Practical 4: Server Authentication

Learning Outcomes

- Prepare user model and controller
- Add post method for user register
- Add post method for user login
- Generate Json Web Token (JWT)

Activity 1: Prepare user model and controller

1. Use Visual Studio to open solution **LearningAPI**.

2. Under folder Models, add new class **User** with the code below:

```
namespace LearningAPI.Models
{
    public class User
    {
        public int Id { get; set; }

        public string Name { get; set; } = string.Empty;

        public string Email { get; set; } = string.Empty;

        public string Password { get; set; } = string.Empty;

        public DateTime CreatedAt { get; set; }

        public DateTime UpdatedAt { get; set; }
    }
}
```

Name, email and password are required for user register.

Email and password are required for user login.

The property Password is used to store hashed password in database.

3. Add data annotations for string and date time properties:

```
using Microsoft.EntityFrameworkCore.Metadata.Internal;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace LearningAPI.Models
{
    public class User
    {
        public int Id { get; set; }

        [MaxLength(50)]
        public string Name { get; set; } = string.Empty;

        [MaxLength(50)]
        public string Email { get; set; } = string.Empty;

        [MaxLength(100)]
        public string Password { get; set; } = string.Empty;

        [Column(TypeName = "datetime")]
        public DateTime CreatedAt { get; set; }

        [Column(TypeName = "datetime")]
        public DateTime UpdatedAt { get; set; }
    }
}
```

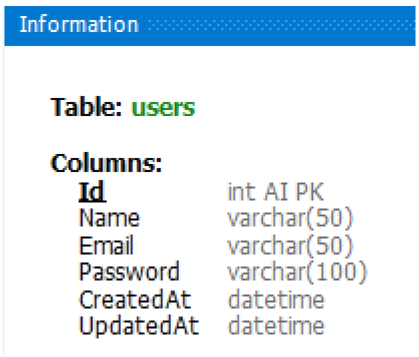
The data annotations are used to determine the database column data types.

4. In class MyDbContext, add new DbSet for User:

```
        public required DbSet<Tutorial> Tutorials { get; set; }

        public required DbSet<User> Users { get; set; }
```

5. In package manager console, run commands:
   - **Add-Migration AddUser**
   - **Update-Database**

6. Open MySQL Workbench. Observe that the table "**users**" is created.



```
Table: users

Columns:
  Id          int AI PK
  Name        varchar(50)
  Email       varchar(50)
  Password    varchar(100)
  CreatedAt   datetime
  UpdatedAt   datetime
```

7. Under folder Controllers, add new class **UserController** with the code below:

```
using LearningAPI.Models;
using Microsoft.AspNetCore.Mvc;

namespace LearningAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class UserController(MyDbContext context) : ControllerBase
    {
        private readonly MyDbContext _context = context;
    }
}
```

The DB context is added and will be used in API methods later.

Activity 2: Add post method for user register

1. Right click on project, select Manage NuGet Packages.

Install new package **BCrypt.Net-Next (latest version or v4.0.3)**

2. Under folder Models, add class **RegisterRequest** with code below:

```
namespace LearningAPI.Models
{
    public class RegisterRequest
    {
        public string Name { get; set; } = string.Empty;

        public string Email { get; set; } = string.Empty;

        public string Password { get; set; } = string.Empty;
    }
}
```

The properties are mapped to the data in the request body.

3. Add data annotations for the properties, which are used for input validation.

```
using System.ComponentModel.DataAnnotations;

namespace LearningAPI.Models
{
    public class RegisterRequest
    {
        [Required, MinLength(3), MaxLength(50)]
        public string Name { get; set; } = string.Empty;

        [Required, EmailAddress, MaxLength(50)]
        public string Email { get; set; } = string.Empty;

        [Required, MinLength(8), MaxLength(50)]
        public string Password { get; set; } = string.Empty;
    }
}
```

4. Add regular expression annotations for Name and Password:

```
        [Required, MinLength(3), MaxLength(50)]
        // Regular expression to enforce name format
        [RegularExpression(@"^[a-zA-Z '-,.]+$",
            ErrorMessage = "Only allow letters, spaces and characters: ' - , .")]
        public string Name { get; set; } = string.Empty;
```

```
        [Required, MinLength(8), MaxLength(50)]
        // Regular expression to enforce password complexity
        [RegularExpression(@"^(?=.*[a-zA-Z])(?=.*[0-9]).{8,}$",
            ErrorMessage = "At least 1 letter and 1 number")]
        public string Password { get; set; } = string.Empty;
```

@"xxx" is a verbatim string literal. It allows you to create string without escaping special characters such as backslashes.

5. In class UserController, add new post method **Register**:

```
        [HttpPost("register")]
        public IActionResult Register(RegisterRequest request)
        {
            // Create user object
            var now = DateTime.Now;
            string passwordHash = BCrypt.Net.BCrypt.HashPassword(request.Password);
            var user = new User()
            {
                Name = request.Name,
                Email = request.Email,
                Password = passwordHash,
                CreatedAt = now,
                UpdatedAt = now
            };

            // Add user
            _context.Users.Add(user);
            _context.SaveChanges();
            return Ok();
        }
```

The URL **path** for the HTTP Post method is “register”.

The package **Bcrypt.Net** is used to hash passwords.

The DB **context** is used to insert data into the database table.

6. Add code to trim string values:

```
        public IActionResult Register(RegisterRequest request)
        {
            // Trim string values
            request.Name = request.Name.Trim();
            request.Email = request.Email.Trim().ToLower();
            request.Password = request.Password.Trim();

            …
        }
```

The input string values are trimmed. Email is converted to lowercase.

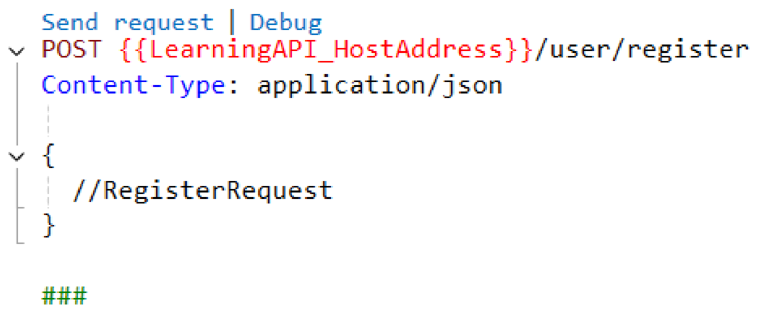
7. Add code to check email after trimming string values:

```
            // Check email
            var foundUser = _context.Users.Where(
                    x => x.Email == request.Email).FirstOrDefault();
            if (foundUser != null)
            {
                string message = "Email already exists.";
                return BadRequest(new { message });
            }
```

User email must be unique. If an existing user with the email is found, return bad request with the error message.

8. Click menu "View > Other Windows > **Endpoints Explorer**" to show the window. Generate request for the endpoint user register.

9. In the file **LearningAPI.http**, observe that the new endpoint is added.

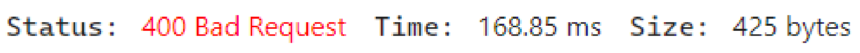


```
Send request | Debug
POST {{LearningAPI_HostAddress}}/user/register
Content-Type: application/json

{
  //RegisterRequest
}

###
```

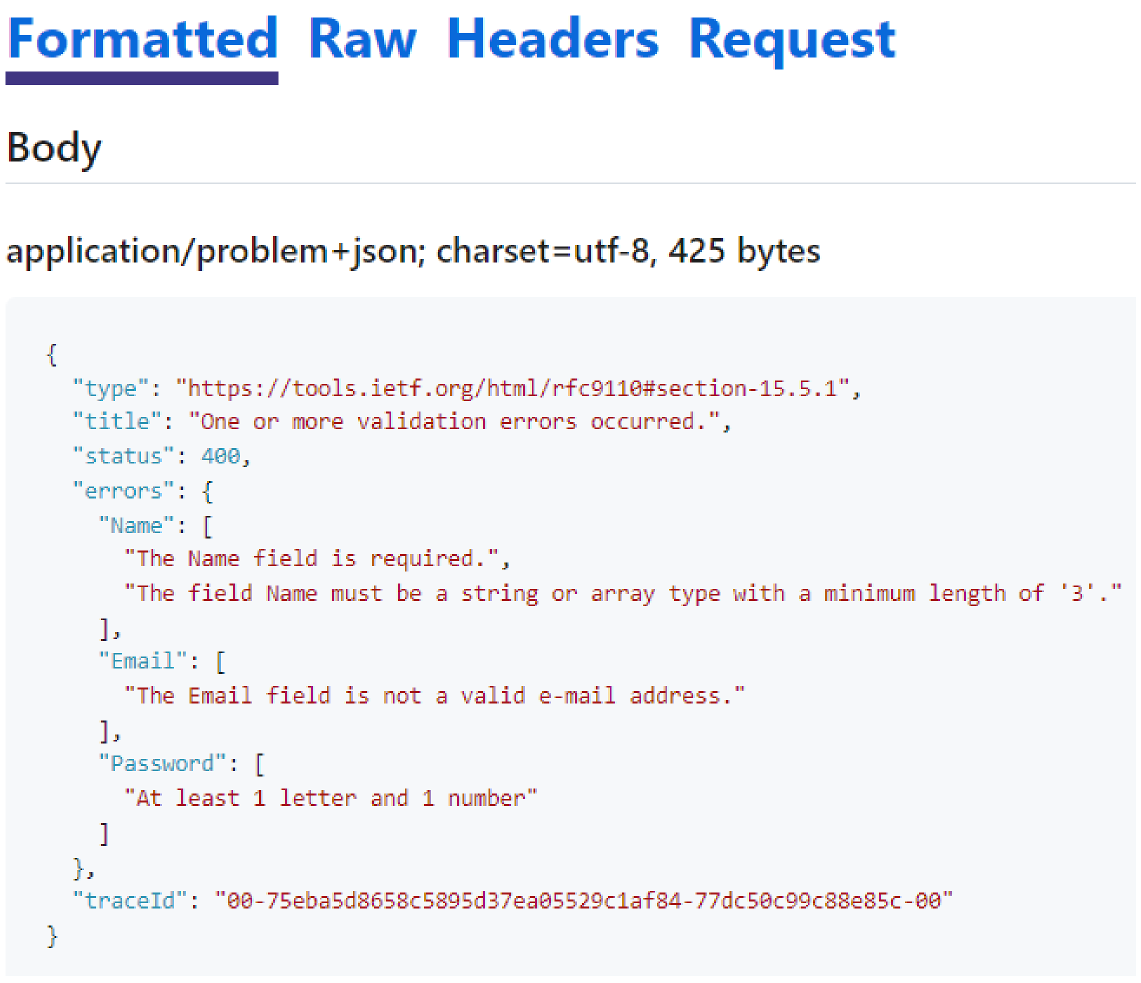
10. Edit the request body to use invalid input:

```
{
    "email": "ryan",
    "password": "password"
}
```

11. Test the endpoint with invalid input. The response status is 400 Bad Request. The validation errors are returned in the response body.



Status: 400 Bad Request Time: 168.85 ms Size: 425 bytes



Body

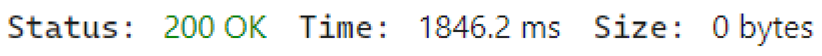
application/problem+json; charset=utf-8, 425 bytes

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Name": [
      "The Name field is required.",
      "The field Name must be a string or array type with a minimum length of '3'."
    ],
    "Email": [
      "The Email field is not a valid e-mail address."
    ],
    "Password": [
      "At least 1 letter and 1 number"
    ]
  },
  "traceId": "00-75eba5d8658c5895d37ea05529c1af84-77dc50c99c88e85c-00"
}
```

12. Correct all the validation errors in the request body and test again.

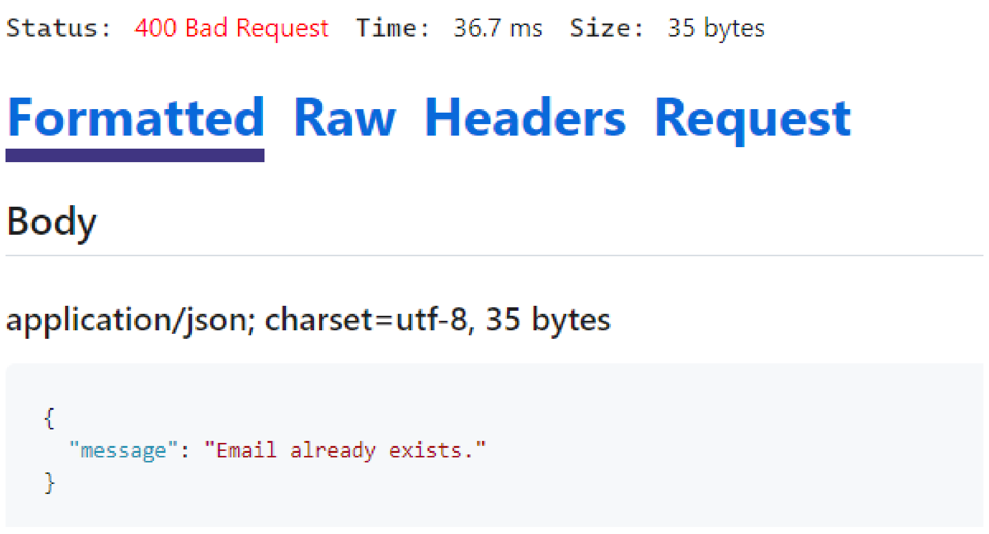
```
{
    "name": "Ryan Tan",
    "email": "ryantan@gmail.com",
    "password": "password123"
}
```

The response status is 200 OK. The user was registered successfully.



Status: 200 OK Time: 1846.2 ms Size: 0 bytes

13. Send the same request again to test unique email. The response status is 400 Bad Request. The error message is returned in the response body.



Status: 400 Bad Request Time: 36.7 ms Size: 35 bytes

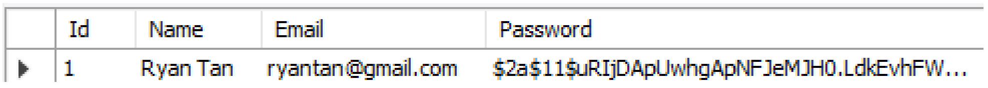
Formatted Raw Headers Request

Body

application/json; charset=utf-8, 35 bytes

```
{
  "message": "Email already exists."
}
```

14. Open MySQL Workbench. Observe that the new user is inserted to the users table and the password is hashed.



	Id	Name	Email	Password
▶	1	Ryan Tan	ryantan@gmail.com	$2a$11$uRIjDApUwhgApNFJeMJH0.LdkEvhFW...

Activity 3: Add post method for user login

1. Under folder Models, add class **LoginRequest** with code below:

```
using System.ComponentModel.DataAnnotations;

namespace LearningAPI.Models
{
    public class LoginRequest
    {
        [Required, EmailAddress, MaxLength(50)]
        public string Email { get; set; } = string.Empty;

        [Required, MinLength(8), MaxLength(50)]
        public string Password { get; set; } = string.Empty;
    }
}
```

The properties are mapped to the data in the request body.
The data annotations are used for input validation.

2. In class UserController, add new post method **Login**:

```
        [HttpPost("login")]
        public IActionResult Login(LoginRequest request)
        {
            // Trim string values
            request.Email = request.Email.Trim().ToLower();
            request.Password = request.Password.Trim();

            // Check email and password
            string message = "Email or password is not correct.";
            var foundUser = _context.Users.Where(
                   x => x.Email == request.Email).FirstOrDefault();
            if (foundUser == null)
            {
                return BadRequest(new { message });
            }
            bool verified = BCrypt.Net.BCrypt.Verify(
                   request.Password, foundUser.Password);
            if (!verified)
            {
                return BadRequest(new { message });
            }

            // Return user info
            var user = new
            {
                foundUser.Id,
                foundUser.Email,
                foundUser.Name
            };
            return Ok(new { user });
        }
```

The URL **path** for the HTTP Post method is “login”.

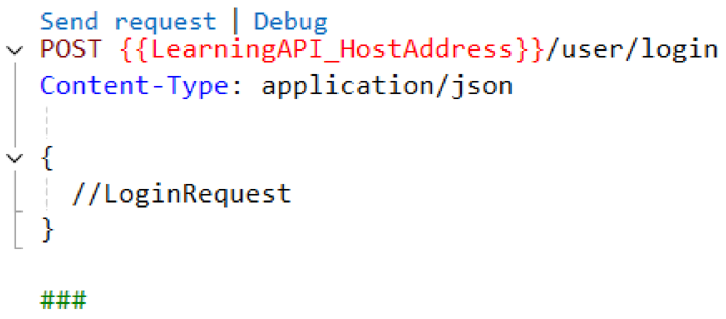
The package **Bcrypt.Net** is used to compare the input plain password with the hashed password stored in database.

If the email or password is not correct, return bad request with the error message. The message is generic for incorrect email or password, following security best practice.

If both email and password are correct, return the basic user info.

3. Generate request for the endpoint user login.

4. In the file **LearningAPI.http**, observe that the new endpoint is added.

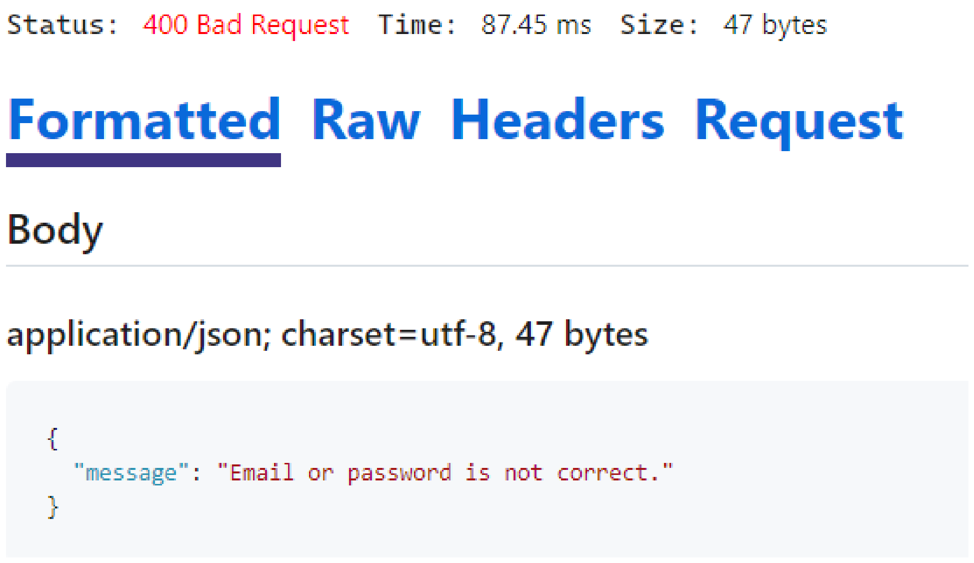


```
Send request | Debug
POST {{LearningAPI_HostAddress}}/user/login
Content-Type: application/json

{
  //LoginRequest
}

###
```

5. Edit the request body to use incorrect email:

```
{
    "email": "test@gmail.com",
    "password": "password123"
}
```

6. Test the endpoint user login. The response status is 400 Bad Request. The error message is returned in the response body.



```
Status: 400 Bad Request  Time: 87.45 ms  Size: 47 bytes
```

Body

application/json; charset=utf-8, 47 bytes

```
{
  "message": "Email or password is not correct."
}
```

7. Test the endpoint user login with incorrect password:

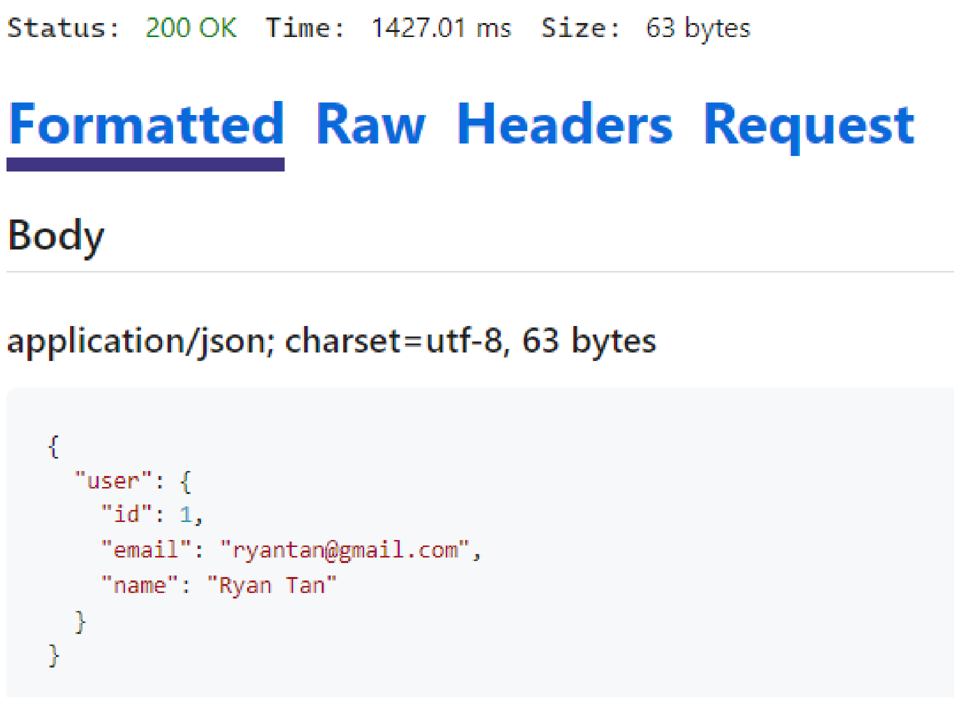
```
{
    "email": "ryantan@gmail.com",
    "password": "password321"
}
```

The response status is 400 Bad Request. The same error message is returned in the response body.

8. Test user login with correct email and password:

```
{
    "email": " ryantan@gmail.com",
    "password": "password123"
}
```

The response status is 200 OK. The user object is returned in the response body.



Status: 200 OK Time: 1427.01 ms Size: 63 bytes

Body

application/json; charset=utf-8, 63 bytes

```
{
  "user": {
    "id": 1,
    "email": "ryantan@gmail.com",
    "name": "Ryan Tan"
  }
}
```

Activity 4: Generate Json Web Token (JWT)

1. Right click on project, select Manage NuGet Packages.

Install new package **Microsoft.AspNetCore.Authentication.JwtBearer** (v8.0.x)

Json Web Token (JWT) is often used in authentication scenarios, where a user logs in to a web application, and upon successful authentication, the server generates a JWT and sends it back to the client.

The client can then include the JWT in subsequent requests to the server to access protected resources without the need to send credentials (e.g., email and password) with every request.

The server can validate the JWT to ensure the user's authenticity and authorization to access the requested resources.

2. In file appsettings.Development.json, add new section for authentication:

```
  "ConnectionStrings": {
    "MyConnection": "…"
  },
  "Authentication": {
    "Secret": "your_app_secret",
    "TokenExpiresDays": 30
  }
```

You can use the online tool to generate your app secret.
https://www.uuidtools.com/v4 Get a random UUID.

Token expires in 30 days.

3. In class UserController, add private attribute **_configuration**:

```
    public class UserController(MyDbContext context, IConfiguration configuration) :
ControllerBase
    {
        private readonly MyDbContext _context = context;
        private readonly IConfiguration _configuration = configuration;
```

The attribute **_configuration** is used to read values in appsettings.

4. Add private method **CreateToken**:

```
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
…

        private string CreateToken(User user)
        {
            string? secret = _configuration.GetValue<string>(
"Authentication:Secret");
            if (string.IsNullOrEmpty(secret))
            {
                throw new Exception("Secret is required for JWT authentication.");
            }
            int tokenExpiresDays = _configuration.GetValue<int>(
"Authentication:TokenExpiresDays");

            var tokenHandler = new JwtSecurityTokenHandler();
            var key = Encoding.ASCII.GetBytes(secret);

            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(
                [
                    new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                    new Claim(ClaimTypes.Name, user.Name),
                    new Claim(ClaimTypes.Email, user.Email)
                ]),
                Expires = DateTime.UtcNow.AddDays(tokenExpiresDays),
                SigningCredentials = new SigningCredentials(
new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
            };
            var securityToken = tokenHandler.CreateToken(tokenDescriptor);
            string token = tokenHandler.WriteToken(securityToken);

            return token;

        }
```

The payload in the access token contains the claims, which are statements about the user.

We put user info including id, name and email in the claims.

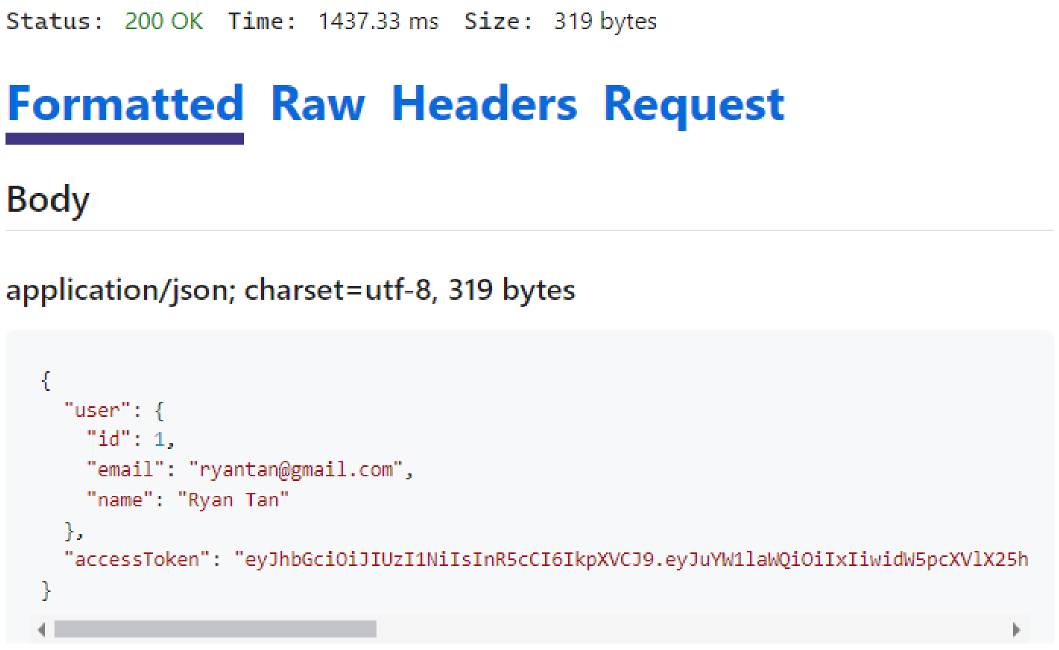
5. Edit method **Login** to return access token:

```
            // Return user info
            var user = new
            {
                foundUser.Id,
                foundUser.Email,
                foundUser.Name
            };
            string accessToken = CreateToken(foundUser);
            return Ok(new { user, accessToken });
```

6. Test user login with correct email and password:

```
{
    "email": "ryantan@gmail.com",
    "password": "password123"
}
```

The Json Web Token is returned together with the user info now.



Status: 200 OK Time: 1437.33 ms Size: 319 bytes

Formatted Raw Headers Request

Body

application/json; charset=utf-8, 319 bytes

```
{
  "user": {
    "id": 1,
    "email": "ryantan@gmail.com",
    "name": "Ryan Tan"
  },
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiIxIiwidW5pcXVlX25h
}
```

We will learn how to verify the token in next practical.