CST-350 Activity 7 REST API and Right Click Events

Introduction

In this lesson, you will create RESTful services using ASP.NET Core and implement advanced user interactions in a web application. RESTful services are a key part of web development, forming the bridge different parts of an application and between different applications altogether. By the end of this lesson, you will be equipped with the skills to build, test, and use REST APIs, as well as handle complex user interactions like right-click events in a button grid.

Learning Goals:

Understand RESTful Services: Learn the fundamental principles of REST (Representational State Transfer) and how to implement these principles using ASP.NET Core.

1. **Create and Test REST Controllers**: Develop REST controllers in ASP.NET Core and test them using Postman.
2. **Differentiate between MVC and REST controllers**: Understand the key differences between traditional MVC (Model-View-Controller) applications and RESTful APIs.
3. **Implement CRUD Operations**: Extend an existing application to support Create, Read, Update, and Delete (CRUD) operations through a REST API.
4. **Handle Right-Click Events**: Enhance the Button Grid application to respond to right-click events using JavaScript.

Part 1 REST Services

Goal:

In this activity, you will learn how to build a REST(ful) Service in ASP.NET

You will need the code from a previous activity and you will use your Browser and **Postman** to test your REST(ful) Services.

Create a Rest Controller:

1. Open the **Products application** that was created in a previous lesson. We will extend the application to use the same data in a REST API format.

2. Duplicate (copy and paste) the **ProductsController**.
3. Rename the new controller to **ProductsRestController**.

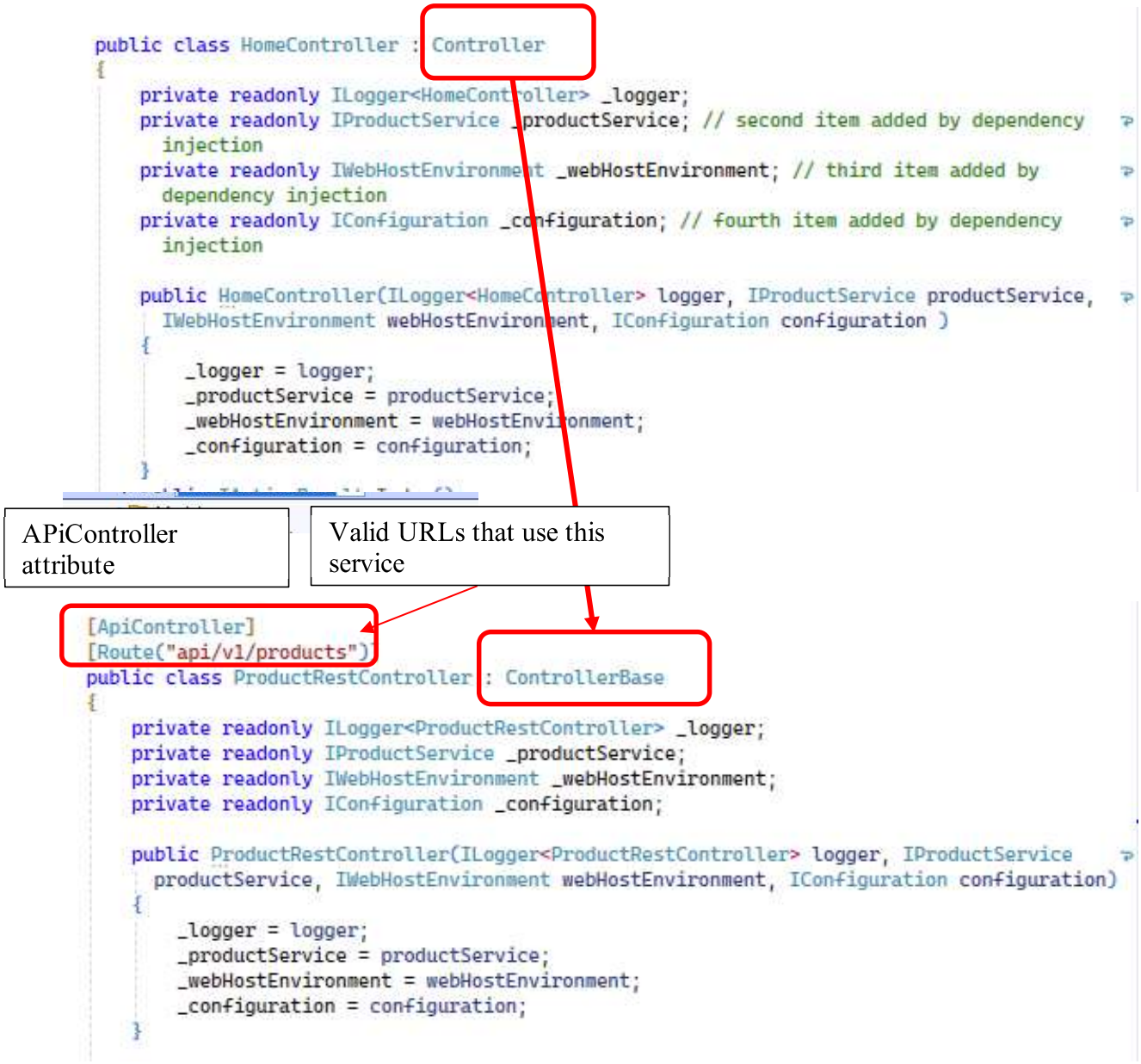


Figure 1 Comparison between an MVC controller and a RESTful API controller

4. Modify the ShowAllProducts method

ShowAllProducts from the HomeController

```
// async method to get all products from the database
public async Task<IActionResult> ShowAllProducts()
{
    List<ProductViewModel> products = _productService.GetAllProducts().Result.ToList();
    return View(products);
}
```

Modified method in the ProductRestController

```
[HttpGet]
public async Task<ActionResult<IEnumerable<ProductViewModel>>> ShowAllProducts()
{
    // get all products.
    IEnumerable<ProductViewModel> products = await _productService.GetAllProducts();

    // OK status code is 200
     return Ok(products);

}
```

Figure 2 Comparison between similar functions in an MVC controller and a REST controller.

Contrasting the MVC method and the Rest Method

Controller Type and Attribute:

MVC Controller Method: It does not have an HTTP method attribute (like [HttpGet]) because it is implicitly handled by the MVC framework based on the action method name and routing configuration.

REST Controller Method: This method is part of a REST controller and explicitly marked with the [HttpGet] attribute, indicating that it handles HTTP GET requests.

Return Type:

MVC Controller Method: The return type is Task<IActionResult>. This type is used for methods that return a view in an MVC application. IActionResult is a base interface for different types of responses, including views.

REST Controller Method: The return type is Task<ActionResult<IEnumerable<ProductViewModel>>>. ActionResult<T> is a more specific type used in Web API controllers. It can represent various HTTP status codes and response types, making it suitable for RESTful APIs.

Response Content:

MVC Controller Method: The method returns a view that is rendered on the server and sent to the client. The View(products) statement returns a ViewResult, which means the products data is passed to a Razor view for rendering HTML.

REST Controller Method: The method returns data directly in the response body. The Ok(products) statement returns an OkObjectResult, which means the products data is serialized to JSON (or another format, like XML, depending on configuration) and sent directly to the client.

HTTP Response Codes:

MVC Controller Method: The method implicitly returns an HTTP 200 OK status code when the view is rendered successfully. However, it does not have explicit control over the HTTP status code beyond the implicit behavior of the MVC framework.

REST Controller Method: The method explicitly returns an HTTP 200 OK status code by using the Ok(products) helper method. This explicitness allows for more precise control over HTTP response codes, which is a key aspect of RESTful API design.

Different Uses:

MVC Controller Method: This method is used in a web application where the server-side rendering of views is required. It is designed to work with Razor views and handle user interactions within the web application's UI.

REST Controller Method: This method is used in a web API, designed to provide data to clients that could be anything from web browsers to mobile apps to other servers. It follows RESTful principles, where endpoints return data in a structured format (like JSON) and allow clients to manipulate resources.

5. Run the application and test the api endpoint

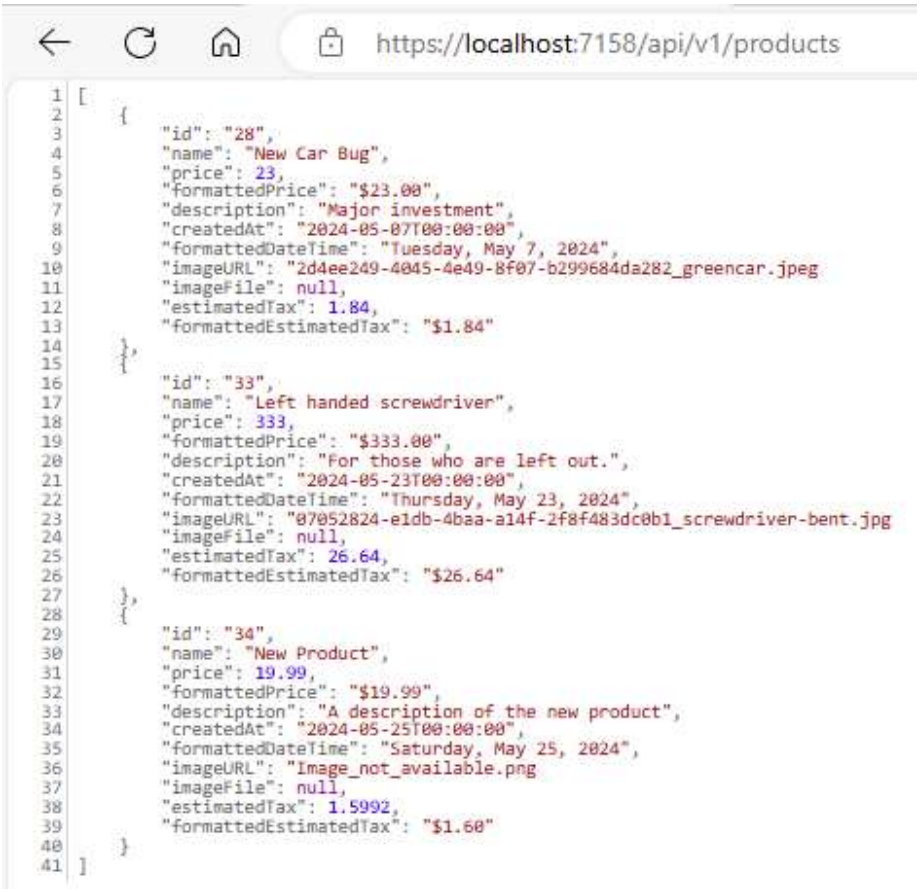


Figure 3 Output for the URL that fetches all products. The output is in formatted JSON text.

7. Right-click the browser to inspect the page.
8. View the Network tab of the developer tools to see the Status number (200) of a successful request.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

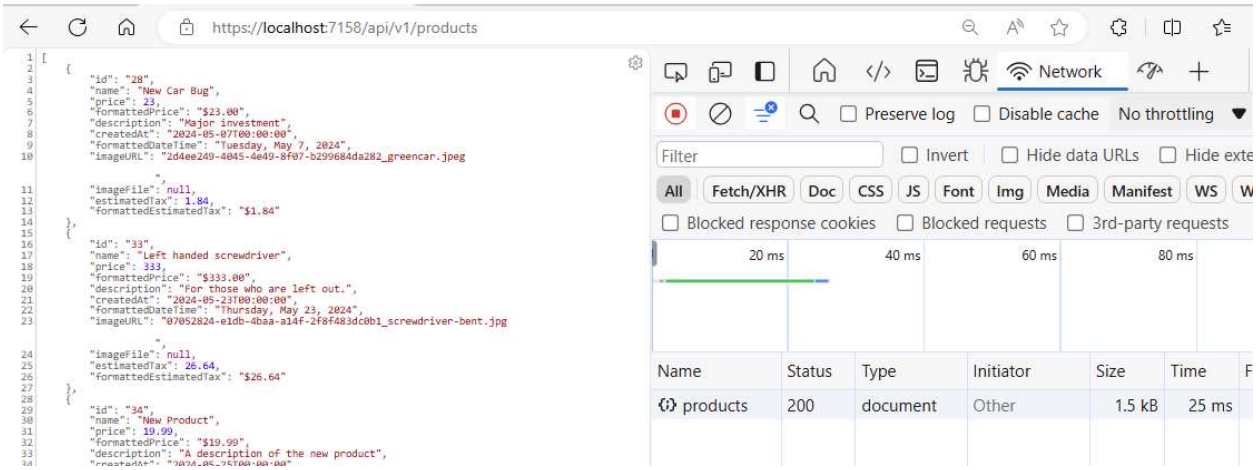


Figure 4 The developer tools (right) are showing the network tab.

About Request Response Numbers

In addition to the commonly known HTTP status codes 200 (OK) and 404 (Not Found), there are several other status codes that are frequently used in REST APIs. These status codes provide important information about the result of the HTTP request. It is possible to create a REST API whose response codes are simply 200. 400 or 500. However, more specific numbers are helpful.

Here are some common response codes:

Successful Responses

200 OK: The request was successful.

201 Created: The request was successful and a resource was created. This is often used for POST requests.

202 Accepted: The request has been accepted for processing, but the processing has not been completed. This is typically used for asynchronous operations.

204 No Content: The request was successful, but there is no content to send in the response. This is often used for DELETE operations.

Client Error Responses

These errors can result from user error since they involve the client request problems.

400 Bad Request: The server could not understand the request due to invalid syntax.

401 Unauthorized: The request requires user authentication. It indicates that the request has not been applied because it lacks valid authentication credentials.

403 Forbidden: The server understood the request, but it refuses to authorize it. This status is similar to 401, but re-authenticating will make no difference.

404 Not Found: The server cannot find the requested resource.

405 Method Not Allowed: The request method is known by the server but has been disabled and cannot be used.

Server Error Responses

The 500 errors almost always indicate that the programmer made a mistake in the coee and caused a run-time error.

500 Internal Server Error: The server encountered an unexpected condition that prevented it from fulfilling the request.

501 Not Implemented: The server does not support the functionality required to fulfill the request.

502 Bad Gateway: The server was acting as a gateway or proxy and received an invalid response from the upstream server.

503 Service Unavailable: The server is not ready to handle the request, usually because it is temporarily overloaded or down for maintenance.

504 Gateway Timeout: The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.

Redirection Message

301 Moved Permanently: The requested resource has been assigned a new permanent URI and any future references to this resource should use one of the returned URIs.

9. Update the GetProductById method to return a JSON-encoded product.

```
[HttpGet("{id}")]
public async Task<ActionResult<ProductViewModel>> GetProductById(int id)
{
    var product = await _productService.GetProductById(id);
    if (product == null)
    {
        return NotFound();
    }
    return Ok(product);
}
```

Figure 5 Get a single product using its id number

10. Test the program again but specifiy a valid id number in the URL.

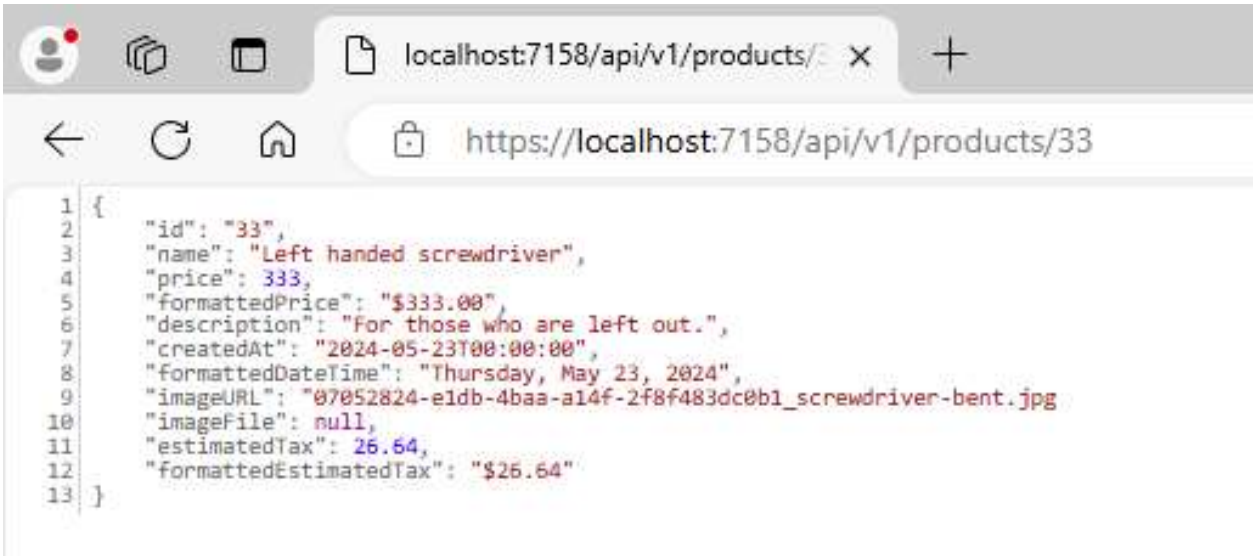


Figure 6 Fetching one product using the id 33

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

11. If you provide an Id number that does not match an existing product, you will receive an error.

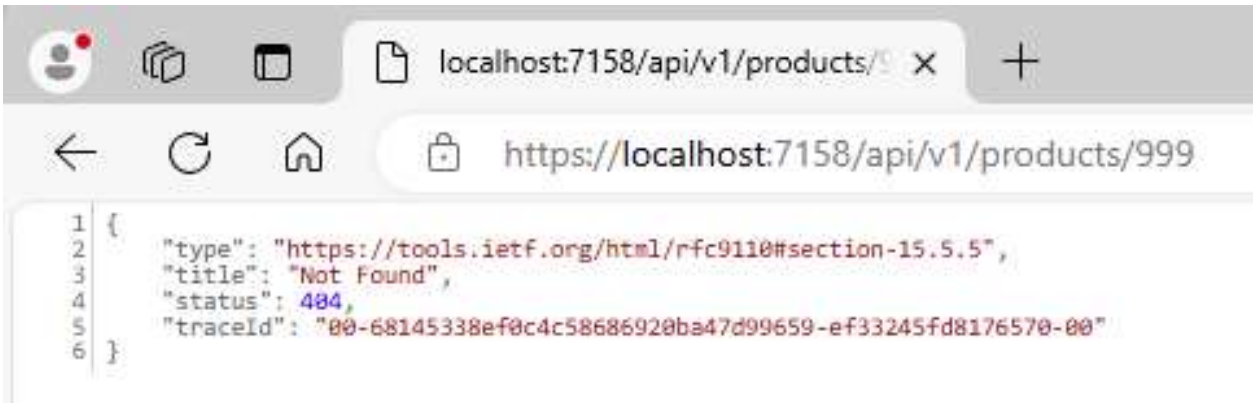


Figure 7 Attempted to fetch a product for an id 999 which does not exists in the database.

Explanation of Each Field in the 404 Error

1. **type**: **"https://tools.ietf.org/html/rfc9110#section-15.5.5"**

- Provides a URI that identifies the problem type. This URI typically points to a document that describes the problem in detail. Clients can use this URI to get more information about the error, potentially including how to avoid or fix it.

2. **title**: **"Not Found"**
    - A short, human-readable summary of the problem type.
3. **status**: **404**
    - Standard HTTP status code indicating the specific error, in this case, "404 Not Found," meaning the requested resource could not be found.
4. **traceId**: **"00-68145338ef0c4c58686920ba47d99659-ef33245fd8176570-00"**
    - **Purpose**: A unique identifier for this specific occurrence of the problem.
    - **Usage**: This trace ID is useful for debugging purposes. It allows developers to trace the request through logs and identify what went wrong in the server processing pipeline.

Search:

1. Update the Search method to return results from the database query.

```
// GET: api/v1/products/search?searchTerm=apple&inTitle=true&inDescription=true
[HttpGet("search")]
public async Task<ActionResult<IEnumerable<ProductViewModel>>> SearchForProducts([FromQuery] string searchTerm, [FromQuery]
  bool inTitle, [FromQuery] bool inDescription)
{
    // using the parameters in the URL, create an instance of the SearchFor class.
    SearchFor searchFor = new SearchFor
    {
        SearchTerm = searchTerm,
        InTitle = inTitle,
        InDescription = inDescription
    };

    var searchResults = await _productService.SearchForProducts(searchFor);

    if (searchResults == null || !searchResults.Any())
    {
        return NotFound("No products found matching the search criteria.");
    }

    return Ok(searchResults);
}
```

Figure 8 Search products method.

2. Run the program and test the search feature.

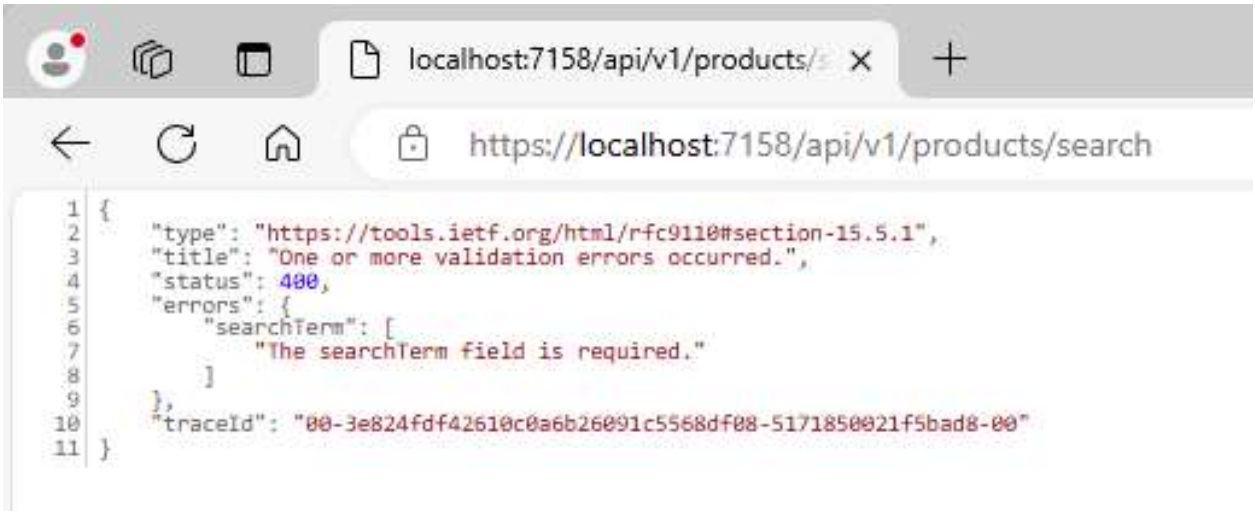


Figure 9 400 error indicates there is something wrong with the request. The user failed to provide the required searchTerm, inTitle or inDescription values.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

3. Be sure to include the SearchTerm, the inTitle and inDescription values as well.

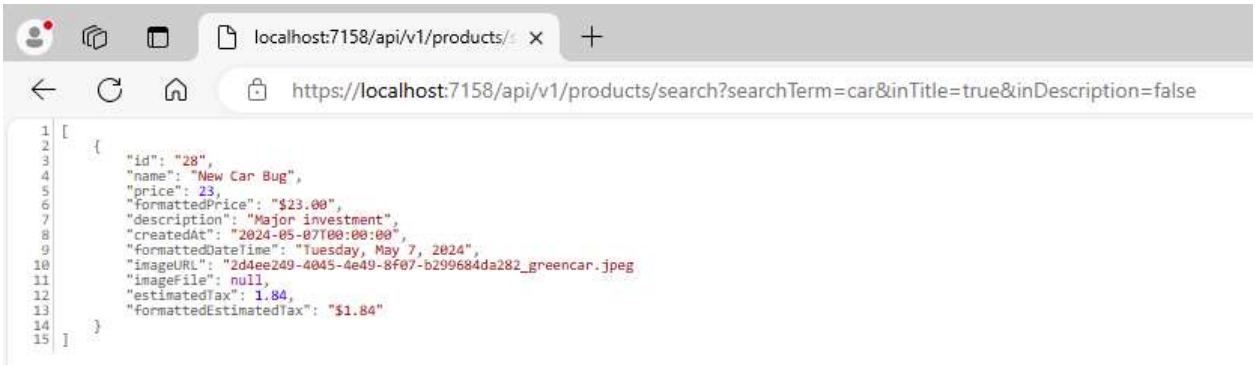


Figure 10 A successful request that includes SearchTerm, inTitle and inDescription values.

Postman Needed

The requests **ShowAllProducts**, **ShowProductById**, and **SearchForProducts** are all GET requests. GET requests can easily be sent by a web browser simply by typing the correct URL in

the browser bar. However, other types of requests, such as DELETE, POST, and PUT, will require a more sophisticated client.

Postman is a tool used by developers to test, document, and monitor APIs. It provides a user-friendly interface to send HTTP requests and view responses, making it easier to develop and debug APIs without relying on traditional HTML form requests.

Replacing Form Requests from HTML Applications

In traditional web development, testing endpoints typically requires setting up HTML forms and submitting them through a browser. This method can be cumbersome and time-consuming, especially for non-GET requests like POST, PUT, and DELETE.

Postman simplifies this process by allowing developers to:

1. **Send Requests Easily**: With Postman, you can configure and send any type of HTTP request (GET, POST, PUT, DELETE, etc.) without writing HTML forms.
2. **Customize Requests**: You can easily add headers, query parameters, and body data in various formats (JSON, XML, form-data, etc.).
3. **Save and Reuse Requests**: Postman allows you to save requests into collections, making it easy to reuse and organize them for different projects or testing scenarios.

Postman:

1. Start the Postman application.
2. Create a new collection named ProductApp in the Workspace. This will allow you to save related requests in a single folder.

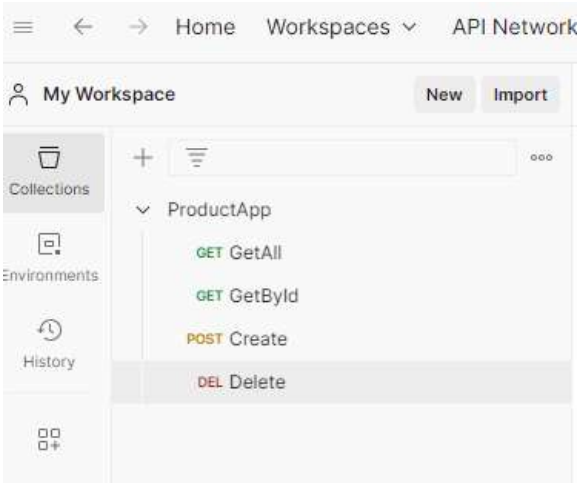


Figure 11 A collection of requests in Postman.

3. Create a new request to test the GetAll request. The response should match the JSON response you saw earlier in the web browser.

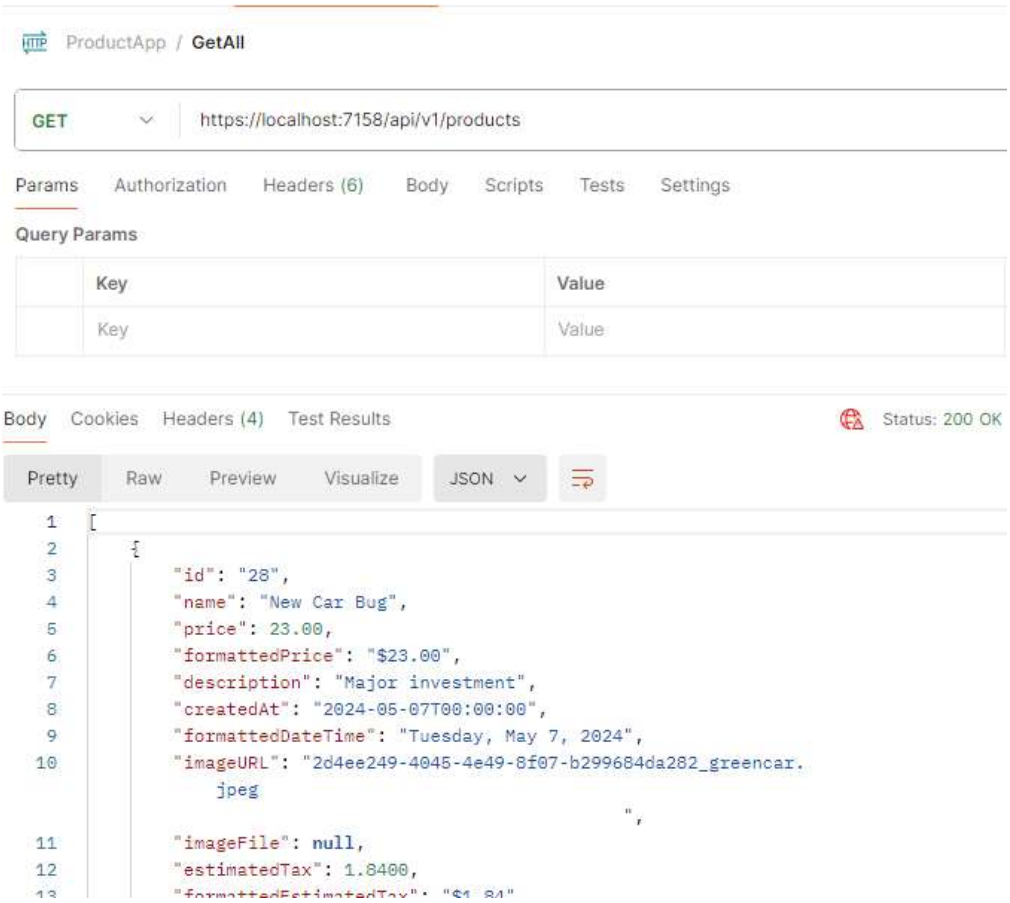


Figure 12 A get all products request made in Postman.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

4. Create another request to test GetProductById. Be sure to provide the id number of a valid product in the URL.

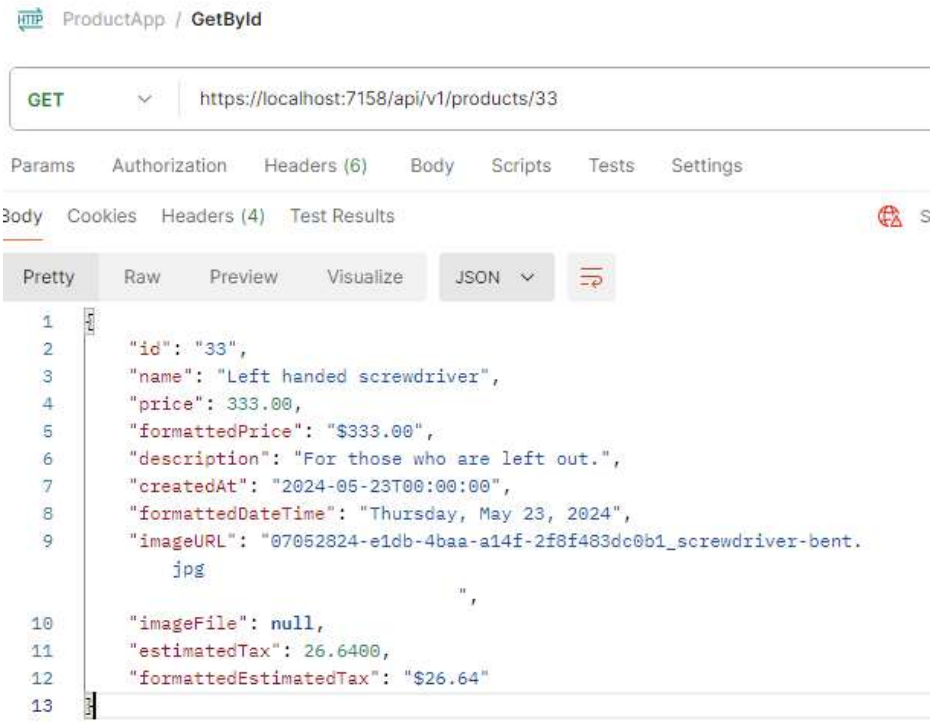


Figure 13 The result of requesting product 33

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

5. Perform a search request. This involves setting Query Params.

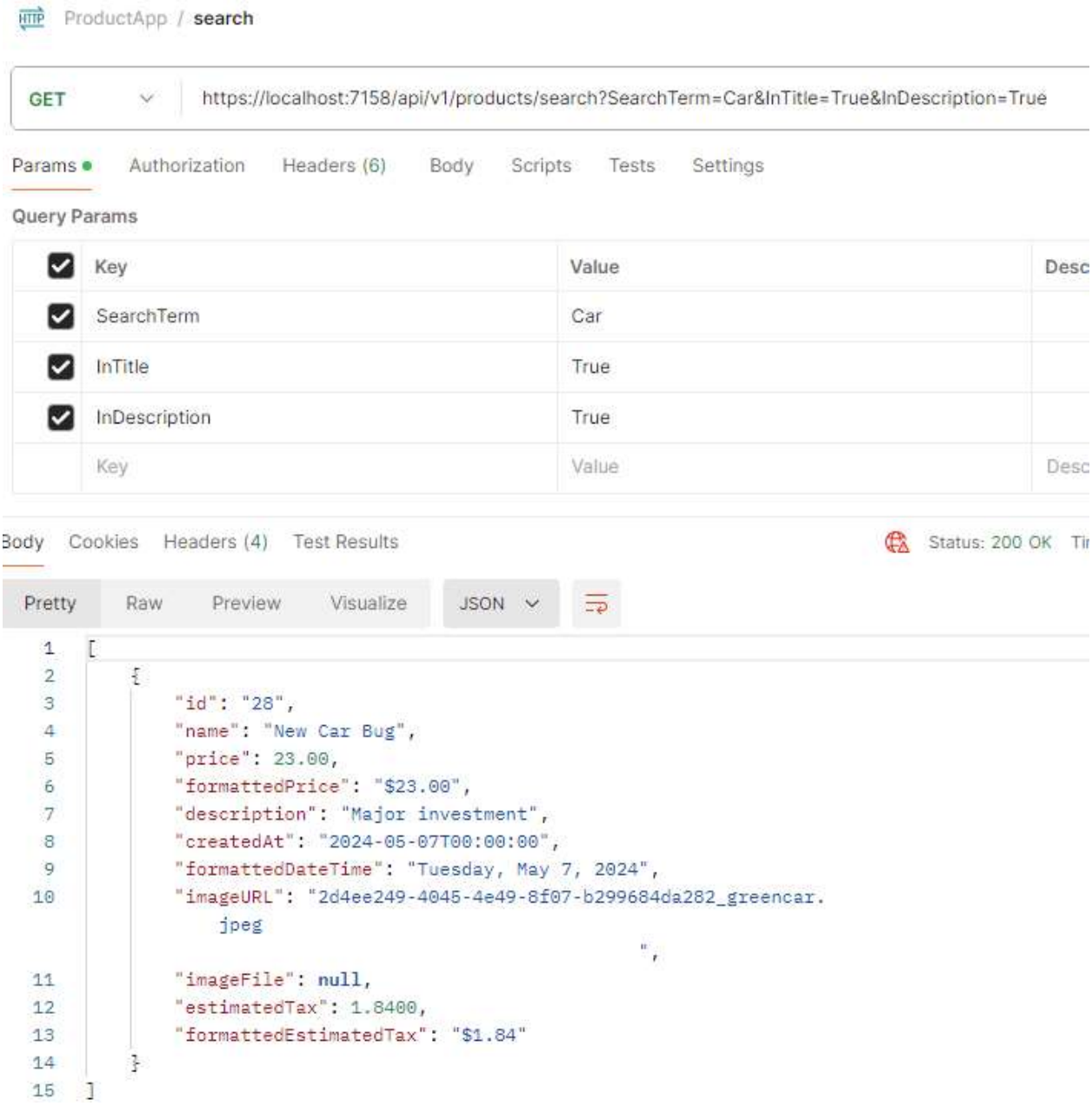


Figure 14 Searching requires three parameters to be supplied.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

6. Create a Create request. Notice the method type changes to POST. Form-data is supplied with key-value pairs that match the properties of the ProductViewModel.

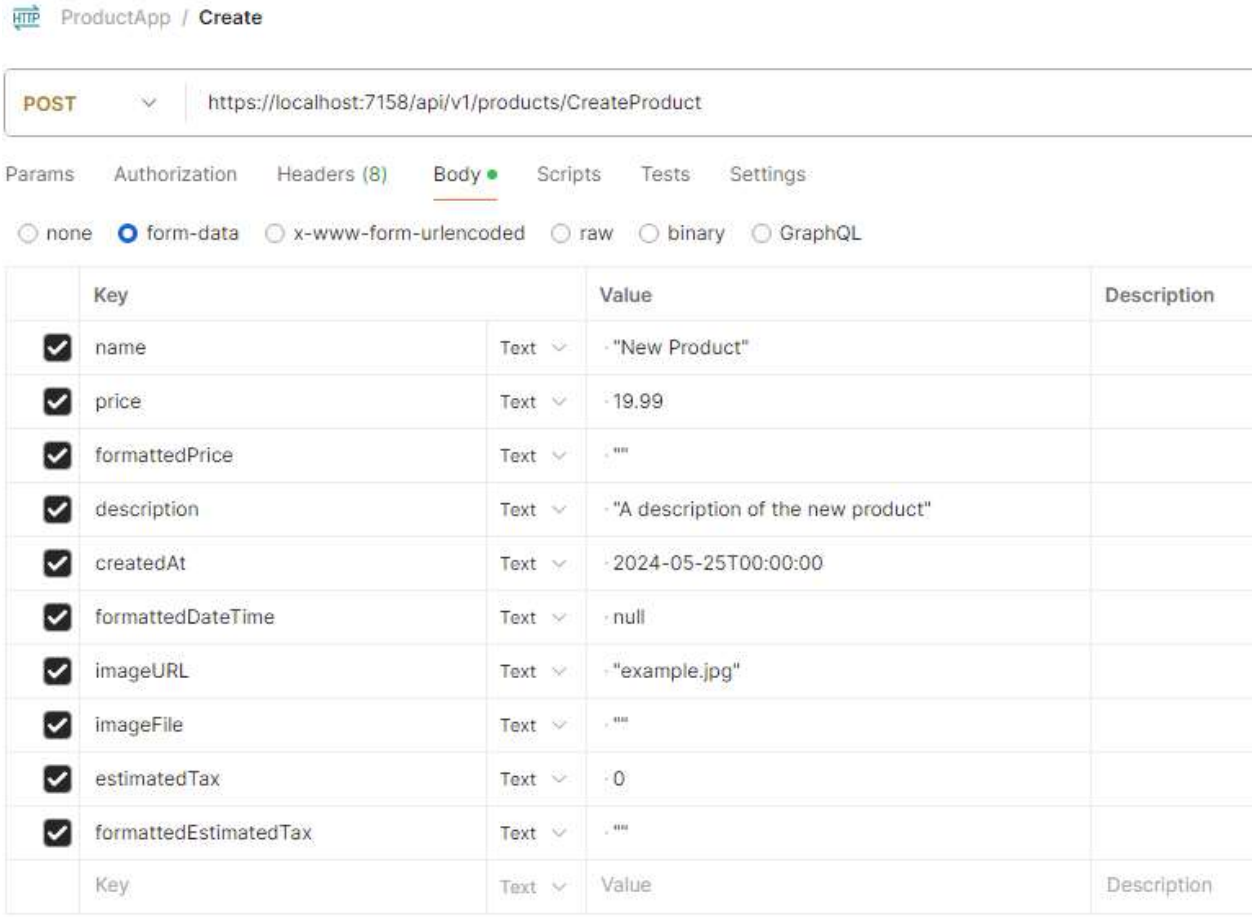


Figure 15 The create product form will send many values to the api.

7. The key-value pairs can also be edited as a list of strings with the “Bulk Edit” button.

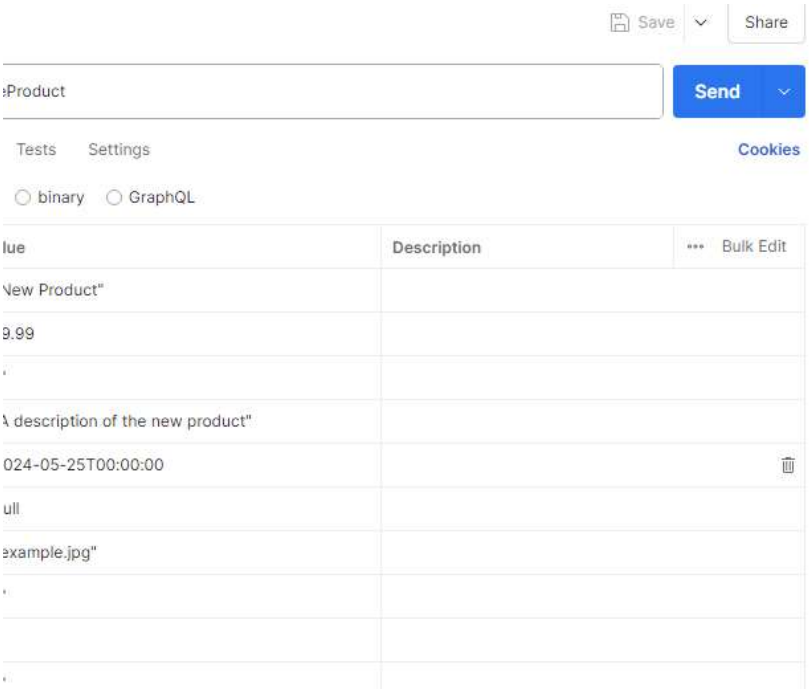


Figure 16 "Bulk Edit" is an option for adding many properties to a request.

8. Bulk edit allows you to type or paste a list of properties.

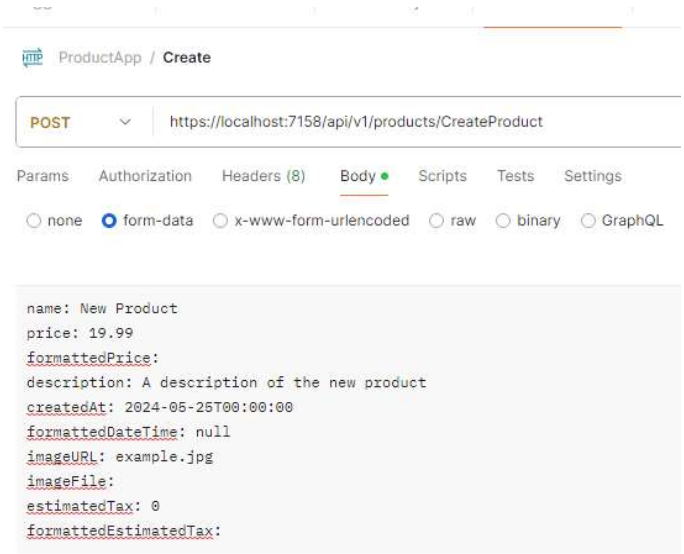


Figure 17 Form data being entered using the "Bulk Edit" option.

9. You should observer a 201 response type.

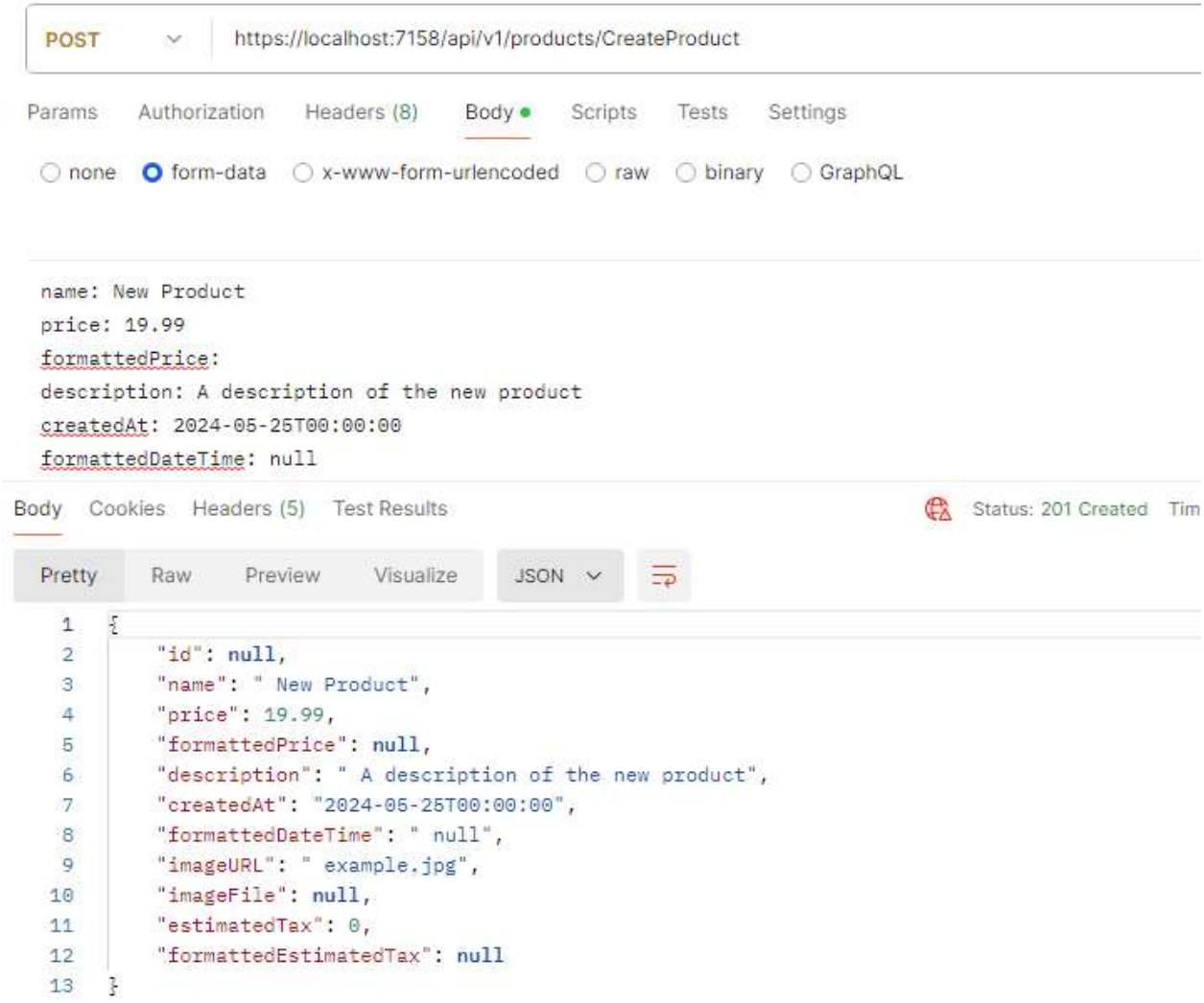


Figure 18 Result of a Create Product action. A 201 status message was returned from the server.

10. Verify the new product has been added by running the GetAll request again. You should see a new product appear in the list.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

11. Create a Delete request to remove the product you just created.

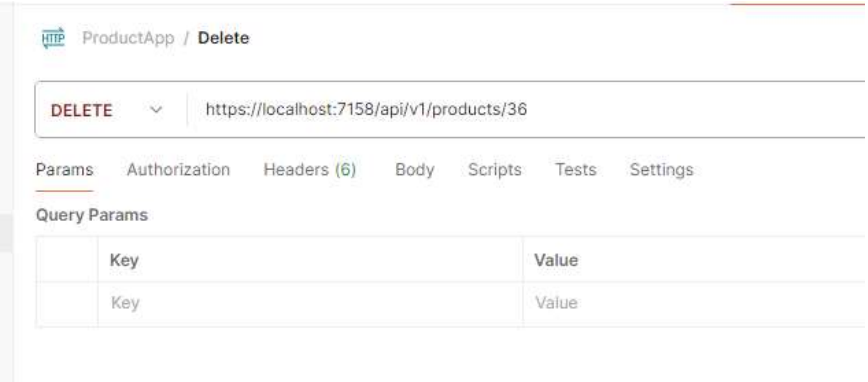


Figure 19 Delete action being performed by Postman.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

12. After the Delete request, the status is 204, success, but with no content to display.

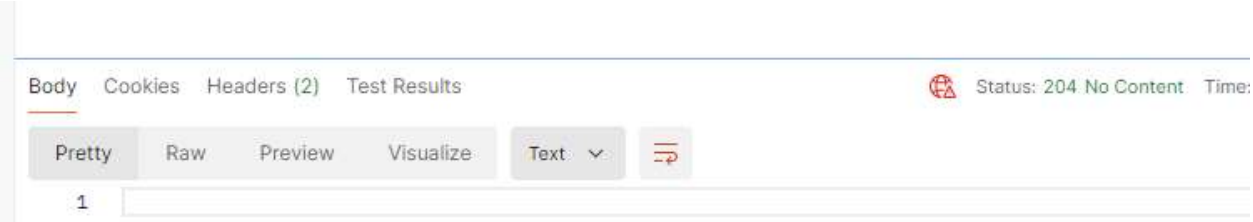


Figure 20 A 204 response shows a successful delete request has been processed.

13. In order to update a product record we submit a PUT request. You can either fill out the form-data fields or choose "Bulk Edit" to input the values.

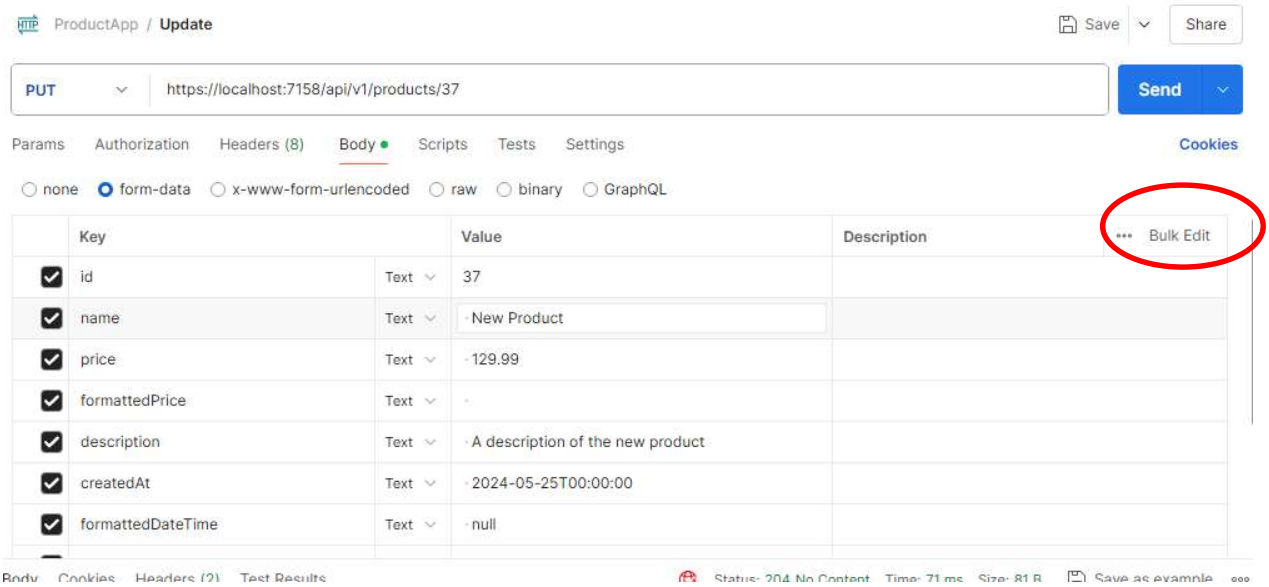


Figure 21 A PUT request for product 37 is submitted along with the form data.

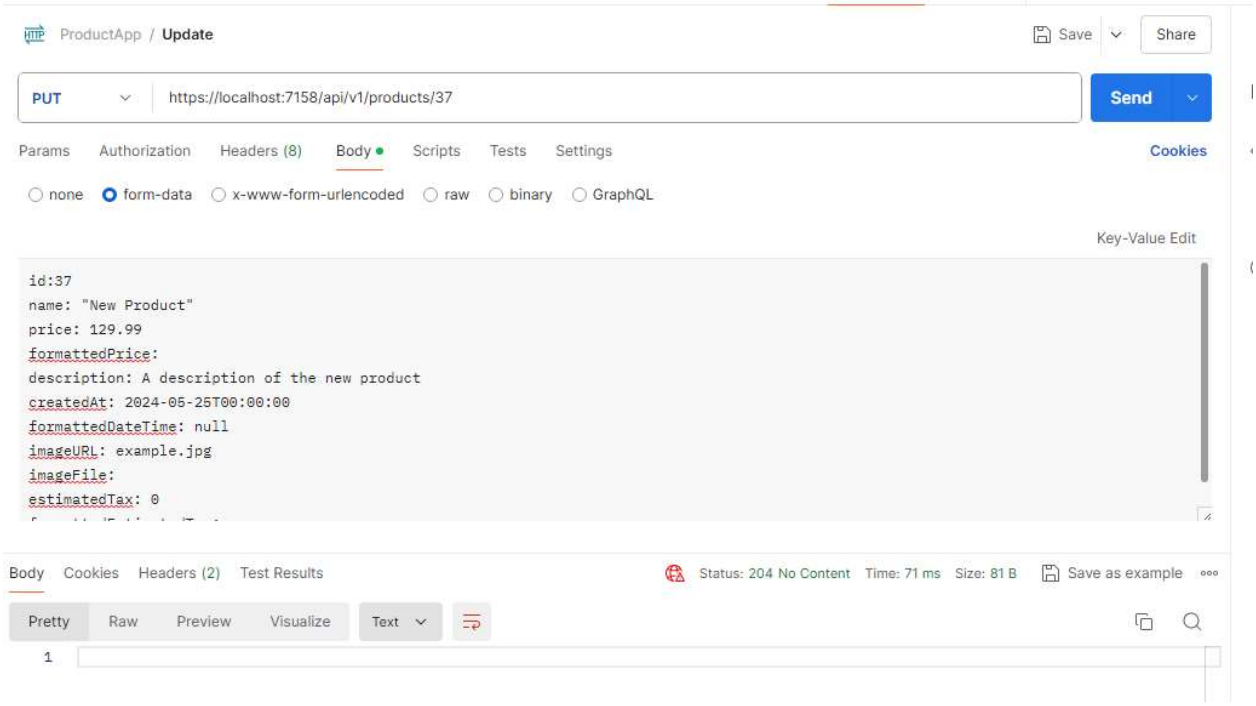


Figure 22 "Bulk Edit" option allows the user of Postman to paste or type a list of value pairs.

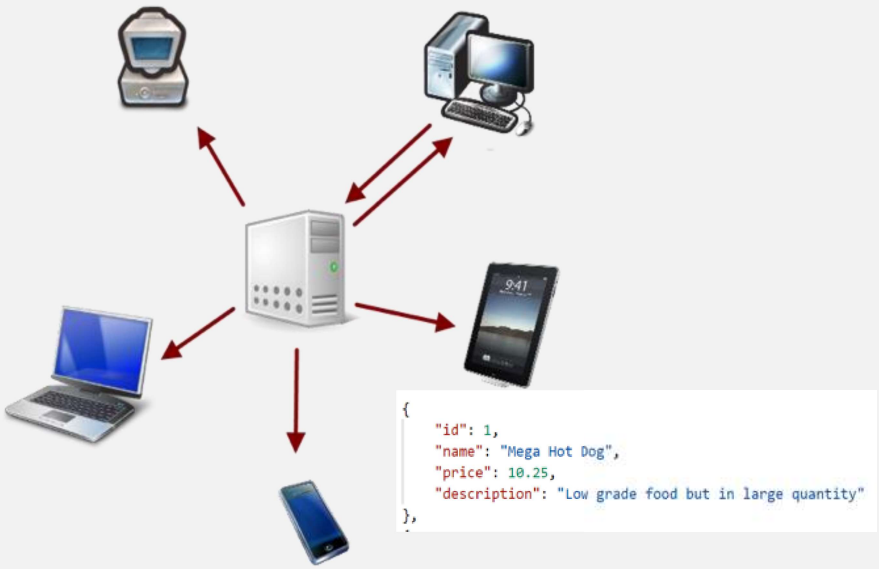
14. The response of a PUT command is a 204 success.

15. Verify the update was successful by viewing the getbyId method.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

How Can I Use an API in Other Applications?

APIs (Application Programming Interfaces) enable different software applications to communicate with each other. Here's how you can use an API to provide services to various types of applications:



Mobile Apps:

Integration: Mobile apps (iOS, Android) can make HTTP requests to your API to fetch data, submit forms, or interact with server-side functionality.

Real-time Updates: APIs can be used to push real-time updates to mobile apps using WebSockets or similar technologies.

Other Web Applications:

Data Sharing: Web applications can consume APIs to share data and functionality. For instance, a front-end application built with React, Vue or Angular can interact with a backend API to manage user data.

Microservices: In a microservices architecture, APIs enable different services to communicate. Each microservice can expose its own API, which other services or client applications can consume.

Other Types of Applications:

IoT Devices: Internet of Things (IoT) devices can use APIs to send data to the cloud or receive instructions from server-side applications.

Desktop Applications: Desktop software can use APIs to extend functionality, such as fetching data from the internet or integrating with other services.

Automation Scripts: APIs can be called from scripts or batch processes to automate tasks like data entry, system monitoring, or batch processing.

How APIs Help Division of Labor Between Front-End and Back-End Teams

APIs (Application Programming Interfaces) play a crucial role in dividing labor between front-end and back-end development teams. Here's how they help:

Clear Separation of Concerns:

Front-End Team: Focuses on building the user interface (UI) and user experience (UX). They work on designing responsive layouts, implementing user interactions, and ensuring accessibility.

Back-End Team: Focuses on server-side logic, database interactions, and business logic. They handle data storage, processing, authentication, and authorization.

Parallel Development:

Independent Progress: Since APIs act as a contract between front-end and back-end teams, both teams can work in parallel. The front-end team can develop against mock APIs or stubs, while the back-end team works on actual API implementation.

Faster Development Cycles: This parallel development accelerates the overall development process, allowing for quicker iterations and faster delivery.

Consistency and Standardization:

Uniform Interface: APIs provide a consistent and standardized way for the front-end to interact with the back-end. This reduces ambiguity and ensures both teams have a clear understanding of how data and services are accessed and manipulated.

Documentation: Good API documentation acts as a guide for front-end developers, detailing available endpoints, request/response formats, and error handling.

Flexibility and Scalability:

Modular Approach: APIs enable a modular approach to development. Front-end and back-end components can be developed, tested, and deployed independently.

Scalability: Backend services can be scaled independently of the front-end. This ensures that performance bottlenecks can be addressed without affecting the user interface.

Reduced Dependencies:

Loose Coupling: APIs create a loose coupling between front-end and back-end components. Changes in the back-end can be made without requiring changes in the front-end, as long as the API contract is maintained.

Maintenance: Easier to maintain and update individual components without risking the entire application's functionality.

Testing and Debugging:

Isolated Testing: Both teams can test their respective components independently. Front-end developers can use tools like Postman to test API endpoints, while back-end developers can use unit tests to ensure their services work correctly.

Error Handling: Clear error messages and status codes in API responses help front-end developers handle issues gracefully and provide better user feedback.

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Part 2 Right Mouse Click

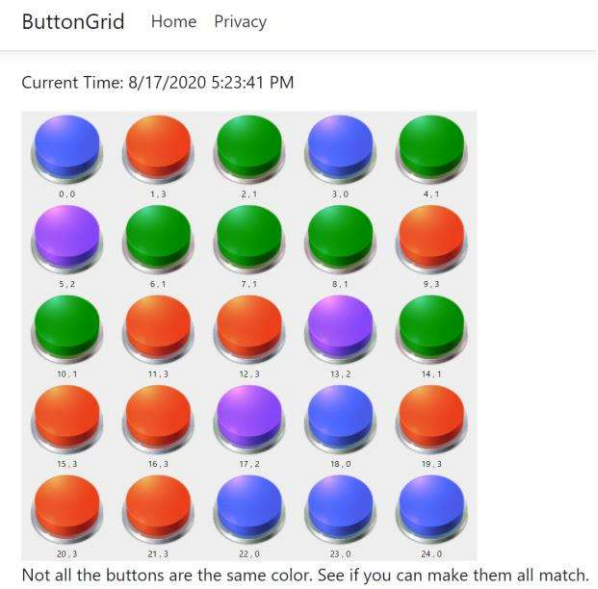
Goal and Directions:

In this activity, you will process a right mouse button click using JavaScript.

You will need the code from the **Button grid** application.

Current State:

After completing the previous tutorial with the Button grid app, the buttons respond to a **left click**. We will add the right click response next.



Instructions

1. In the **ButtonController**, create a new method called **RightClickShowOneButton**. The new method will reset the button state to 0.

```
// a right click subtracts 1 (same as add 3) from the button state and updates the image
public IActionResult RightClickPartialPageUpdate(int id)
{
    ButtonModel button = buttons.FirstOrDefault(b => b.Id == id);
    if (button != null)
    {
        button.ButtonState = (button.ButtonState + 3) % 4;
        button.ButtonImage = butonImages[button.ButtonState];
    }
    return PartialView("_Button", button);
}
```

2. In the button.js file, change the click event to a mousedown event. Handle each mouse button in a different method.

```
@section Scripts {
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    <script>
        $(document).ready(function () {
            $(document).on("mousedown", ".game-button button", function (e) {
                e.preventDefault();
                var buttonId = $(this).val();
                switch (e.which) {
                    case 1:
                        alert("Left mouse button clicked on item " + buttonId + ".");
                        doLeftClick(buttonId);
                        break;
                    case 2:
                        alert("Middle mouse button clicked on item " + buttonId + ".");
                        // do nothing
                        break;
                    case 3:
                        alert("Right mouse button clicked on item " + buttonId + ".");
                        doRightClick(buttonId);
                        break;
                    default:
                        alert("Unknown mouse button clicked on item " + buttonId + ".");
                }
            });  // end document on mousedown


            function doLeftClick(buttonId) {
                alert("do left click" + buttonId);
                $.ajax({
                    type: "POST",
                    url: "/Button/PartialPageUpdate",
                    data: { id: buttonId },
                    success: function (data) {
                        alert(data);
                        var buttonDiv = $(".game-button[data-id='" + buttonId + "']");
                        buttonDiv.html(data);
                    },
                    error: function (xhr, status, error) {
                        console.error("An error occurred: " + error);
                    }
                });
            } // end doLeftClick

            function doRightClick( buttonId) {
                alert("do right click" + buttonId);
                $.ajax({
                    type: "POST",
                    url: "/Button/RightClickPartialPageUpdate",
                    data: { id: buttonId },
                    success: function (data) {
                        alert(data);
                        var buttonDiv = $(".game-button[data-id='" + buttonId + "']");
                        buttonDiv.html(data);
                    },
                    error: function (xhr, status, error) {
                        console.error("An error occurred: " + error);
                    }
                });
            } // end do right click

            $(document).on("contextmenu", ".game-button button", function (e) {
                e.preventDefault(); // Prevent the default context menu on right-click
            });


        });
    </script>
}
```

mousedown method replaces click.

e parameter is “event”.

Send control to a helper function with id number,

Same behavior as the previous click event.

Only change in right click is a different method in the button controller.

3. Run the program. Test the status messages for each event.
4. After testing, removing the annoying alert statements.

- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Conclusions

What You Learned

Here are the key takeaways from this lesson

1. **RESTful Services Principles**: You learned the fundamental principles of REST (Representational State Transfer). This was put into practice by creating a REST controller that handles various HTTP requests to manage product data.
2. **Creating and Testing REST Controllers**: You developed REST controllers in ASP.NET Core, modifying existing controllers to return JSON responses instead of views. This included:
   - Changing **ProductsController** to **ProductsRestController**
   - Using attributes like **[ApiController]** and **[Route("api/[controller]")]**
   - Implementing methods like **ShowAllProducts** to return JSON data
3. **MVC vs. REST Controllers**: You understood the key differences between traditional MVC applications and RESTful APIs, focusing on aspects like controller type, return type, response content, and HTTP response codes. This was demonstrated through the comparison of **ShowAllProducts** methods in MVC and REST contexts.
4. **CRUD Operations via REST API**: You extended an existing application to support Create, Read, Update, and Delete (CRUD) operations through a REST API. This involved:
   - Implementing methods for creating, updating, and deleting products
   - Handling HTTP verbs such as GET, POST, PUT, and DELETE
   - Testing these methods using Postman to ensure correct functionality.
5. **Using Postman for API Testing**: You used Postman to test your API endpoints, which included sending different types of HTTP requests and verifying the responses. This tool allowed you to:
   - Send and customize HTTP requests easily
   - Save and organize requests for reuse
   - Test the functionality of your RESTful services without relying on traditional HTML forms
6. **HTTP Status Codes**: You learned about various HTTP status codes beyond the commonly known 200 (OK) and 404 (Not Found). These codes provide important information about the result of HTTP requests, and you implemented these in your API responses to give precise feedback to clients.
7. **Handling Right-Click Events**: You enhanced the Button Grid application to respond to right-click events using JavaScript. This included:
   - Adding a method in **ButtonController** to handle right-click interactions
   - Updating the JavaScript file to manage mouse events and trigger the appropriate actions based on the mouse button clicked

You have acquired valuable skills in developing and testing RESTful services, understanding the division of labor between front-end and back-end teams, and enhancing user interactions in web applications using JavaScript.

Check for Understanding

Although these questions are not graded, they will help you prepare for upcoming assessments.

1. **Which attribute is used to help the controller interpret URLs and parameters correctly in a REST API?**
    - A) [Controller]
    - B) [ApiController]
    - C) [HttpController]
    - D) [Route]

2. **What is the return type used in REST controllers to represent various HTTP status codes and response types?**
    - A) Task<IActionResult>
    - B) ActionResult
    - C) Task<ActionResult<IEnumerable<ProductViewModel>>>
    - D) JsonResult

3. **Which tool was used in the lesson to test and debug APIs without relying on HTML form requests?**
    - A) Visual Studio
    - B) Fiddler
    - C) Postman
    - D) Swagger

4. **What HTTP status code indicates that a resource was successfully created?**
    - A) 200 OK
    - B) 201 Created
    - C) 204 No Content
    - D) 400 Bad Request

5. **What is the primary benefit of using the [Route("api/[controller]")] attribute in a REST controller?**
    - A) It enables the controller to return views.
    - B) It helps to validate data models.
    - C) It specifies the URLs the service will respond to.
    - D) It handles HTTP POST requests.

6. **In the context of RESTful APIs, what does the term "CRUD operations" stand for?**
    - A) Create, Read, Update, Delete
    - B) Create, Retrieve, Update, Delete

- C) Compile, Run, Update, Debug
- D) Connect, Request, Update, Delete

7. **Why is the ControllerBase class used instead of the Controller class in REST APIs?**
   - A) To support server-side rendering of views.
   - B) To add members that are needed to support views.
   - C) To focus on returning data directly in the response body.
   - D) To handle user interactions within the web application's UI.

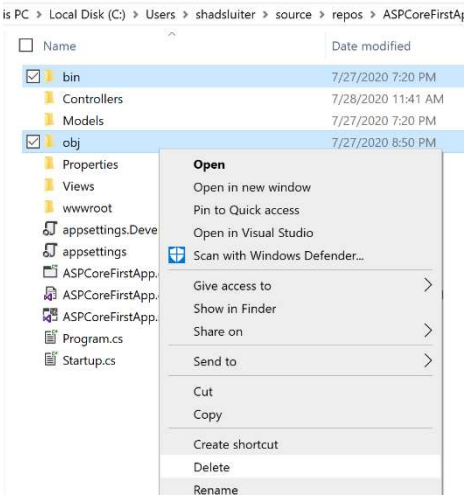
8. **Which method in JavaScript was used to handle right-click events in the Button Grid application?**
   - A) .click()
   - B) .on("mousedown")
   - C) .contextmenu()
   - D) .dblclick()

9. **What is the purpose of the Ok(products) statement in a REST controller method?**
   - A) To render a view with the products data.
   - B) To return an HTTP 200 OK status code and serialize the products data to JSON.
   - C) To trigger a validation error.
   - D) To update the products in the database.

10. **How do APIs help in dividing labor between front-end and back-end development teams?**
    - A) By allowing both teams to work independently and in parallel.
    - B) By focusing only on front-end development.
    - C) By ensuring that back-end developers design the user interface.
    - D) By combining front-end and back-end code into a single monolithic application.

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.



Check For Understanding Answers

1. B
2. C
3. C
4. B
5. C
6. A
7. C
8. B
9. B
10. A