🎓 Design Patterns in Blazor

Overview

In this lecture we'll explore **four core design patterns** used *inside* Blazor:

1. **Observer Pattern** — reactive UI updates
2. **Composite Pattern** — component structure
3. **Proxy Pattern** — Blazor Server communication
4. **Adapter Pattern** — .NET ↔️ JavaScript interop

Each of these is not only theoretical — it's built directly into Blazor's runtime.

1️⃣ Observer Pattern

The foundation of reactive UI

Idea:

An object (component) notifies its observers (renderer) when its state changes.

In Blazor:

- Each component observes its own state.
- When state changes → `StateHasChanged()` triggers a re-render.
- The renderer updates only the modified parts of the DOM.

Example

```
@page "/counter"

<h3>Counter: @count</h3>
<button @onclick="IncrementCount">Add</button>

@code {
    private int count = 0;

    void IncrementCount()
    {
        count++;
        StateHasChanged(); // Notify observers (renderer)
    }
}
```

Mapping:

- `Component` → Subject
- `Renderer` → Observer

Concept diagram

2️⃣ Composite Pattern

Building UI as a component tree

Idea:
Compose complex structures from smaller, reusable objects — all sharing the same interface.

In Blazor:

- Components can contain other components.
- Each implements the `IComponent` interface.
- The renderer treats all components uniformly.

Example

```
<!-- ParentComponent.razor -->
<h3>Student list</h3>

<StudentCard Name="Alice" />
<StudentCard Name="Bob" />
```

```
<!-- StudentCard.razor -->
<div class="card">
    <p>@Name</p>
</div>

@code {
    [Parameter] public string Name { get; set; }
}
```

Structure visualization

```
ParentComponent (Composite)
│
├── StudentCard (Leaf)
└── StudentCard (Leaf)
```

➡️ Result: Hierarchical, modular, and reusable UI.

3️⃣ Proxy Pattern

Blazor Server communication model

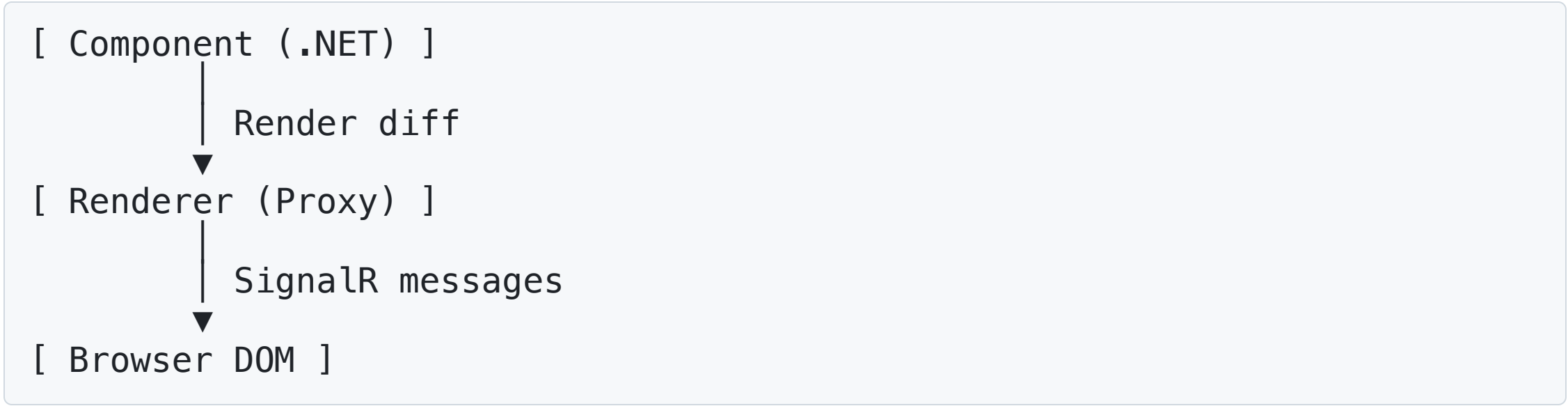
Idea:

A proxy acts as a substitute for a real object located elsewhere.

In Blazor Server:

- Components run **on the server**.
- The DOM exists **in the browser**.
- Communication happens through **SignalR**.
- The renderer proxies DOM operations.

Architecture diagram



➡️ The component "thinks" it’s manipulating a local DOM,
but the Proxy handles remote synchronization.

Why it matters

- Enables thin clients (only HTML & SignalR needed).
- Reduces browser-side code complexity.
- Demonstrates the **Proxy pattern** in a real-world distributed UI system.

4️⃣ Adapter Pattern

Bridging .NET and JavaScript

Idea:

An adapter translates one interface into another so they can work together.

In Blazor:

- `IJSRuntime` acts as an adapter between .NET and JavaScript.
- Developers can call JS functions from C# seamlessly.

Example

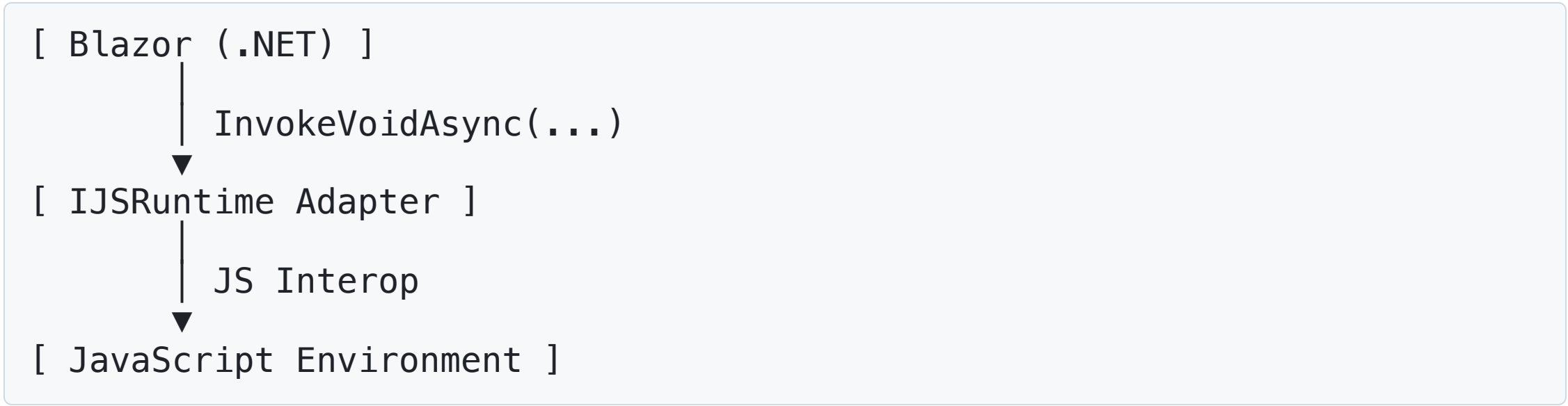
```
@page "/interop"
@inject IJSRuntime JS

<button @onclick="ShowAlert">Alert</button>

@code {
    private async Task ShowAlert()
    {
        await JS.InvokeVoidAsync("alert", "Hello from .NET!");
    }
}
```

Mapping:

- `IJSRuntime` → Adapter
- JavaScript runtime → Adaptee

➡️ Unified, type-safe access to JavaScript APIs.

Visual overview



🧭 Summary — Four pillars of Blazor design

Pattern	Where it appears	What it provides
Observer	`StateHasChanged`, rendering	Reactive UI updates
Composite	Component tree (`IComponent`)	Hierarchical, modular UI
Proxy	Blazor Server / SignalR	Remote rendering as if local
Adapter	`IJSRuntime`, JS interop	Seamless .NET ↔️ JS bridge

🧩 Key takeaway

Blazor's strength comes from combining classical design patterns with modern web technologies:

- **Observer** → Reactive rendering
- **Composite** → Component architecture
- **Proxy** → Transparent remote UI
- **Adapter** → Interop across runtimes

These patterns make Blazor not just a UI framework — but a *design pattern showcase in action*.