(str, out int result))
        {
            parsedIntegers.Add(result);
        }
        else
        {
            return BadRequest("Bad Request");
        }
    }

    if (parsedIntegers.Count == 0)
    {
        return Ok(new int[] { }); // Return an empty array with status code 200 if no valid integers are provided
    }

    parsedIntegers.Sort();
    return await Task.FromResult( Ok(parsedIntegers));

}
```

*Figure 3 Getting sorted list with Get method.*

The method gets the user inputted list of numbers and sorts them and finally returns the sorted list back to the user.

## Creating a User

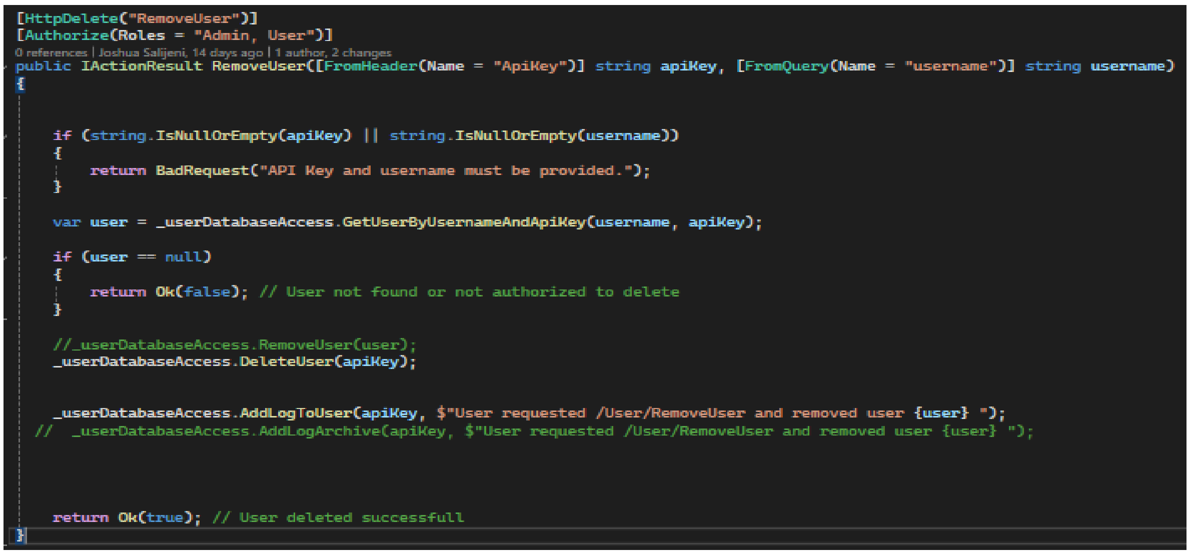


```
[HttpPost("new")]
//create a new user with given username and creating a new GUID which is saved as a strignto the database as the ApiKey and return the api
0 references | Joshua Salijeni, 30 days ago | 1 author, 1 change
public IActionResult CreateUser([FromBody] string username)
{

    if (_userDatabaseAccess.CheckUserExists(username))
    {
        return BadRequest("Oops. This username is already in use. Please try again with a new username");
    }
    if (username == null)
    {
        return StatusCode(403, "Oops. Make sure your body contains a string with your username and your Content-Type is application/json");
    }

    var user = new User
    {
        UserName = username,
        ApiKey = Guid.NewGuid().ToString(),
        Role = _userDatabaseAccess.GetTotalUserCount() == 0 ? "Admin" : "User"
    };

    _userDatabaseAccess.CreateUser(username, user.Role);
    //_userDatabaseAccess.AddUser(user);
    // _userDatabaseAccess.SaveChanges();
    return Ok(user.ApiKey);

}
```

*Figure 4 Creating a user using the http Post method.*

A user is created using the Post method, the method takes in username as input and prompts the server to create a new user with an Api key and a role.

## Deleting User

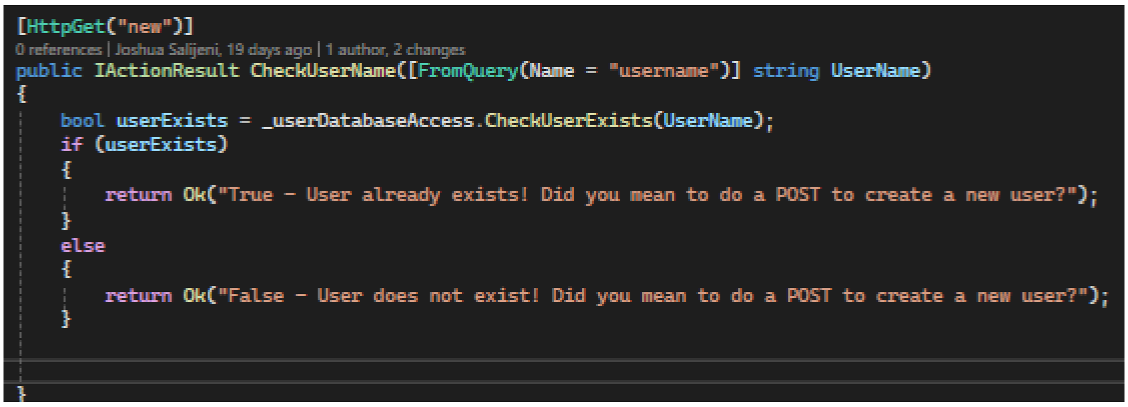


```
[HttpDelete("RemoveUser")]
[Authorize(Roles = "Admin, User")]
0 references | Joshua Salijeni, 14 days ago | 1 author, 2 changes
public IActionResult RemoveUser([FromHeader(Name = "ApiKey")] string apiKey, [FromQuery(Name = "username")] string username)
{


    if (string.IsNullOrEmpty(apiKey) || string.IsNullOrEmpty(username))
    {
        return BadRequest("API Key and username must be provided.");
    }

    var user = _userDatabaseAccess.GetUserByUsernameAndApiKey(username, apiKey);

    if (user == null)
    {
        return Ok(false); // User not found or not authorized to delete
    }

    //_userDatabaseAccess.RemoveUser(user);
    _userDatabaseAccess.DeleteUser(apiKey);


    _userDatabaseAccess.AddLogToUser(apiKey, $"User requested /User/RemoveUser and removed user {user} ");
  //  _userDatabaseAccess.AddLogArchive(apiKey, $"User requested /User/RemoveUser and removed user {user} ");



    return Ok(true); // User deleted successfull
}
```

*Figure 5 Deleting user using the http Delete method.*

After a user has been created and saved in the database when the method is called a user with the role with admin passes the API key of the user and the username that needs to be deleted. The request deletes the user in the database if they exist.

## Get User

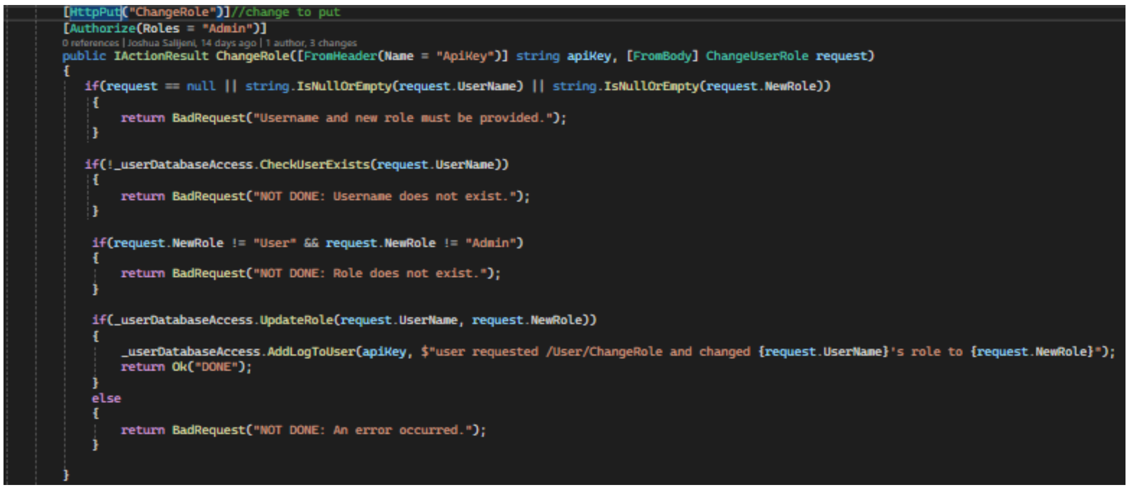


```
[HttpGet("new")]
0 references | Joshua Salijeni, 19 days ago | 1 author, 2 changes
public IActionResult CheckUserName([FromQuery(Name = "username")] string UserName)
{
    bool userExists = _userDatabaseAccess.CheckUserExists(UserName);
    if (userExists)
    {
        return Ok("True - User already exists! Did you mean to do a POST to create a new user?");
    }
    else
    {
        return Ok("False - User does not exist! Did you mean to do a POST to create a new user?");
    }


}
```

*Figure 6 Getting user using the http method.*

After a user is created, the user can be able to get the existing user by calling the get user request with a username they would like to get. The Request checks for the existing user in the database and returns the results as required.

## Update User Role

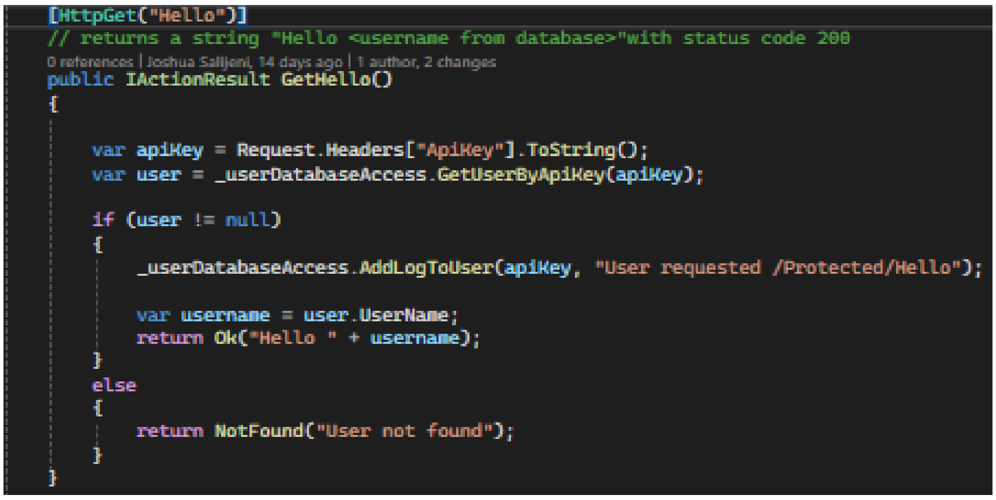


```
[HttpPut("ChangeRole")]//change to put
[Authorize(Roles = "Admin")]
0 references | Joshua Salijeni, 14 days ago | 1 author, 3 changes
public IActionResult ChangeRole([FromHeader(Name = "ApiKey")] string apiKey, [FromBody] ChangeUserRole request)
{
    if(request == null || string.IsNullOrEmpty(request.UserName) || string.IsNullOrEmpty(request.NewRole))
    {
        return BadRequest("Username and new role must be provided.");
    }

    if(!_userDatabaseAccess.CheckUserExists(request.UserName))
    {
        return BadRequest("NOT DONE: Username does not exist.");
    }

     if(request.NewRole != "User" && request.NewRole != "Admin")
     {
        return BadRequest("NOT DONE: Role does not exist.");
     }

     if(_userDatabaseAccess.UpdateRole(request.UserName, request.NewRole))
     {
        _userDatabaseAccess.AddLogToUser(apiKey, $"user requested /User/ChangeRole and changed {request.UserName}'s role to {request.NewRole}");
        return Ok("DONE");
     }
     else
     {
        return BadRequest("NOT DONE: An error occurred.");
     }

}
```

*Figure 7 Updating user role using the http Put method.*

The user with the role of admin in the database is only granted authorization to update the role of a user. By providing the API key in the header and username with the role to be changed to in the body, the put request will update the user role in the database.

## Protected Get Hello

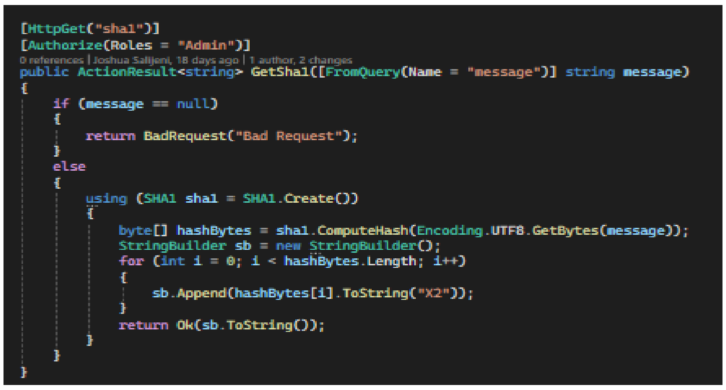


```
[HttpGet("Hello")]
// returns a string "Hello <username from database>"with status code 200
0 references | Joshua Salijeni, 14 days ago | 1 author, 2 changes
public IActionResult GetHello()
{

    var apiKey = Request.Headers["ApiKey"].ToString();
    var user = _userDatabaseAccess.GetUserByApiKey(apiKey);

    if (user != null)
    {
        _userDatabaseAccess.AddLogToUser(apiKey, "User requested /Protected/Hello");

        var username = user.UserName;
        return Ok("Hello " + username);
    }
    else
    {
        return NotFound("User not found");
    }
}
```

*Figure 8 Getting the username of a user and greeting them.*

The method gets the user from the database and returns a message back to user once the action is complete.

## Protected Get SHA1

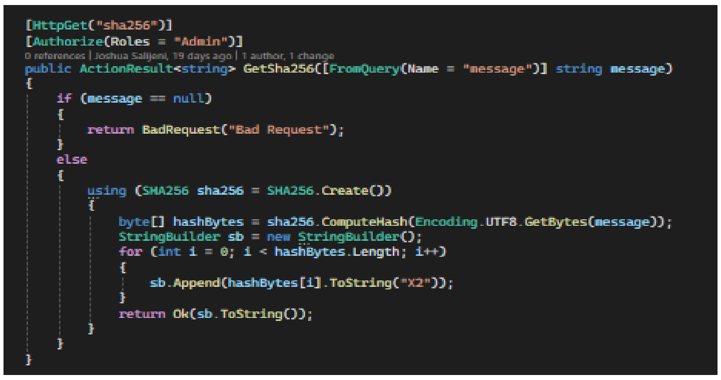


```
[HttpGet("sha1")]
[Authorize(Roles = "Admin")]
0 references | Joshua Salijeni, 18 days ago | 1 author, 2 changes
public ActionResult<string> GetSha1([FromQuery(Name = "message")] string message)
{
    if (message == null)
    {
        return BadRequest("Bad Request");
    }
    else
    {
        using (SHA1 sha1 = SHA1.Create())
        {
            byte[] hashBytes = sha1.ComputeHash(Encoding.UTF8.GetBytes(message));
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < hashBytes.Length; i++)
            {
                sb.Append(hashBytes[i].ToString("X2"));
            }
            return Ok(sb.ToString());
        }
    }
}
```

*Figure 9 Getting the hashed message using get method.*

The method takes in the users input and returns the SHA1 hashed message to the user. The get request is used to get this hashed message back to the user.

## Protected Get SHA256

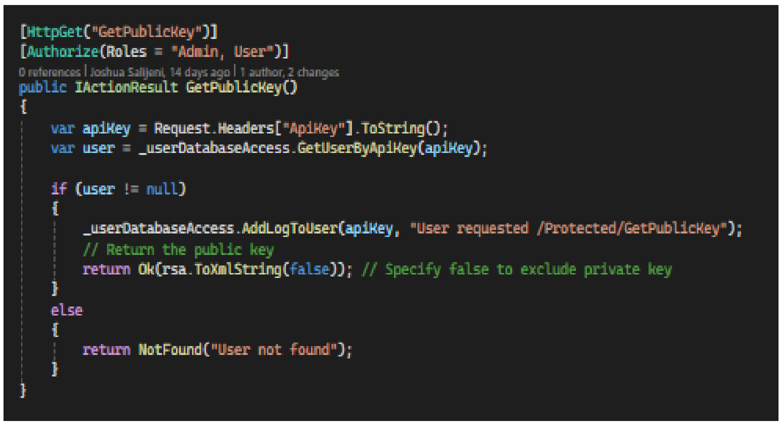


```
[HttpGet("sha256")]
[Authorize(Roles = "Admin")]
0 references | Joshua Salijeni, 19 days ago | 1 author, 1 change
public ActionResult<string> GetSha256([FromQuery(Name = "message")] string message)
{
    if (message == null)
    {
        return BadRequest("Bad Request");
    }
    else
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(message));
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < hashBytes.Length; i++)
            {
                sb.Append(hashBytes[i].ToString("X2"));
            }
            return Ok(sb.ToString());
        }
    }
}
```

*Figure 10 Getting hashed message with Get method.*

The method takes in the users input and returns the SHA256 hashed message to the user. The get request is used to get the hashed message back to the user.

## Protected Get SHA256



```
[HttpGet("GetPublicKey")]
[Authorize(Roles = "Admin, User")]
0 references | Joshua Salijeni, 14 days ago | 1 author, 2 changes
public IActionResult GetPublicKey()
{
    var apiKey = Request.Headers["ApiKey"].ToString();
    var user = _userDatabaseAccess.GetUserByApiKey(apiKey);

    if (user != null)
    {
        _userDatabaseAccess.AddLogToUser(apiKey, "User requested /Protected/GetPublicKey");
        // Return the public key
        return Ok(rsa.ToXmlString(false)); // Specify false to exclude private key
    }
    else
    {
        return NotFound("User not found");
    }
}
```

*Figure 11 Getting public Key using Get Method.*

The method gets the public key using the user API key and returns this key using the get request method.

## Using User API Key and Middleware

The client and server used an API key given to a user when they are created in the database, this API key had various uses in the program and these included authentication and authorization. The API key was used to authenticate a user when performing actions in the server, ensured that the user sending a request existed in the database. When a user wanted to perform some action on the server, they needed to have authorization to do so. for instance, when a user wants to update a role of another user, they needed to have authorization, and this was confirmed by checking the API of the user to confirm their role and proceed with the action requested. The application used middleware for authentication, authorization, and data access. The middleware handled security checks to ensure that only authenticated and authorized users had access to specific API endpoints. For the data access the middleware it handled data retrieval and manipulation.

The use of the API key is a good option for identifying users as it offers benefits such as tracking user activities and it is a good solution to the implemented application as it offers the ability to easily authorize users perform specific tasks. In the real world this API can be kept safe by implementing measures like key encryption, key validation, and key regeneration.

## Entity Framework

Microsoft entity framework was used in the application by creating a model class which represented data to be stored in the database followed by a database context which allowed database interactions. The database was kept in sync with the model class by using migration to update the schema and finally perform CRUD (Create, Read, Update, Delete) operations.

The code is loosely coupled by using dependency injection and a code first approach. Dependency injection was implemented by injecting the database context to the services this ensured that functionality can be changed without compromising an entire class. Instead of generating a schema first and then writing code for it, the code first approach ensured that classes were created first which defined what the schema would look like.

## Reflection

The program consisted of 14 task to complete and up to 13.5 of them were fully completed.

- From task 1 to 12 all tasks were completed and fully functional.
- Task 13 was left half completed with the ability to log deleted request left out.
- The beginning part of the API URL could have been stored in constant variables
- Task 14 was not completed at all.

During development a few problems were encountered, some were fixed, and some were not. The first problem that was encountered was the inability to create a user to the database due to the format of the API string and the middle ware to facilitate the connection to the database. This was fixed by implementing middleware and changing the API variable from Guid to string.

The second problem was the inability to keep track of the delete request and log this activity to the new tables. This problem was not fixed and finally task 14 was not completed due to its level of complexity.

## Conclusion

RESTful API's and entity framework can be used to build data driven applications, the implementation of dependency injection and code first approach ensures that code is loosely coupled and facilitates easy management and maintenance to an application. Despite encountering challenges during development, such as formatting issues and incomplete task implementations, the iterative nature of the process allowed for valuable insights into problem-solving and system refinement.

## References

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L. and Percannella, G., 2016. REST APIs: A large-scale analysis of compliance with principles and best practices. In *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16* (pp. 21-39). Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-38791-8_2

*Stateful vs stateless* (2023) *Red Hat - We make open-source technologies for the enterprise*. Available at: https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless

Rick-Anderson (2022) *Attribute routing in ASP.NET web API 2*, *Microsoft Learn*. Available at: https://learn.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2