DISTRIBUTED SYSTEMS API

Distributed Systems ACW

2024/25

You have been hired as a distributed systems programmer after Gribbald, their previous expert disappeared one foggy night. You've been promised that your first job for them won't be very hard and, thankfully, it's really not. The task is to develop a client/server, using ASP.NET Core Web API, which provides a number of *extra secret* security and encryption services… sort of.

You're lucky though. Gribbald was working on this project before you and has already created an outline solution with some 'TODO' regions, based on the original specification. They have also broken down the specification into tasks for you, which should make your life even easier!

On your first day you were handed this specification:

- The server *must* be able to handle multiple client requests simultaneously.
- The server *must* use Entity Framework, Code First, to create and manage a local database of Users (that will, in the future be moved over to a production database).
- The client must be able to get a **TalkBack/Hello** message from the server. The server will respond with "Hello World".
- The client must be able to send a **TalkBack/Sort** message as a get request to the server with an array of integers as parameters. The server will sort the integers into ascending order and return the sorted array to the client, because – well, sorting data is tedious.
- The client must be able to send a **User/New** get request to the server which has a username string as a parameter. The server must return a string identifying if the username already exists in the database.
- The client must be able to send a **User/New** post request to the server which has a username string in the request body. The server must create a new user, generate a new GUID as an API Key, save the user to the database and return the API Key to the user. If this is the first user they should be saved as Admin role, otherwise just with User role.
- The client must be able to send a **User/RemoveUser** delete request to the server with an API Key in the header and a string username in the URI. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must check that the username and API Key are the same user and if they are, it must delete this user from the database.
- The client must be able to send a **User/ChangeRole** Post request to the server with an API Key in the header which links to an Admin role user, a string username in the body and a string role in the body. If the server receives this request, it must check to see if the API Key is in the database and whether the role is Admin, if it is, it must update the role for the given username to the role provided (User or Admin)
- The client must be able to send a **Protected/Hello** get request to the server with an API Key in the header. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must get the username associated with the API Key and send back "Hello <username>" (e.g. "Hello UserOne").
- The client must be able to send a **Protected/SHA1** get request to the server with an API Key in the header and a string message as a parameter. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must compute the SHA1 hash of the message and return it in hexadecimal form to the client.
- Somebody told the boss that SHA256 is more secure than SHA1 so the client must be able to send a **Protected/SHA256** get request to the server with an API Key in the header and a string message as a parameter. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must compute the SHA256 hash of the message and return it in hexadecimal form to the client.

- The client must be able to send a **Protected/GetPublicKey** get request to the server with an API Key in the header. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must send back its RSA public key.
- The client must be able to send a **Protected/Sign** request with an API Key in the header and a string message as a parameter. If the server receives this request, it must check to see if the API Key is in the database and, if it is, it must digitally sign the message with its private RSA key and send the signed message back to the client. The client must then be able to verify that the server signed the message by using the server's public RSA key.
- Finally, the client must be able to send a **Protected/Mashify** get request to the server with an API Key in the header which links to an Admin role user, and three strings in the body comprising:
    - A string of text, encrypted using the server's public RSA key
    - A symmetric key (using AES encryption), encrypted using the server's public RSA key
    - An IV (initialization vector) for the symmetric key, also encrypted using the server's public RSA key

  If the server receives this request, it must…
    - check to see if the API Key is in the database and whether the role is Admin,
    - if it is, it must decrypt all three parameters using its private RSA key.
    - It must then 'mashify' the string it was given (see task 14), encrypt the mashified string using the client's symmetric (AES) key and IV and
    - finally, it must send the newly encrypted string back to the client.

  The client must then be able to…
    - decrypt the string using its symmetric (AES) key and IV and
    - finally, output the new mashified string to the console.

Your task is to follow the next instructions carefully and develop a **console-based client** and **Web API-Based server** with a **Code First Entity Framework Database**.

- You **must** use the skeleton solution as your starting point, and
- You **must** ensure that you carefully follow instructions on request types and names, parameter naming, responses and response types – if you do not conform to these instructions, some of the marking tools will not be able to find your code and you will receive a mark of 0 in affected areas.

Please do not, under any circumstances, publish your work on a public source code repository – even after you leave the University. We have had issues in the past with students using public repositories (e.g. on GitHub) and their work being plagairised (e.g. by people doing reassessment). In these circumstances it becomes very difficult to ascertain who was cheating and both parties may be penalised for collusion under the academic misconduct policy, which can have severe academic consequences. Furthermore, the skeleton code remains the intellectual property of the University of Hull and you are not permitted to share this publicly. You may, however, use any of the solution (including the skeleton code) in your future work and share your solution privately (e.g. a repository set to private) with prospective employers, etc., as you wish.

Instructions

Download the skeleton solution from Canvas and unzip it. You *may* delete or modify *any* of the code that has already been written if you want, but for the most part you will only need to add to it. Note how there are a number of regions and comments that identify tasks by number – use these as a guide to identify where you might want to add your code.

The Server

The server that you have been given in the skeleton solution is an ASP.NET Web API. You may find it useful to refer to the Microsoft documentation on Web API. You should also recognise it from labs and lecture material.

There are a few things you should look at before you start working on the server:

1. Open *TalkBackController.cs* – Gribbald started writing this controller before they mysteriously disappeared. It will contain two get request actions for /API/**TalkBack/Hello** and /API/**Talkback/Sort**. Task1 will be to complete these methods, but they should also be a guide to help you get started – you may change any of the code you feel needs to be changed.
2. Open *Pipeline->Auth->CustomAuthenticationHandler.cs* – this is custom authentication middleware that is added inside `Program.cs`. This means that *whenever* an HTTP request is made, this method will run *before* the request gets to your controller. You will need to modify this method in TASK5 to verify that the API Key in the header is correct and authenticate the user. There is a similar *CustomAuthorizationHandler* – middleware that runs immediately after authentication if the authentication succeeds. You will need to make changes to this code in TASK6. The use of these handlers is directly linked to the Authorize attribute.
3. Notice the error handling middleware and filter inside *Pipeline->ErrorHandling*. These
4. Open *BaseController.cs* – you should note four things:
    a. This is an abstract class. You cannot make an instance of this controller.
    b. It inherits from ControllerBase and has the attribute ApiController – all Controllers need to descend from ControllerBase and have the attribute ApiController.
    c. It stores a protected instance of the Database Context that you'll need to use for your data access.
    d. The route mapping has been changed to *`api/{controller}/{action}`* which will allow you to call actions inside your controller (e.g. *`/API/TalkBack/Hello`*, where `TalkBack` is the controller and `Hello` is the action).

    `BaseController` is an abstract base class you should inherit from to retain the above functionality in your controllers. If you open *TalkbackController.cs* you can see that it inherits from BaseController. To make your life easier use this as a template for any new controllers you create.

Before you start working on your server, you may find it useful to use a tool which allows you to craft requests (with control over the header/body/URI/etc.) send them to your server for debugging purposes and receive RESTful responses. This means you won't need to have a working client to test your server out. A suitable (and free) tool is PostMan: https://www.getpostman.com/apps Follow the module instructions on setting up a student account.

Finally, make sure you use the **test server**. This server responds as per this coursework specification. If your server responds in exactly the same way as the test server then you are likely to get a good mark for the server elements of this coursework. If your client works properly with the test server then you can be confident it is working properly too. You may use Postman to analyse the responses from the test server before you start producing your server. ***There's more info at the end of TASK10*** but the test server can be accessed by pointing your client to:

http://150.237.94.9/<myUniqueCode>/
e.g. http://150.237.94.9/1234567/Api/TalkBack/Hello

Your unique 7 digit code should have been distributed to you via email already but if you have not received this then email me: john.dixon@hull.ac.uk

Tasks

The following tasks are designed to help you identify exactly what to do to meet the specification and they give a logical order for doing it too. You must ensure that your server and client respond *exactly* as specified to get the marks for that section, so read this specification very carefully. Most tasks also have a section dedicated to testing so that you can test your solution against expected results. If your server and client do not respond exactly as specified you will not receive marks for the corresponding marking criteria.

Task1:

When the client makes a get request for api/**TalkBack/Hello** or api/**TalkBack/Sort**, the talkback controller must handle the responses.

Complete these methods so that they offer the responses detailed in the specification.

Once you have created these methods, right-click the server project, select Debug and click Start New Instance. This should fire up IIS, open your browser and take you to a web page with an address similar to localhost:53415 where 53415 is your portnumber. Identify what this port number is as it will be required in your client and for testing.

For your personal testing you can identify if your server is working by sending a get request with the URI (replace <portnumber> with your port number) of:

For **Hello**: `localhost:<portnumber>/api/talkback/hello`
Must return `"Hello World"` in the body of the result with a status code of OK (200)

For **Sort**: `localhost:<portnumber>/api/talkback/sort?integers=8&integers=2&integers=5`
Must return `[2,5,8]` in the body of the result with a status code of OK (200)

If there are no integers submitted, must return [] in the body with a status code of OK (200)

If the submitted integers are invalid (e.g. a char is submitted) the server must return with a with a status code of BAD REQUEST (400)

Task2:

In order to work with databases, you have been asked to use Entity Framework Core.

Gribbald has already put all of the references, etc. that you will need into the project. You just need to define a User class.

Open *Models->User.cs*, where you will find the Task2 region. Complete the User class with:

- An empty constructor
- A public string ApiKey property (which is the unique database key and is the API Key for the user)
- A public string UserName property (which is the username of the user)
- A public string or enum Role property (which saves the role of the user)

UserContext.cs has been created for you already, but to create the database correctly you'll need to generate a migration*.

*You may first need to set the project directory in the Package Manager Console

Task3:

You should keep your controller classes only loosely coupled to the database access code. Ensuring your code is nicely abstracted is good coding practice as it will allow you to easily swap out the data access logic in the future without editing your controllers. You may like to create these data access functions now.

For the next tasks, you may need database access such as:

1. Create a new user, using a username given as a parameter and creating a new GUID which is saved as a string to the database as the ApiKey. This must return the ApiKey or the User object so that the server can pass the Key back to the client.
2. Check if a user with a given ApiKey string exists in the database, returning true or false.
3. Check if a user with a given ApiKey and UserName exists in the database, returning true or false.
4. Check if a user with a given ApiKey string exists in the database, returning the User object.
5. Delete a user with a given ApiKey from the database

Hint: The BaseController already contains an instance of UserContext

Consider: What happens in your solution if two admin users simultaneously try to change a user's role?

Task4:

When the client makes a api/**User/New** get request or api/**User/New** post request, the server must be able to handle them and provide a response.

Create a new controller called UserController. The easiest way to do this is by right-clicking on Controllers and adding a new, Empty API Controller. You may want to inspect the existing TalkBack Controller to help set up your new controller – note it's base type, constructor and routing.

Both of these actions have the name 'New' but one is a GET request that takes its parameter from the URI and the other is a POST request that takes a JSON string parameter from the body. The POST request must use a content-type of application/json. Note the difference between a JSON string and a JSON Object with a string.

For your personal testing you can identify if your server is working by sending a get request with the URI (replace <portnumber> with your port number) of:

For **GET**: `localhost:<portnumber>/api/user/new?username=UserOne`

If a user with the username 'UserOne' exists in the database, the server must return `"True - User Does Exist! Did you mean to do a POST to create a new user?"` in the body of the result with a status code of OK (200)

If a user with the username 'UserOne' does not exist in the database, the server must return `"False - User Does Not Exist! Did you mean to do a POST to create a new user?"` in the body of the result with a status code of OK (200).

If there is no string submitted, the server must return `"False - User Does Not Exist! Did you mean to do a POST to create a new user?"` in the body of the result with a status code of OK (200).

For **POST**: `localhost:<portnumber>/api/user/new` with only *"UserOne"* in the body of the request

Should create a new User with the username 'UserOne', generate a new GUID as the user's API Key, and then add the new user to the database. Finally, the server must return the API Key as a string to the client with a status code of OK (200). If this is the first user they should be saved as Admin role, otherwise just with User role.

If there is no string submitted in the body, the result should be `"Oops. Make sure your body contains a string with your username and your Content-Type is Content-Type:application/json"` with a status code of BAD REQUEST (400)

If the username is alrady taken, the result should be `"Oops. This username is already in use. Please try again with a new username."` with a status code of FORBIDDEN (403)

Task5:

The client now has the ability to ask for an API Key, so our server will now need to be able to determine if a request has a valid API Key in its header. A useful way to do this is to use an Authentication Scheme. Ours is a custom Authentication Scheme so we can investige its functionality. Return to *CustomAuthenticationHandler.cs*. You must add code to `HandleAuthenticateAsync` which tries to get the header 'ApiKey', and if it does exist, checks the database to determine if the given API Key is valid. You should use your `UserDatabaseAccess` class that you created in TASK3 to do the database access to loosen your coupling.

If the API Key is valid, you must get the relevant User from your database and set up a:

- Claim of type ClaimTypes.Name, using the user's UserName as the string value
- Claim of type ClaimTypes.Role, using the user's Role as the string value
- ClaimsIdentity with an authentication type of "ApiKey", using an array containing both of your Claims
- ClaimsPrinciple, using the ClaimsIdentity above.

Finally, you must create an AuthenticationTicket using the ClaimsPrinciple and the scheme name (you can get this using the property: `this.Scheme.Name).` You can now use the ticket to create a Success AuthenticateResult and return as a Task.

If this is working, you should be able to use attributes such as `[Authorize(Roles = "Admin")]` or `[Authorize(Roles = "Admin, User")]` to authorise requests which require a valid API key to be present.

If a user is not found, return a Fail AuthenticateResult and pass a 401 error with the JSON string `"Unauthorized. Check ApiKey in Header is correct."` back to the user (see HandleChallengeAsync).

Task6:

So far we haven't used any of our Roles. Before we can do this we'll need to check that our users have the roles they are supposed to. We *are* already looking at whether or not they have a valid API key (Task5), and we're using that easily by applying an attribute, so we'll use a filter to modify the response.

Most of the filter has already been written. Open *CustomAuthorizationHandler.cs* and modify the code so that, when the action requires a user to be in Admin role ONLY (e.g. `[Authorize(Roles = "Admin")]`) and the user does not have the Admin role, you return a Forbidden status (403) with the message: `"Forbidden. Admin access only."`. You will need the injected IHttpContextAccessor for this.

Task7:

Add to your UserController a method to handle an api/**User/RemoveUser** DELETE request.

- A user with role User *or* Admin should be able to use this request.
- The client must send its API Key in the header and a string username in the URI.

If the server receives this request, it must extract the ApiKey string from the header to see if the API Key is in the database and, if it is, it must check that the username and API Key are the same user and if they are, it must delete this user from the database. You should probably use your `UserDatabaseAccess` class that you created in TASK3 to do the database access.

This method must return a Boolean value only. If a user has been deleted, the server must return `true`, otherwise, the server must return `false`. In both cases, the server must return a status code of OK (200).

For your personal testing you can identify if your server is working by sending a delete request with the URI (replace <portnumber> with your port number and <username> with a username) of:

RemoveUser: `localhost:<portnumber>/api/user/removeuser?username=<username>` with an ApiKey in the header of the request
Must return `true` if the ApiKey and username match, are valid, and a user has been deleted or `false` if not.

Task8:

Inside UserController Add the api/**User/ChangeRole** method.

This method must only be accessible to users who are authorised by API key *and* have the role of Admin.

Update the role for the given username to the role provided (User or Admin).

The body should contain a JSON object in the form (where <username> is the given username string and <role> is the given role string):

```
{  "username":<username>, "role":<role>  }
```

For your personal testing you can identify if your server is working by sending a post request with the URI (replace <portnumber> with your port number) of:

ChangeRole: `localhost:<portnumber>/api/user/changerole` with an ApiKey in the header of the request, a string username in the body and a string role in the body

If success: Must return `"DONE"` in the body of the result, with a status code of OK (200)
If username does not exist: Must return `"NOT DONE: Username does not exist"` in the body of the result, with a status code of BAD REQUEST (400)
If role is not User or Admin: Must return `"NOT DONE: Role does not exist"` in the body of the result, with a status code of BAD REQUEST (400)
In all other error cases: Must return `"NOT DONE: An error occured"` in the body of the result, with a status code of BAD REQUEST (400)

Task9:

Create a new ProtectedController. Add the api/**Protected/Hello** method, api/**Protected/SHA1** method and api/**Protected/SHA256** method.

All of these requests must be authorised (User *or* Admin role) and all three must return strings to the client.

You may use the .NET `SHA1` and `SHA256` types for SHA1 and SHA256 hashing respectively.

Both SHA1 and SHA256 methods must take a string `message` from the URI and both must return the hexadecimal hash as a string with no additional characters (e.g. no delimiters like `-`)

For your personal testing you can identify if your server is working by sending a get request with the URI (replace <portnumber> with your port number) of:

For **Hello**: `localhost:<portnumber>/api/protected/hello` with an ApiKey in the header of the request
Must return `"Hello <UserName>"` in the body of the result, where UserName is the User's UserName from the database, with a status code of OK (200). E.g. "Hello UserOne".

For **SHA1**: `localhost:<portnumber>/api/protected/sha1?message=hello`
Must return `"AAF4C61DDCC5E8A2DABEDE0F3B482CD9AEA9434D"` in the body of the result with a status code of OK (200)

If there is no message string submitted, the result should be `"Bad Request"` with a status code of BAD REQUEST (400)

For **SHA256**: `localhost:<portnumber>/api/protected/sha256?message=hello`
Must return `"2CF24DBA5FB0A30E26E83B2AC5B9E29E1B161E5C1FA7425E73043362938B9824"` in the body of the result with a status code of OK (200)

If there is no message string submitted, the result should be `"Bad Request"` with a status code of BAD REQUEST (400)

Task10:

If you have not already started, begin to write the client (in project *DistSysAcwClient*) now as the next server tasks are the most difficult and you should get the basic functionality of your client working before you attempt them.

The client must be a console application. It would make sense to use System.Net.Http.HttpClient. You don't need to use HttpClient but your client must be able to perform all of the same functions. Your client must request and expect a JSON response type.

When your client is opened it should show: "Hello. What would you like to do?" and wait for user input.

Below is what the client must be able to do…

Input String from Console Window	Example	Example Request	Response	Client Function	Output to Console Window
TalkBack Hello	TalkBack Hello	localhost:<portnumber>/api/talkback/hello	string	None	Response string
TalkBack Sort <integer array>	TalkBack Sort [6,1,8,4,3]	localhost:<portnumber>/api/talkback/sort?integers=6&integers=1&integers=8&integers=4&integers=3	int[]	None	Response as a string
User Get <name>	User Get UserOne	localhost:<portnumber>/api/user/new?username=UserOne	string	None	Response string
User Post <name>	User Post UserOne	localhost:<portnumber>/api/user/new with "UserOne" in the request body	string	Check status of response. Store API Key and username as variables if response: OK	"Got API Key" if OK, else print response string.
User Set <name> <apikey>	User Set UserOne 004b620d-a523-4b1b-8bdc-857d8a5541b9	None – this function is only in the client	n/a	Store given API Key and username as variables	"Stored"
User Delete	User Delete	Client must get the locally stored username and ApiKey. If they don't yet exist the console must print "You need to do a User Post or User Set first". localhost:<portnumber>/api/user/removeuser?username=<username> (with <username> in the URI) with ApiKey:<apikey> in the header	Boolean	None	"True" if delete succeeded, otherwise "False"

User Role <username> <role>	User Role UserOne Admin	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first
		
		localhost:<portnumber>/api/user/changerole
		
		with ApiKey:<apikey> in the header and with the JSON object:
		{
		"username" : "UserOne",
		"role" : "Admin"
		}
		in the request body
			string	None	Response as a string

Protected Hello	Protected Hello	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first".	string	None	Response string

		localhost:<portnumber>/api/protected/hello with ApiKey:<apikey> in the header			
Protected SHA1 <message>	Protected SHA1 Hello	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first localhost:<portnumber>/api/protected/sha1?message=Hello with ApiKey:<apikey> in the header	string	None	Response String
Protected SHA256 <message>	Protected SHA256 Hello	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first". localhost:<portnumber>/api/protected/sha256?message=Hello with ApiKey:<apikey> in the header	string	None	Response String

If an error is returned, from your server, you must write out the error to the console or write out the given error string if one is specified in the Output to Console Window column. The Console **must output error messages** and **never crash**.

The client application should be asynchronous. The console must write **"...please wait..."** to the console window as the request is sent and then write the output once the asynchronous Task has been completed. After the output has been written, the console must write (on a new line) **"What would you like to do next?"**. When a new input is entered**, the Console Window must be cleared**.

If the user enters **"Exit"** (without quotes), the console must close.

Whilst you do not have to, you may wish to have the console read/write to a file to save a valid UserName and ApiKey, so that you don't have to do a User Post or User Set every time you restart your application to test any of the requests that require a username or ApiKey. If you do this, you should ensure that the software will run on another computer.

Hint: you may find some of the information on this page useful: https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/console-webapiclient

TEST SERVER

As specified previously in this document, there is a test server from which to test your client and check that your server responds in the same way. This test server acts as the 'gold standard'. If your server responds in the same way you can be certain that you will get the associated mark(s). You are *very much* encouraged to use the test server as you will receive no marks for client functionality which does not work against the test server (even if it works against *your* server).

I would very much advise you to create a single place in your client-side code (e.g. a const) which stores the first part of your URL and then allows you to revert to using your server by changing only one line or config file.

The test domain is: http://150.237.94.9/<*myUniqueCode*>/

e.g. http://150.237.94.9/1234567/Api/TalkBack/Hello

Your unique 7 digit code should have been distributed via email but if you have not received this or you find a bug in the test server prototype then please email me: john.dixon@hull.ac.uk. I would recommend you don't share your code as it provides data isolation - someone else editing your data may interfere with your testing.

You should ensure that when you hand in your solution it is configured to access and use *your local server*, not the test server.

- There is an additional useful command on the test server. You can run this command from any browser or from Postman so you don't need to implement it into your Client:
  http://150.237.94.9/<*myUniqueCode*>/Api/Other/Clear

This command clears all users and logs from your account on the test server, refreshing it back to empty. This may help during testing. You do not need to replicate this command on your server/client unless you want to; there are no marks for other/clear.

Task11:

First, set up an RSA Crypto Service Provider which is configured once and the same across all threads. You ***must not*** make a new key pair for each client but you may generate a new key pair each time the server starts. The private RSA key must be stored securely (it would be a good idea to use the machine key store as we are using Windows).

Now, returning the public key should be as simple as returning the XmlString created by your RSA Crypto Service Provider, containing the public RSA key for your server.

This request must be authorised (User *or* Admin role).

On receiving an api/**Protected/GetPublicKey** get request, the server must check to see if the API Key is in the database and, if it is, it must send back its RSA public key.

Remember not to send over the private key!!

For your personal testing you can identify if your server is working by sending a get request with the URI (replace <portnumber> with your port number) of:

GetPublicKey: `localhost:<portnumber>/api/protected/getpublickey` with an ApiKey in the header of the request
Must return the XmlString containing the public key, which should look something like this:

"<RSAKeyValue><Modulus>rSJEEeLC4H452XFL+taho/473M5OKdfSC/PKNFk55xOx/M5HbDxG9ihoENpazG7OHsGitb0aXfn2qEVzhaaTtUJUKyO+sV2nQ6aaE0rwFUK0XxttFX1ann/d3qNTrlVxWdzygGq8ODn7qzvijcXjR/S2iyEguSNOuwhUO/M98sk=</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>"

Now add the functionality to your client which will allow it to store the public key. You can store the key however you like, but it must be usable when you need to encrypt something to send it to the server.

Input String from Console Window	Example	Request	Response	Client Function	Output to Console Window
Protected Get PublicKey	Protected Get PublicKey	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first". localhost:<portnumber>/api/protected/getpublickey with ApiKey:<apikey> in the header	string	Store the public key Xml (however you like – can be as a variable or to file)	"Got Public Key" if a key was returned or "Couldn't Get the Public Key" if an error occurs

Task12:

If the server receives an api/**Protected/Sign** request with an API Key in the header and a string message as a parameter it must check to see if the API Key is in the database and, if it is, it must digitally sign the message with its private RSA key (using SHA1 hash in the signing), and send the signed message back to the client in a hexadecimal format. The string to sign must be sent as it is given by the user, should be ASCII encoded and should not be modified prior to signing/verification.

The hex string must include dashes as delimiters (-) between each hex value.

This request must be authorised (User *or* Admin role).

For your personal testing you can identify if your server is working by sending a get request with the URI (replace <portnumber> with your port number) of:

Sign: `localhost:<portnumber>/api/protected/sign?message=Hello` with an ApiKey in the header of the request
Must return the signed message, e.g.

"98-4B-61-F0-77-3F-93-81-27-B0-E3-02-C2-38-56-FD-77-57-BC-E5-6E-43-D7-1E-59-EA-F6-E7-9C-8E-F5-F1-1C-06-C1-8B-E4-C8-E0-E5-7D-DE-BE-99-A8-15-C3-8F-F6-2B-1D-43-2D-C2-33-90-17-FB-D4-D7-B9-15-44-77-5C-C0-01-D8-40-76-15-DC-5B-E8-CF-CA-F6-33-1E-C1-DC-4B-CE-D4-38-71-46-6E-97-C0-C4-E8-0A-B0-90-32-55-18-7C-06-1C-AE-0A-06-3F-3D-3B-B5-FE-B4-C7-FA-91-9A-23-1D-04-6A-35-0D-29-78-FA-4C-D1-C8-6A-D5"

Now add the functionality to your client to allow it to make a Sign request. On receipt of the response, the client must verify that the server signed the message by using the server's public RSA key.

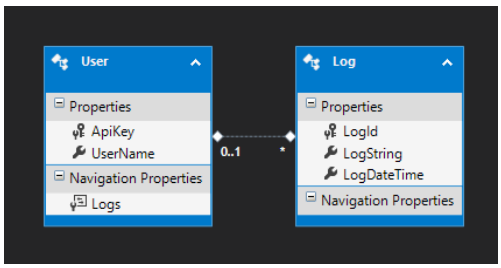
Input String from Console Window	Example	Request	Response	Client Function	Output to Console Window
Protected Sign <message>	Protected Sign Hello	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first". localhost:<portnumber>/api/protected/sign?message=Hello with ApiKey:<apikey> in the header	string	Use the server's public key to verify that the message was signed by the server.	"Message was successfully signed" if signed response was valid, "Message was not successfully signed" if signed response was invalid, "Client doesn't yet have the public key" if the client doesn't know the server public key yet.

TEST YOUR CLIENT AGAINST THE TEST SERVER – IF THIS ELEMENT DOES NOT FUNCTION WHEN TESTED AGAINTST THE TEST SERVER YOU WILL NOT RECEIVE MARKS FOR YOUR CLIENT *OR* SERVER IMPLEMENTATION!

Task13:

The boss has been in, and is really impressed with your work so far, but is worried that the company won't know who is doing what on the server. They have asked you to add a new table to the database that stores logs.

Each User must have a collection of Log entries which must be accessible through `User.Logs` but all logs must be saved in the same table on the database. A log must comprise LogID (unique primary key for the log), LogString (the text describing what happened) and LogDateTime (the date and time of the log). Although you should note that your database should be created using Code-First, the model for this is simply :



You will need to create a new Class (Log), create a virtual ICollection of Logs in your User class and add a Logs DbSet to the context you already have.

You must then go back to all request handlers which require an API key to be passed in the header and add a log to the relevant User identifying what request they made (e.g. "User requested /Protected/Hello") and giving the current DateTime.Now. You can leave the database to automatically generate the LogId.

You may find it useful to create another public constructor for Log that takes a string and creates a new Log object with the passed-in string and DateTime.Now. Remember that you'll have to add the log to a User's collection of logs and save the database. It would be good practice to do this work in the `UserDatabaseAccess` class you created earlier.

Next, you'll need to create a new database migration and update your database so that you can begin logging.

- Be aware that if a user is deleted, the logs should remain. The logs should still be linked to a user API key (even if the user is deleted) for security reasons. You may decide how to do this but a good option would be to create a new table: Log_Archives.

Finally, you should check everything you've coded so far still works as expected and make any changes if it does not.

Task14:

If you haven't already… you should start writing your report (see TASK15) before attempting this task.

Add a final method to your ProtectedController that handles an api/**Protected/Mashify** get request. The request must be authenticated with an API Key in the header, may only be used by users with an Admin role and must have a JSON object in the body comprising three strings:

- A message, encrypted using the server's public RSA key, in hexadecimal format
- A symmetric key (using AES encryption), encrypted using the server's public RSA key, in hexadecimal format
- An IV (initialization vector) for the symmetric key, also encrypted using the server's public RSA key, in hexadecimal format

These three hex strings must include dashes as delimiters (`-`) between each hex value.

Note that you must use the padding mode `RSAEncryptionPadding.OaepSHA1` for encryption and decryption using RSA*.

You should add logging for this request (but you must not compromise any encrypted data in your log).

When the server receives this request, it must: check to see if the API Key is in the database and, if it is, it must:

- Decrypt all three parameters using its private RSA key.
- Then 'mashify' the string it was given (see below)
- Then encrypt the mashified string using the client's symmetric (AES) key and IV.

Finally, it must send the newly encrypted string back to the client as a hexadecimal string.
This hex string must also include dashes as delimiters (`-`) between each hex value.

Mashify-ing a string:

To mashify a string, follow the below steps:

1. Convert all vowels into the upper-case character 'X'
2. Reverse the string

Ensure that you keep all spaces, symbols and upper/lower case characters.

Example mashified strings:

Input string: *Hello World*
Mashified: *dlrXW XllXH*

Input string: *I ate 12 cucumbers in one hour! Now I have a stomach ache!*
Mashified: *!XhcX hcXmXts X XvXh X wXN !rXXh XnX nX srXbmXcXc 21 XtX X*

* Why OaepSHA1? Wouldn't OaepSHA256 or something be more secure? Yes it would. However, there are a few ways to implement the RSA algorithm in .NET Core. and using OaepSHA1 should work for all of them – so hopefully this minimises any clashes between implementations.

For your personal testing, I very much recommend that you first write the client side in the server so that you can easily insert breakpoints and test it before you need to start sending 'stuff' over the network. Otherwise, you will likely only be able to identify if your server is working by using your client to send a get request.

Here is an example to show you what the request/response looks like, although it won't work for you as it will use a different RSA key pair from your solution.

Mashify: `localhost:<portnumber>`/api/protected/mashify
with an ApiKey in the header of the request
and JSON object with three strings in the body of the request:
{"EncryptedString":"7B-05-B3-A2-80-8A-EB-5B-7B-7E-BB-F9-0A-5D-81-B1-B2-82-B2-52-5E-CA-FC-12-54-78-FE-6D-23-A4-E0-DE-BD-36-2B-1F-44-5C-EB-BC-90-AF-60-02-FD-6F-FE-BF-27-37-FB-0F-04-64-C1-5E-61-CB-59-CA-20-9D-3B-F6-67-8C-CE-A9-3E-AE-D7-EB-48-C0-D9-C1-B5-6C-27-90-06-9C-23-1D-04-5E-B4-E1-EA-03-87-CB-E6-A7-DE-49-85-3E-30-D1-1E-54-4E-88-91-1A-49-60-D8-12-41-62-46-3E-84-F5-C5-E3-1E-B2-5A-B6-F2-04-2B-0C-7D-43",
"EncryptedSymKey":"C7-DC-29-E8-36-E7-61-65-5E-BB-CE-CD-63-6B-58-EF-C3-46-5B-72-9E-D2-51-78-B8-C4-6A-C8-D6-21-D0-D6-F8-DB-47-FD-8E-99-DC-D7-14-A1-32-05-BD-BF-61-28-EF-58-46-69-96-59-98-39-E1-05-4D-12-33-62-F3-A1-F7-C9-00-4C-A8-D1-46-44-DB-09-E8-E4-F9-F5-B6-1E-C5-D9-15-7F-09-5F-D5-D3-2F-6E-BF-DB-D4-69-77-0E-6D-5B-C7-6C-E9-E1-54-E6-6C-A1-0A-FD-DF-69-4E-65-2A-CE-86-1A-61-DA-F8-A5-1B-85-E2-39-ED-01-C3-A3", "EncryptedIV":"D5-F0-CE-E8-C1-CE-F0-2E-98-39-10-40-F2-B0-C2-A8-90-46-6A-06-2C-63-CE-B1-40-94-AF-2F-C5-C9-20-4E-13-21-2D-6A-FC-06-E7-65-1B-F6-2E-51-6F-A6-F4-92-97-62-9D-FF-EC-A6-99-66-AE-12-5D-E2-DC-69-45-C3-5C-33-A6-DC-15-E3-DC-22-C0-25-23-C3-EE-58-91-CE-25-6B-B8-3E-64-FF-74-16-95-9F-23-1A-98-49-4C-78-BE-FF-A9-32-F2-E8-A3-0D-46-DA-B7-E4-DA-C2-E8-0B-A0-2E-1D-97-EE-FF-96-8A-6C-67-6C-AC-CF-C8-7D-BD" }

Must return the given string mashified and then encrypted with the given AES key, as a hexadecimal string
e.g. "0E-94-10-D7-44-A1-1C-D6-86-21-A2-47-0A-4A-AA-73"

If an error occurs, the result should be `"Bad Request"` with a status code of BAD REQUEST (400)

The client must then be able to decrypt the string using its symmetric (AES) key and IV, and finally output the new string to the console.

Input String from Console Window	Example	Request	Response	Client Function	Output to Console Window

Protected Mashify <string>	Protected Mashify Hello World!	Client must get the locally stored ApiKey. If it doesn't yet exist the console must print "You need to do a User Post or User Set first". Client must then generate and store a new AES key and IV. It must now encrypt <string>, the AES key and the AES IV, using the public key, which must have already been requested from the server. Remember to encrypt using the OaepSHA1 padding mode.
		
		"Client doesn't yet have the public key" must be written to the console if the client doesn't know the server public key yet.
		
		localhost:<portnumber>/api/protected/mashify
		
		with ApiKey:<apikey> in the header and with the JSON object:
		{
		"EncryptedString":"<encryptedstring>",
		"Encryptedsymkey":"<encryptedAESkey>",
		"EncryptedIV":"<encryptedAESiv>"
		}
		in the request body
			string	Decrypt the returned hex using the AES key and IV that was previously generated	Decrypted string which should be "!dlrXW XllXH" in this example.
					
					"An error occurred!" if the returned hex is not a valid string

TEST YOUR CLIENT AGAINST THE TEST SERVER – IF THIS ELEMENT DOES NOT FUNCTION WHEN TESTED AGAINTST THE TEST SERVER YOU WILL NOT RECEIVE MARKS FOR YOUR CLIENT *OR* SERVER IMPLEMENTATION!

Task15 (Report) – 10%

You will need to write and submit up to 1,500 words in a report addressing these points:

1. Outline what **your API** is and does, how it manages requests and discuss why this API is stateless. Describe the difference behaviours **your server** would exhibit if it had been stateful.
2. Briefly explain how **you** have implemented Route Mapping and specify how else you could have done it using the ASP.NET WebAPI framework.
3. Briefly outline the different RESTful request methods **you have used** are and provide screenshots of where you have used these requests in your server project to illustrate your written work.
4. Briefly describe how **your Server and Client** use the API key. Ensure you discuss the Authentication, Authorization and middleware in **your** server. Identify if you think an API key is a good or bad option for identifying users, giving your reasons. Is the API key safe for **your solution**? How would you ensure this API key was kept safe if you were developing this Server/Client in the 'real world'?
5. Briefly describe how **you** have used Entity Framework and how **your** code is loosely coupled.
6. Finally, write a short reflective statement about:
    a. which tasks you completed and to what level,
    b. any problems you had with any of the implementation/functionality, and
    c. how you overcame these problems (if you managed to).

NOTE:

You must ensure that your report is written by you, is free of plagiarism and that any text you paraphrase or quotes you use are appropriately referenced using the University of Hull Harvard referencing style. More information on this and advice on writing reports is available here: https://libguides.hull.ac.uk/skillsguides

You may like to use bulletpoints to be concise in your report and save words.

Submission:

This coursework is to be submitted via Canvas. To submit your coursework:

1. Clean your solution by right-clicking on the solution in Solution Explorer (Visual Studio) and choosing Clean.
2. You must place your Visual Studio solution files in a single folder. Ensure you submit the entire solution.
3. Add an electronic copy (preferably PDF) of your report into the folder
4. ZIP the folder (not 7Zip, RAR, etc.). This ZIP file must then be submitted via Canvas.

E.g.