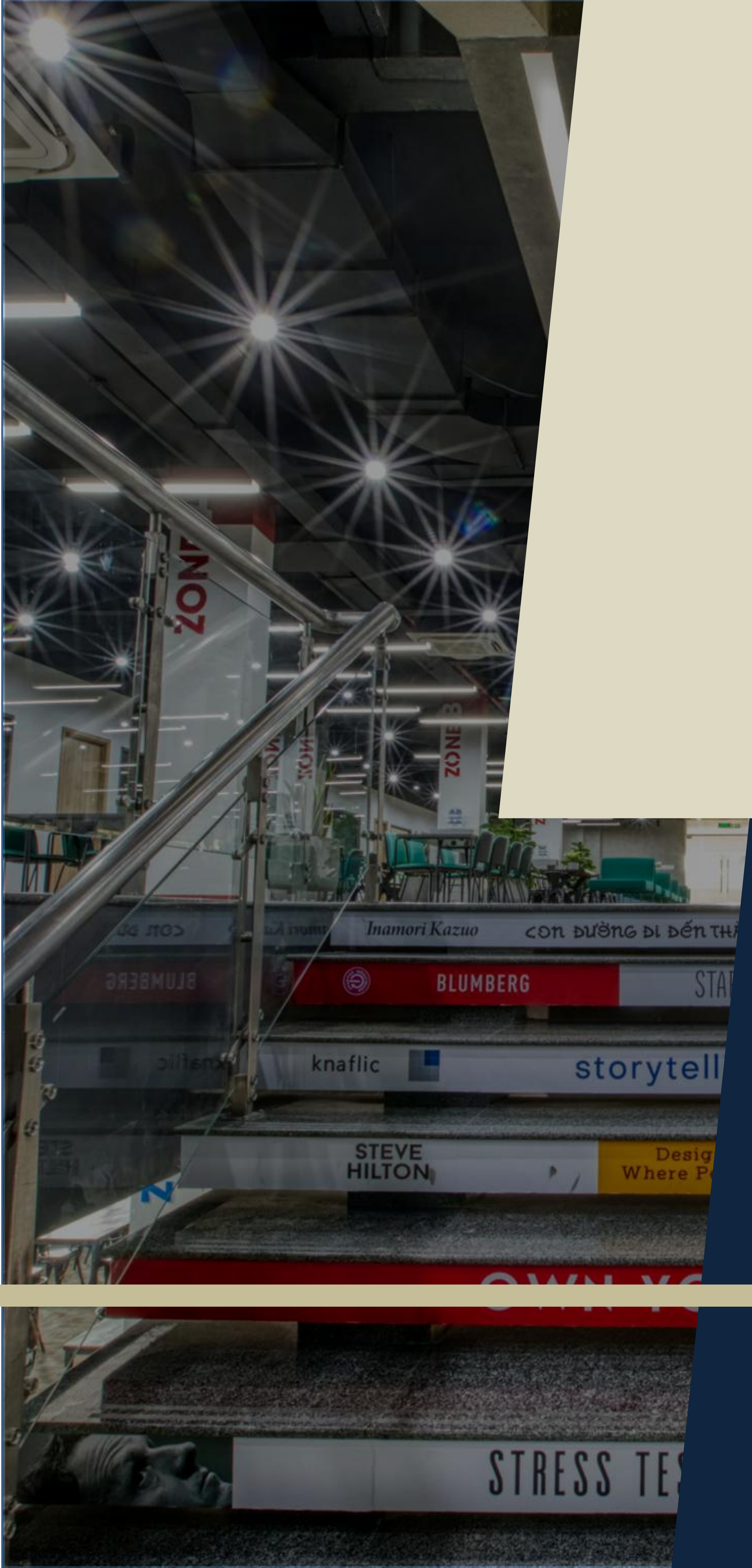


TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Backend Development

Chapter 5: Advanced ASP.NET CORE

TABLE CONTENT

Web API and AJAX

EIU

Introduction to Web API and MVC Integration

Integrating ASP.NET Core Web API with MVC

- Introduction to what you'll be covering: integration of Web API with MVC and using Ajax to call Web API from the MVC view.
- Why this integration is crucial for modern web applications (separating concerns, flexibility, and API consumption).

Introduction to Web API and MVC Integration

Benefits of Using Web API with MVC

- Explain the benefits of keeping APIs and views separate.
- Flexibility for mobile applications, external consumers, and enhancing modularity.
- How Web APIs are lightweight and easily consumable via HTTP (without full-page reloads).

Introduction to Web API and MVC Integration

Real-Time Communication with Ajax

- Introduce Ajax (Asynchronous JavaScript and XML).
- Its role in Web API calls from MVC, allowing partial page updates without refreshing the whole page.
- Example scenario: updating a product list without reloading the entire page.

Introduction to Web API and MVC Integration

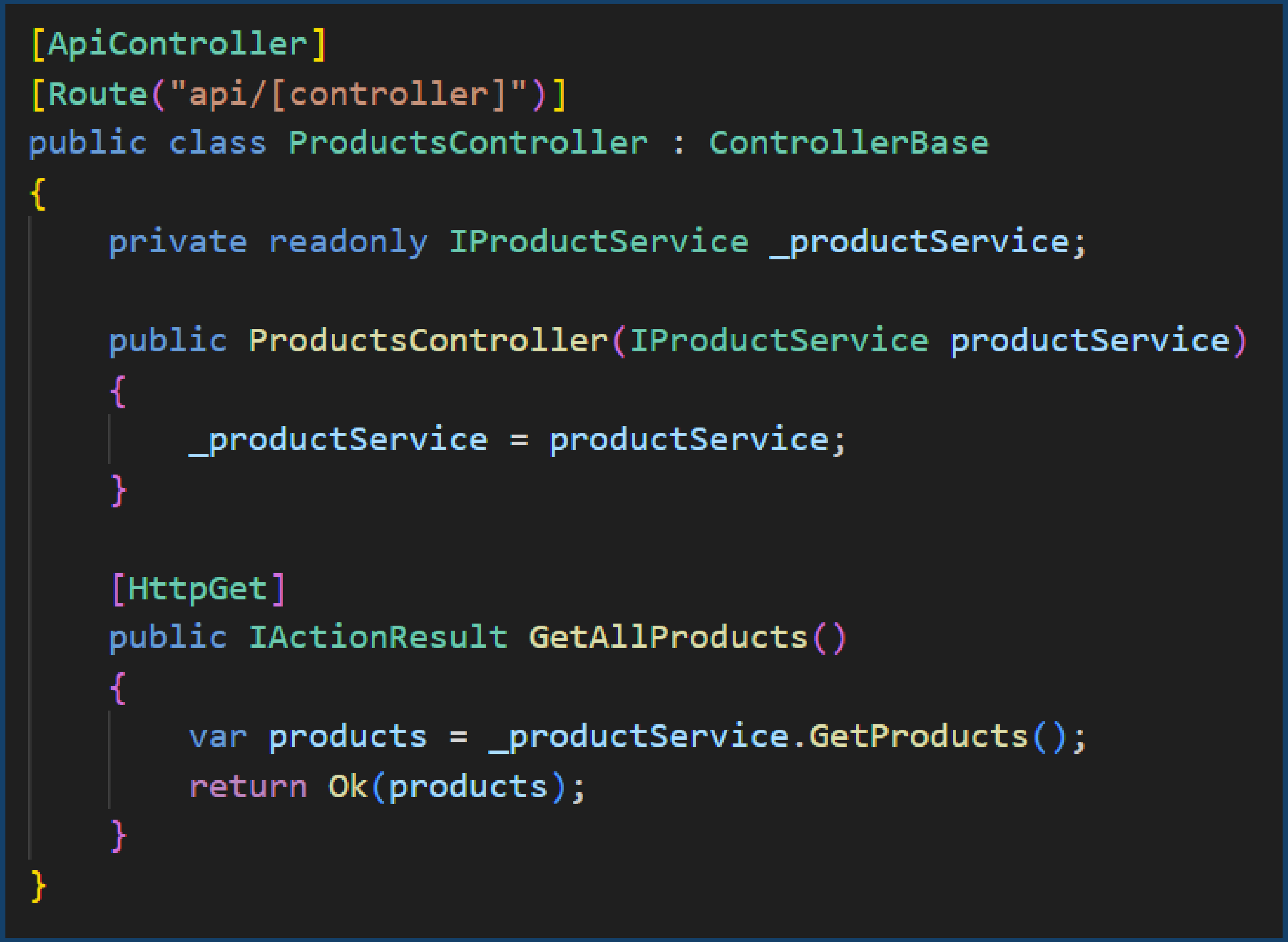
Overview of ASP.NET Core MVC Structure

- Briefly explain how MVC (Model-View-Controller) works in ASP.NET Core.
- Highlight the relationship between controllers, views, and models, and how Web API fits into the controller role.

Setting Up Web API in an MVC Project

Adding a Web API Controller in ASP.NET Core MVC

- Explain how to add a Web API controller to an existing MVC project.
- Show a basic code example of a Web API controller (ProductsController).



```csharp
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly IProductService _productService;

    public ProductsController(IProductService productService)
    {
        _productService = productService;
    }

    [HttpGet]
    public IActionResult GetAllProducts()
    {
        var products = _productService.GetProducts();
        return Ok(products);
    }
}
```

Setting Up Web API in an MVC Project

Web API Routing

- Explain the differences between MVC routing and Web API routing.
- How Web API uses attribute-based routing ([Route("api/[controller]")]).
- Example of routing in the ProductsController for various actions (e.g., GET, POST).

Setting Up Web API in an MVC Project

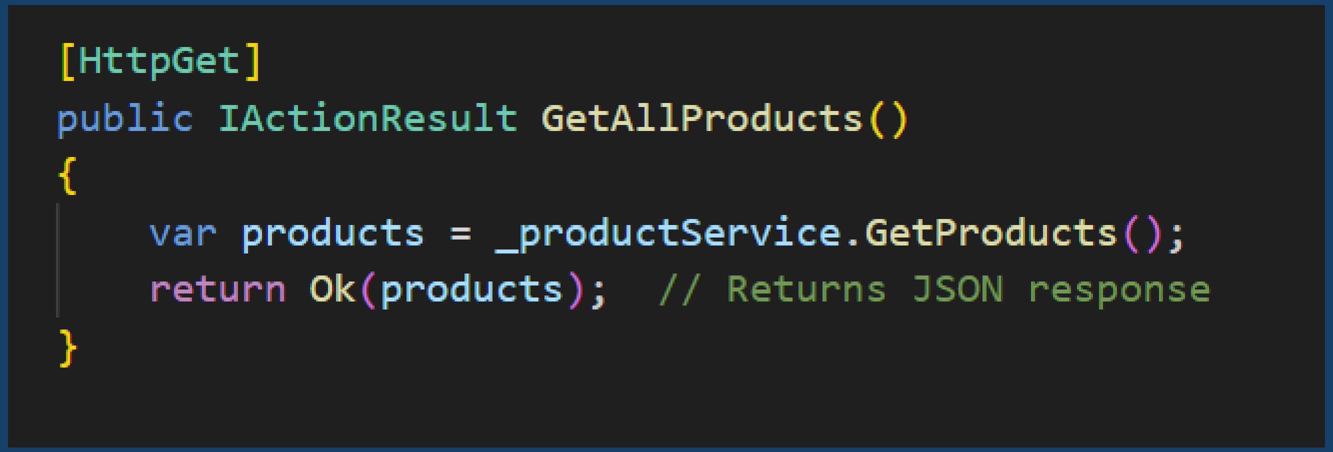
Testing API Endpoints in Postman

- Before integrating with MVC, demonstrate how to test Web API endpoints using Postman.
- Example of testing the GET /api/products endpoint to ensure it works.

Setting Up Web API in an MVC Project

Adding JSON Response in API

- Explain how Web API controllers return JSON by default (as the expected format for frontend applications).
- Show an example of returning JSON data from a Web API.

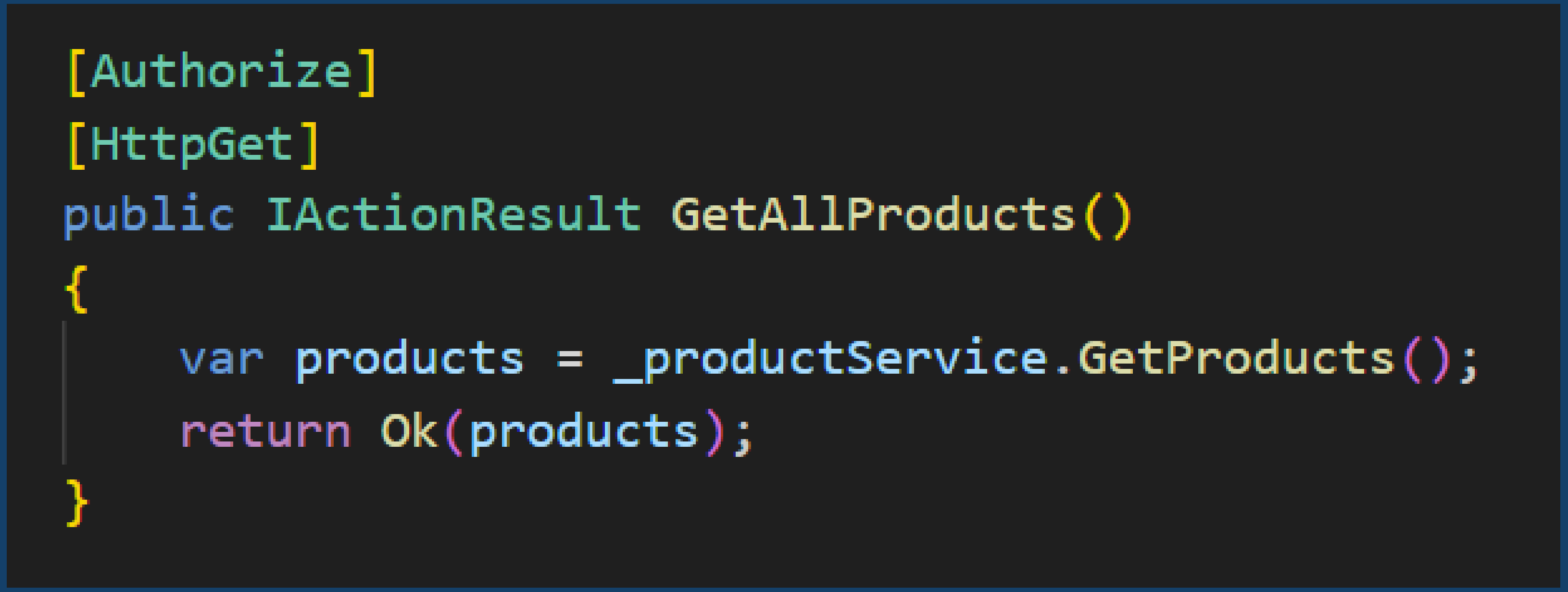


```
[HttpGet]
public IActionResult GetAllProducts()
{
    var products = _productService.GetProducts();
    return Ok(products);  // Returns JSON response
}
```

Setting Up Web API in an MVC Project

Securing Web API Endpoints

- Briefly discuss securing Web API endpoints using [Authorize] to ensure authenticated users can only access the API.
- Example: Adding the [Authorize] attribute to protect the ProductsController.



```
[Authorize]
[HttpGet]
public IActionResult GetAllProducts()
{
    var products = _productService.GetProducts();
    return Ok(products);
}
```

Calling Web API from an MVC View using Ajax

Introduction to Ajax in ASP.NET Core MVC

- Explain what Ajax is (asynchronous calls to the server using JavaScript).
- How it improves user experience by loading data dynamically without a page reload.

Calling Web API from an MVC View using Ajax

Setting Up jQuery for Ajax

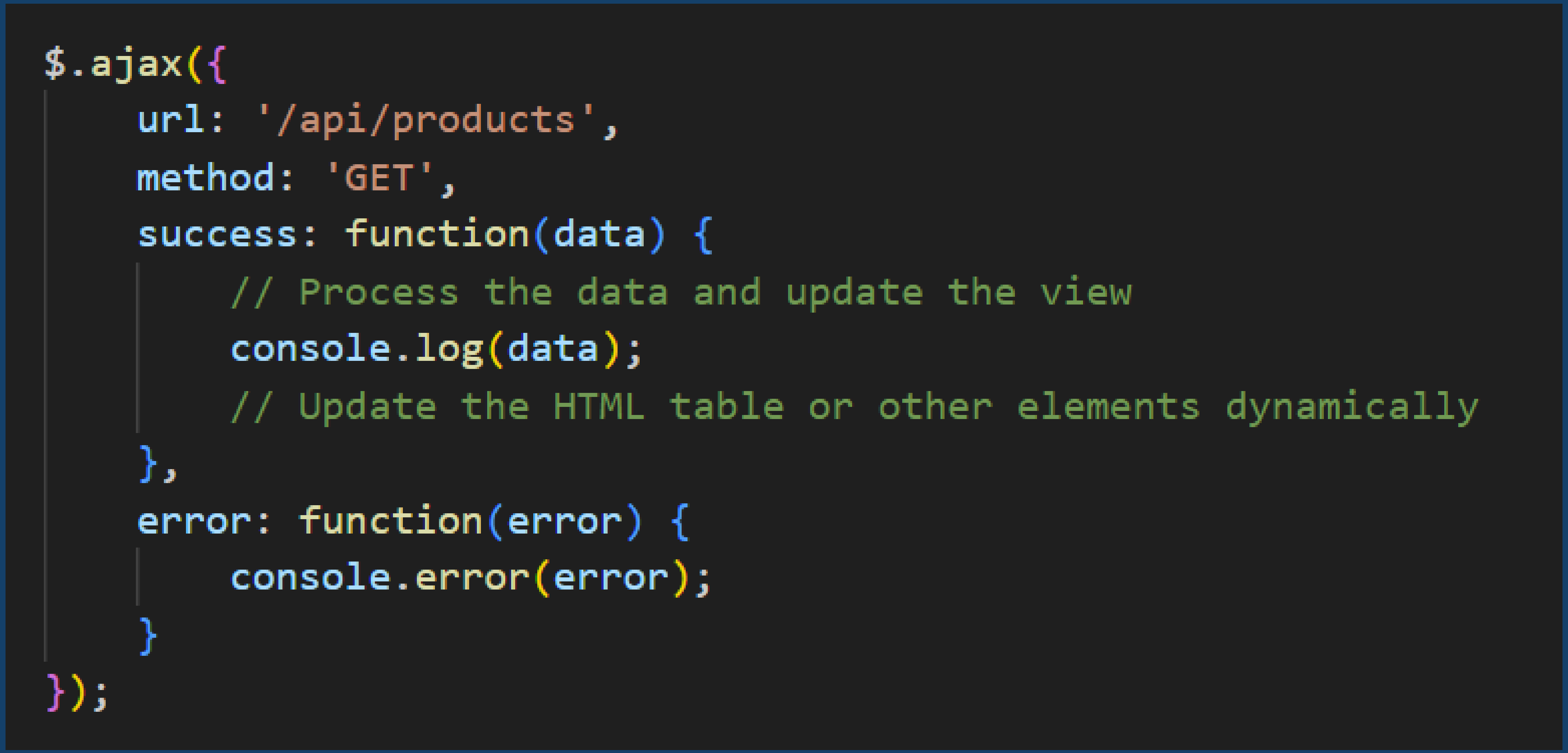
- Show how to add jQuery to an MVC project for making Ajax calls.
- Example: Include jQuery in the _Layout.cshtml.

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
```

Calling Web API from an MVC View using Ajax

Ajax GET Request to Web API

Example of how to make a GET request to a Web API from the MVC view using Ajax.

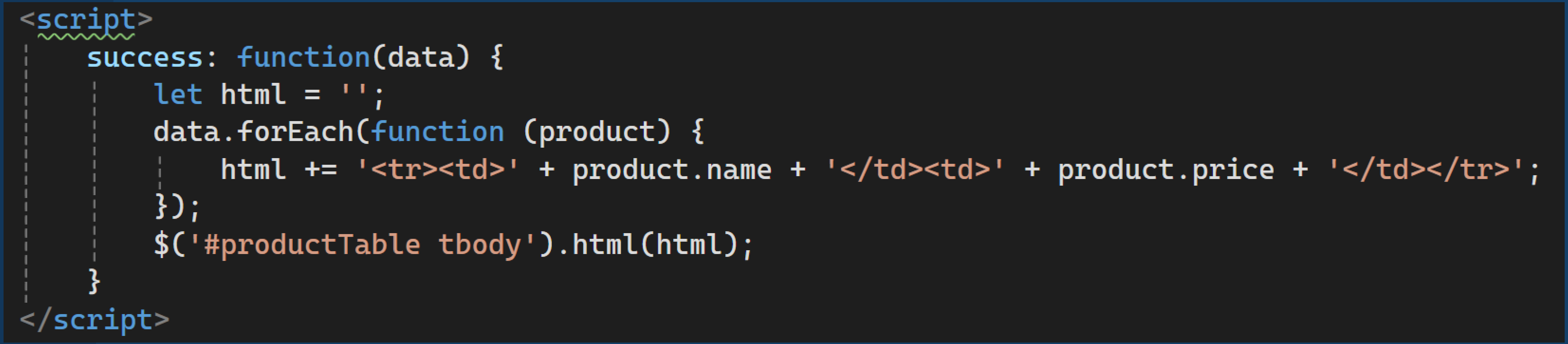


```
$.ajax({
    url: '/api/products',
    method: 'GET',
    success: function(data) {
        // Process the data and update the view
        console.log(data);
        // Update the HTML table or other elements dynamically
    },
    error: function(error) {
        console.error(error);
    }
});
```

Calling Web API from an MVC View using Ajax

Handling Ajax Responses in the View

- How to dynamically update the HTML view (e.g., a table) with the data returned from the Ajax call.
- Example of updating an HTML table with product data:

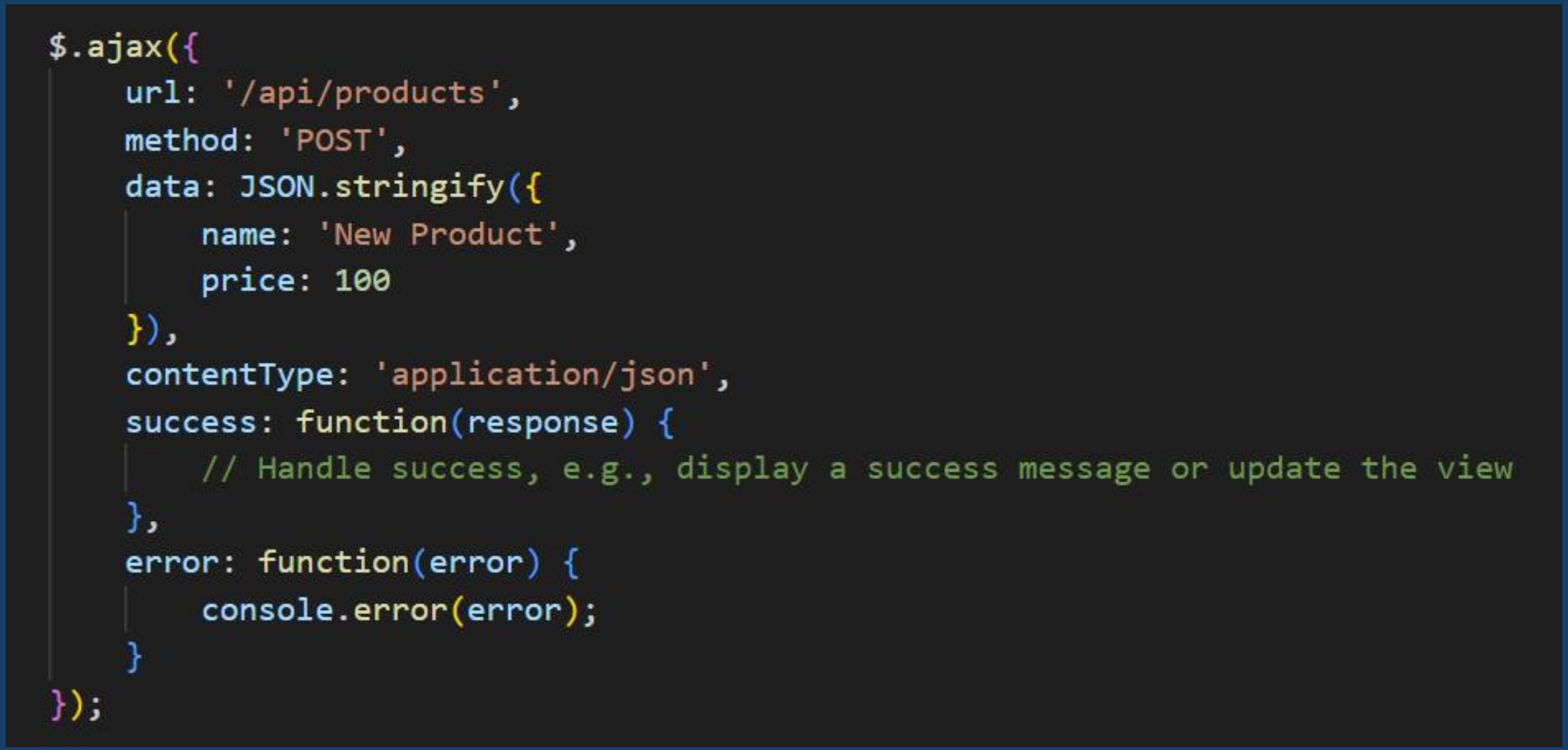


```
<script>
    success: function(data) {
        let html = '';
        data.forEach(function (product) {
            html += '<tr><td>' + product.name + '</td><td>' + product.price + '</td></tr>';
        });
        $('#productTable tbody').html(html);
    }
</script>
```

Calling Web API from an MVC View using Ajax

Ajax POST Request to Web API

- Example of using Ajax to send data from a form in the MVC view to the Web API using a POST request.



```
$.ajax({
    url: '/api/products',
    method: 'POST',
    data: JSON.stringify({
        name: 'New Product',
        price: 100
    }),
    contentType: 'application/json',
    success: function(response) {
        // Handle success, e.g., display a success message or update the view
    },
    error: function(error) {
        console.error(error);
    }
});
```

Real-time Applications

Introduction to Real-time Applications

Real-time Applications with ASP.NET Core 8.0

ASP.NET Core 8.0 provides the tools to build fast, scalable real-time web applications. We’ll explore real-time technologies like SignalR and how to integrate them into your web applications.

Introduction to Real-time Applications

What are Real-time Applications ?

- Real-time applications enable dynamic updates to content or UI without requiring the user to refresh the page. This provides an instant response to user actions or external events.
- **Examples of Real-time Applications**:
    - **Online Chat Applications**: Messages appear instantly as users send them.
    - **Collaborative Document Editing**: Multiple users can edit a document in real-time (e.g., Google Docs).
    - **Live Stock Market Feeds**: Prices update in real-time as the stock market fluctuates.
    - **Online Gaming**: Player positions and actions update instantly in multiplayer games.

Introduction to Real-time Applications

Challenges in Real-time Web Development

Traditional web applications operate on a request-response cycle, where the client sends a request and waits for a server response. This is not sufficient for real-time updates.

Challenges:

- **Latency**: Delays between user actions and updates are not acceptable in real-time applications.
- **Scalability**: Handling many simultaneous real-time connections can strain server resources.
- **Connection Management**: Keeping a persistent connection between client and server is difficult across different network conditions and devices.

Solution: Real-time web apps require technologies like WebSockets, Server-Sent Events (SSE), or Long Polling, which provide two-way communication between the client and server.

Introduction to Real-time Applications

ASP.NET Core for Real-time Applications

ASP.NET Core is optimized for real-time applications due to its high performance and scalability features. It supports the following real-time communication technologies:

- **SignalR**: An open-source library that provides real-time functionality for ASP.NET Core.
- **WebSockets**: Enables bi-directional communication over a single, long-lived TCP connection.
- **Server-Sent Events (SSE)**: A lightweight solution for server-initiated updates over HTTP.
- **Long Polling**: A fallback mechanism that simulates real-time behavior for older browsers or networks where WebSockets aren’t available.

ASP.NET Core is also cross-platform, meaning you can deploy real-time applications to Windows, macOS, or Linux servers.

Introduction to Real-time Applications

Introduction to SignalR

SignalR is a framework that simplifies adding real-time web functionality to applications. It abstracts the complexities of real-time communication, automatically handling WebSockets and providing fallbacks for older browsers.

Why SignalR?:

- **Automatic Fallback**: SignalR automatically chooses the best transport method available (WebSockets, Server-Sent Events, or Long Polling).
- **Client-Server Communication**: With SignalR, the server can push updates to the client instantly, without the need for the client to request them.
- **Real-World Use Cases**: SignalR is ideal for chat applications, live dashboards, real-time notifications, and multiplayer games.

SignalR Basics in ASP.NET Core 8.0

What is SignalR ?

SignalR allows real-time communication between client and server in a way that’s easy to implement. It supports **WebSockets** for high-performance communication but can also fall back to **Server-Sent Events (SSE)** and **Long Polling** when necessary.

Core Features:

- **Real-time Messaging**: Clients can send messages to the server, and the server can broadcast messages to all connected clients.
- **Automatic Reconnect**: SignalR handles reconnecting automatically if the connection is lost.
- **Scalability**: SignalR can scale out to multiple servers by using backplane technologies like Redis or Azure SignalR Service.

EIU

SignalR Basics in ASP.NET Core 8.0

How SignalR Works

Hub-based Communication: SignalR uses a hub model where clients communicate with a central hub on the server. Clients can call methods on the hub, and the hub can call methods on the client.

Communication Flow:

- The **client** establishes a connection with the server hub.
- The **hub** sends or receives data from the client.
- **WebSockets** are used as the default communication method. If WebSockets aren’t available, SignalR falls back to SSE or Long Polling.
- **Scalability**: For high-traffic applications, you can scale SignalR using a backplane like **Redis**, **SQL Server**, or **Azure SignalR Service**.

SignalR Basics in ASP.NET Core 8.0

Real-time Communication Models

SignalR offers two communication models:

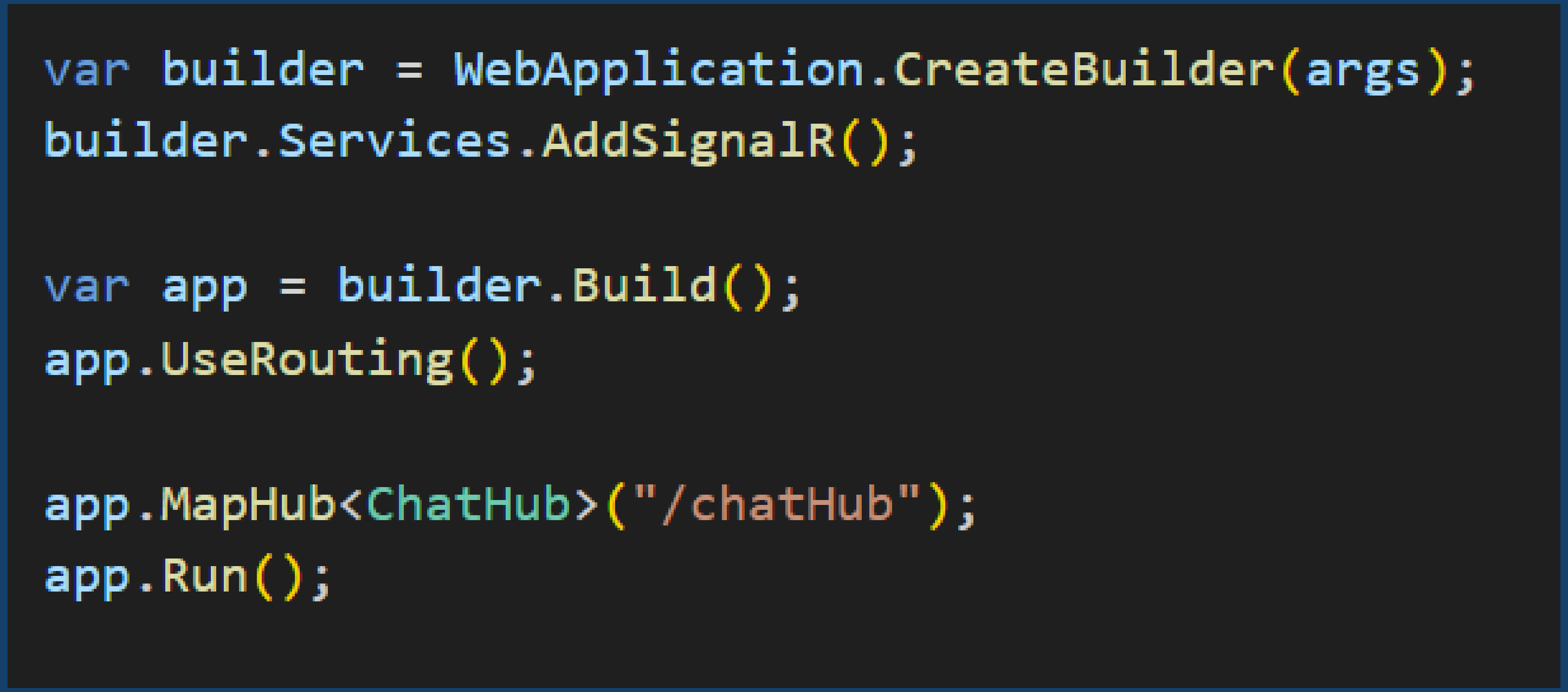
- **Hub-based Communication**: The server interacts with connected clients through a hub. This is the most commonly used approach for real-time apps, where the server broadcasts updates to all clients or sends data to specific ones.
- **Persistent Connections**: This is a more advanced model where you manage individual client connections directly. It's used for scenarios requiring more control over the connection, like low-level messaging.

Most applications use **hub-based communication** for simplicity and ease of use.

SignalR Basics in ASP.NET Core 8.0

Setting up SignalR in ASP.NET Core 8.0

Step 1: Install the SignalR package via NuGet:
Microsoft.AspNetCore.SignalR
Step 2: Register SignalR services in Program.cs



```csharp
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSignalR();

var app = builder.Build();
app.UseRouting();

app.MapHub<ChatHub>("/chatHub");
app.Run();
```

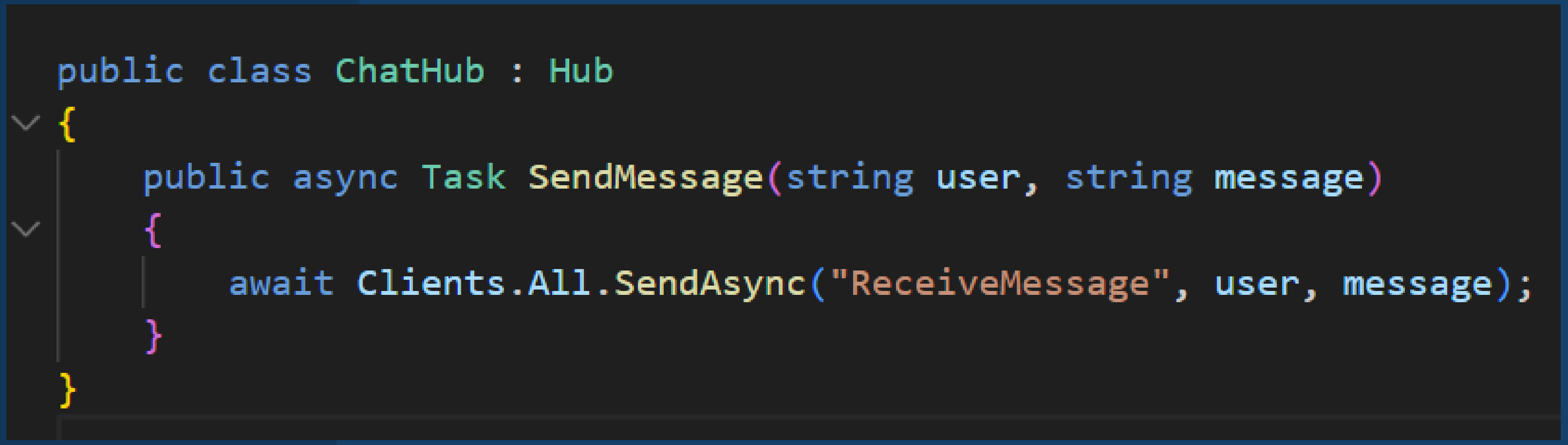
Explanation:

- AddSignalR: Registers the SignalR services.
- MapHub<ChatHub>("/chatHub"): Sets up a route for the SignalR hub at /chatHub, which clients will connect to for real-time communication.

SignalR Basics in ASP.NET Core 8.0

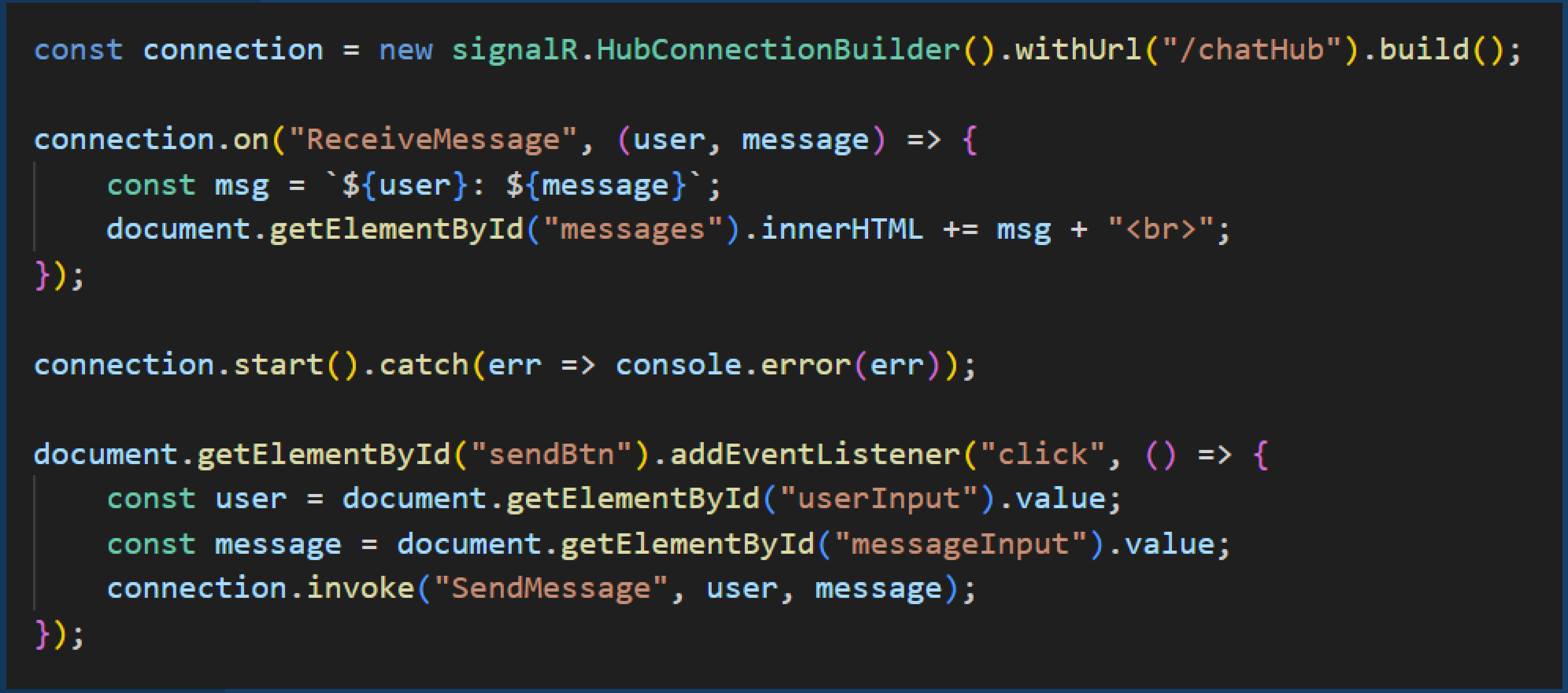
Code Example: Basic SignalR Setup

Server-side (Hub):



```csharp
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

Client-side (JavaScript):



```javascript
const connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

connection.on("ReceiveMessage", (user, message) => {
    const msg = `${user}: ${message}`;
    document.getElementById("messages").innerHTML += msg + "<br>";
});

connection.start().catch(err => console.error(err));

document.getElementById("sendBtn").addEventListener("click", () => {
    const user = document.getElementById("userInput").value;
    const message = document.getElementById("messageInput").value;
    connection.invoke("SendMessage", user, message);
});
```

Explanation:

- The server-side hub (ChatHub) defines a method (SendMessage) that clients can call to send messages.
- The client establishes a connection to the hub, listens for incoming messages (ReceiveMessage), and displays them in the UI.
- When the user clicks the "Send" button, the client invokes the SendMessage method on the hub.

EIU

Real-time Applications in MVC

Using SignalR in MVC

SignalR is fully compatible with MVC applications. It can be used to update the user interface in real-time from the server side.

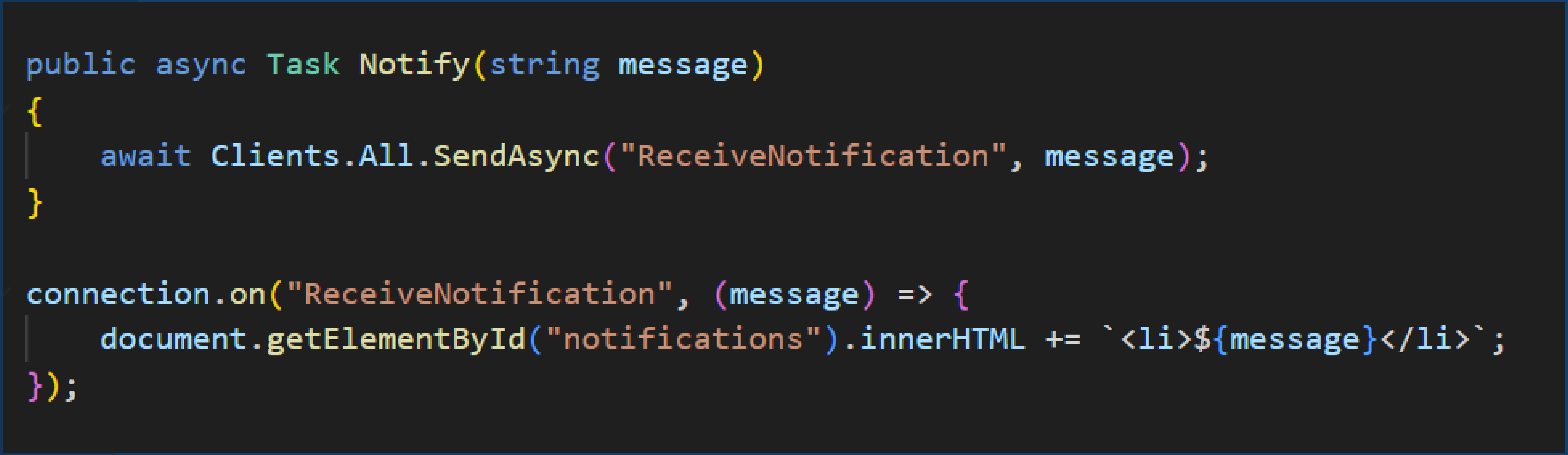
Scenario: Imagine a live comments feature for a blog post where new comments appear instantly without requiring page refresh.

Real-time Applications in MVC

Real-time UI Updates with SignalR

Scenario: Live Notifications

Example: Add a notification system where updates (e.g., new messages or alerts) automatically appear in the UI.

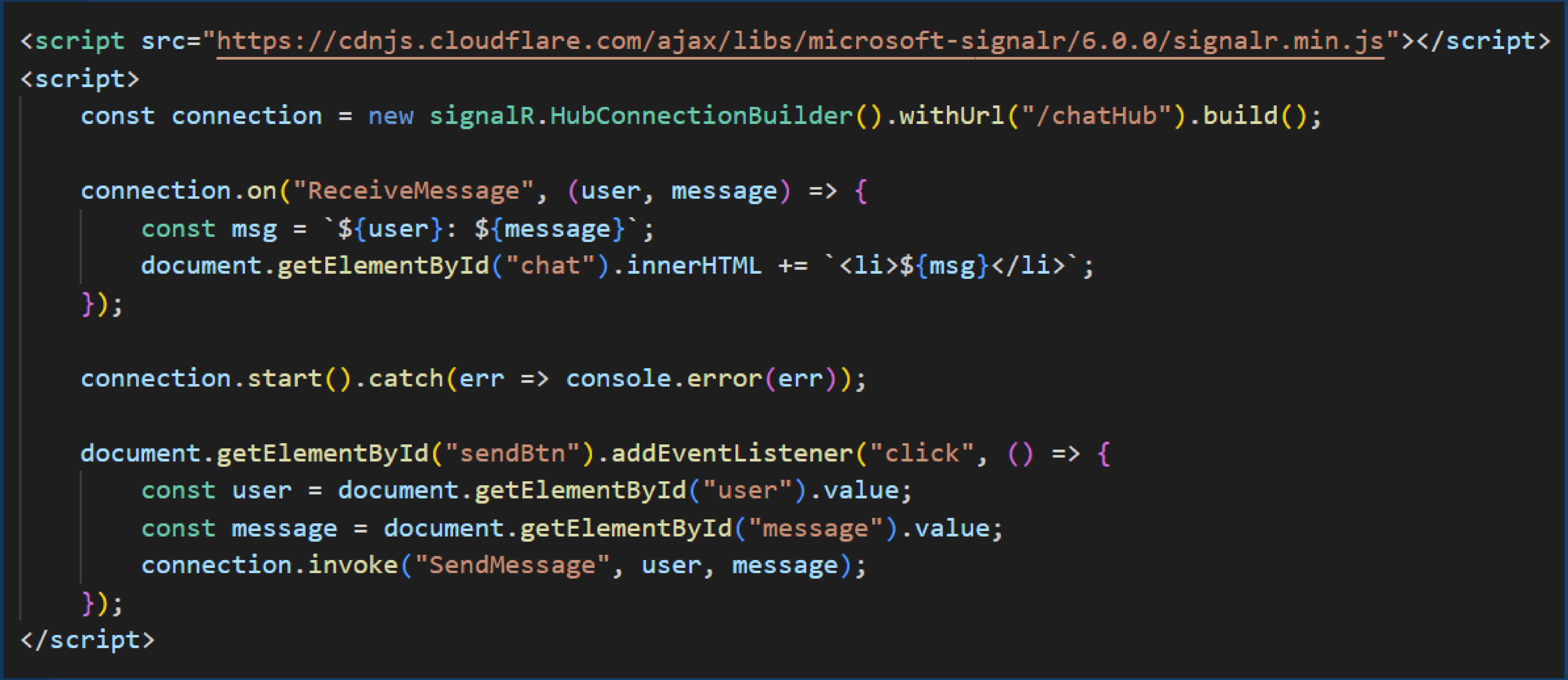


```
public async Task Notify(string message)
{
    await Clients.All.SendAsync("ReceiveNotification", message);
}

connection.on("ReceiveNotification", (message) => {
    document.getElementById("notifications").innerHTML += `<li>${message}</li>`;
});
```

Real-time Applications in MVC

Adding SignalR to an MVC View

In ASP.NET Core MVC, you can add SignalR scripts and calls directly in your views using Razor.

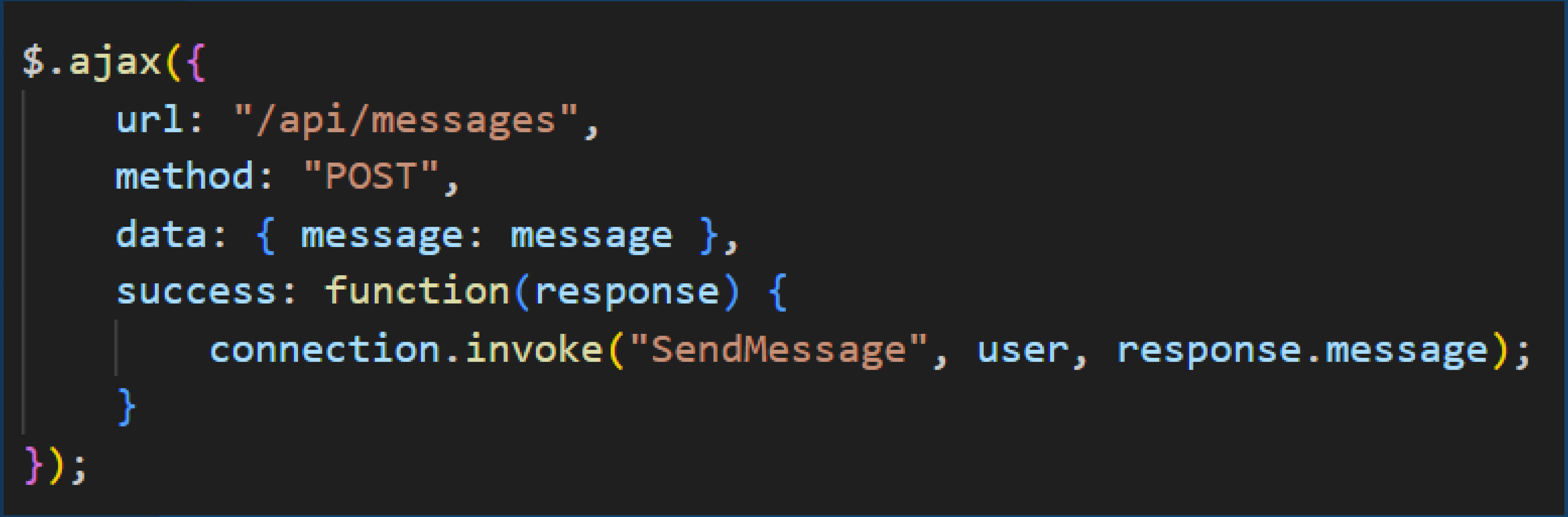


```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-signalr/6.0.0/signalr.min.js"></script>
<script>
    const connection = new signalR.HubConnectionBuilder().withUrl("/chatHub").build();

    connection.on("ReceiveMessage", (user, message) => {
        const msg = `${user}: ${message}`;
        document.getElementById("chat").innerHTML += `<li>${msg}</li>`;
    });

    connection.start().catch(err => console.error(err));

    document.getElementById("sendBtn").addEventListener("click", () => {
        const user = document.getElementById("user").value;
        const message = document.getElementById("message").value;
        connection.invoke("SendMessage", user, message);
    });
</script>
```

Real-time Applications in MVC

Integrating SignalR with Ajax

Use **Ajax** calls to interact with your API endpoints in MVC and then update the UI with SignalR.



```
$.ajax({
    url: "/api/messages",
    method: "POST",
    data: { message: message },
    success: function(response) {
        connection.invoke("SendMessage", user, response.message);
    }
});
```

Real-time Applications in MVC

Example: Real-time Data Updates

Scenario: A real-time dashboard that updates as new data comes in (e.g., financial updates or user statistics).

- **SignalR Hub**: Handles broadcasting data.
- **Client-side JavaScript**: Updates the UI as new data is received.
- **Benefits**: Instant updates for end-users without reloading the page.

Security in Real-time Applications

Securing Real-time Applications

Security is critical in real-time applications, especially with SignalR. You must protect the communication channel and authenticate users.

Security Concerns:

- **Unauthorized Access**: Only authorized users should be allowed to connect to the hub.
- **Data Integrity**: Ensure the messages exchanged between clients and the server are not tampered with.

Security in Real-time Applications

SignalR Authentication

Authentication: You can use standard ASP.NET Core authentication mechanisms (cookies, JWT, etc.) with SignalR.

Example: Configure JWT in Startup.cs to authenticate SignalR connections.

Client Connection: Include authentication tokens when connecting:

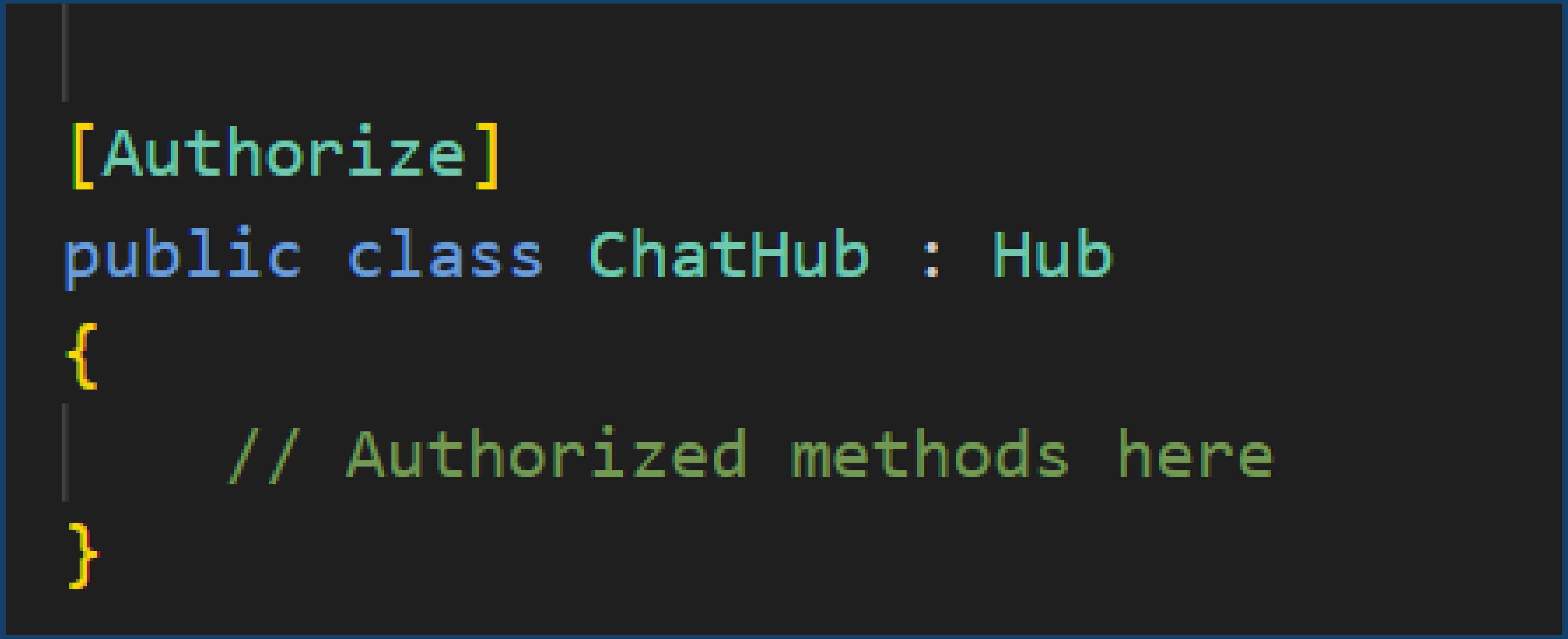
```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chatHub", { accessTokenFactory: () => "your-jwt-token" })
    .build();
```

Security in Real-time Applications

Authorization in SignalR Hubs

Use ASP.NET Core’s **[Authorize]** attribute to restrict access to certain hubs or methods based on the user's role or claims.

Example:



```
[Authorize]
public class ChatHub : Hub
{
    // Authorized methods here
}
```

Security in Real-time Applications

Data Privacy Considerations

- Ensure that sensitive data is never exposed in client-side code.
- **Use HTTPS**: Ensure all communication is encrypted.
- **Secure Data Transmission**: Encrypt messages between clients and the server to protect data integrity.

Security in Real-time Applications

Real-time Applications in the Cloud

Azure SignalR Service: A managed service for scaling real-time applications with SignalR in the cloud.

Benefits:

- Simplifies scaling for large applications.
- Offloads connection management and scaling complexities.
- **Integration**: Works seamlessly with ASP.NET Core apps.

Case Study and Conclusion

Case Study: Building a Live Chat Application

Walkthrough of creating a live chat system with SignalR in ASP.NET Core 8.0.

Key Concepts:

- Client-server communication.
- Real-time message updates.
- Authentication and authorization.

Case Study and Conclusion

Recap and Final Thoughts

Key Takeaways:

- ASP.NET Core 8.0 is optimized for real-time applications with SignalR.
- Real-time apps offer a better user experience by providing instant feedback and dynamic content updates.
- Scalability and security are critical for deploying real-time apps in production environments.

Next Steps: Explore cloud-based scaling with Azure SignalR Service and build more complex real-time features.

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Thank You