### ****1. Demonstrate creation of an Action method to return list of custom class entity****

To demonstrate the creation of an action method that returns a list of a custom class entity, we first created a model class named Employee that included properties such as Id, Name, Salary, Permanent, DateOfBirth, as well as complex types like Department and a list of Skill objects. These supporting classes (Department and Skill) were also created as part of the Models folder. In the EmployeeController, a private method called GetStandardEmployeeList() was used to generate a predefined list of Employee objects with realistic data. An action method GetStandard() decorated with [HttpGet] and [AllowAnonymous] was then implemented to return this list. This method also included [ProducesResponseType(200)] and [ProducesResponseType(500)] attributes to describe possible responses for documentation via Swagger. This demonstrated how to build and expose a fully functional custom object list via a Web API endpoint.

### ****2. Explain the usage of FromBody attribute****

The [FromBody] attribute is used in Web API to bind complex types from the **request body** rather than from the query string or URL parameters. In this project, [FromBody] was applied to the POST method inside the EmployeeController to accept an Employee object directly from the JSON body of the HTTP request. This allows clients (like Swagger, Postman, or frontend apps) to send full JSON objects containing nested data structures (like a Department object and a list of Skill objects) in the body of the request. The Web API model binder deserializes this JSON content into the Employee object automatically, thanks to the [FromBody] attribute. This clearly demonstrates how ASP.NET Core enables receiving rich data structures from HTTP requests in a clean and readable manner.

### ****3. Demonstrate Custom filter****

To demonstrate a custom filter, a class named CustomAuthFilter was created that inherited from ActionFilterAttribute. Within this class, the OnActionExecuting method was overridden to inspect incoming HTTP requests before they reached the controller action. The filter checked whether the request contained an Authorization header, and if so, whether its value included the word "Bearer". If the header was missing or improperly formatted, the filter returned a BadRequestObjectResult with appropriate error messages. Additionally, a custom exception filter named CustomExceptionFilter was implemented by using the IExceptionFilter interface. The OnException method was used to catch unhandled exceptions globally. Any exception thrown in a controller action (such as from the GetStandard() method) was logged into a local errors.txt file using File.AppendAllText, and a 500 Internal Server Error response was returned. Although the documentation mentioned using Microsoft.AspNetCore.Mvc.WebApiCompatShim, it was not required in this scenario because the built-in filter interfaces in ASP.NET Core 8.0 were sufficient. Together, these custom filters demonstrated how to build powerful and reusable components to handle authorization and error logging in Web APIs.