

# Managing Application State in Blazor

## What is Application State?

**Application state** is data that:

- Lives beyond a single component
- Is shared across multiple components/pages
- May need persistence (refresh, navigation, reconnect)
- Must be predictable and traceable

Examples:

- Logged-in user info
- Selected role (Admin/User)
- Cart items
- UI preferences (theme, language)

## 1. Having Bookmarkable State

### Concept

**Bookmarkable state** means:

- State is encoded in the **URL**
- Refreshing the page or sharing the link preserves the state

### Why it matters

- Browser refresh safety
- Shareable links
- Back/forward navigation works correctly

## Example: Bookmarkable Order Selection

*URL*

/orders/101

*Orders.razor*

```
@page "/orders/{OrderId:int}"
```

```
<h3>Order Details</h3>
```

```
<p>Order Id: @OrderId</p>
```

```
@code {  
    [Parameter]  
    public int OrderId { get; set; }  
}
```

## Navigation

```
<button @onclick="() => Nav.NavigateTo("/orders/101")">  
    View Order 101  
</button>
```

```
@inject NavigationManager Nav
```

## 2. Implementing an In-Memory State Container

### Concept

An **in-memory state container**:

- Is a plain C# class
- Holds shared state
- Raises change notifications

### Example: ApplicationState Container

```
public class AppState  
{  
    public string? UserName { get; private set; }  
    public string? Role { get; private set; }  
  
    public event Action? OnChange;  
  
    public void SetUser(string userName, string role)  
    {  
        UserName = userName;  
        Role = role;  
        NotifyStateChanged();  
    }  
  
    public void Clear()  
    {  
        UserName = null;  
        Role = null;  
        NotifyStateChanged();  
    }  
  
    private void NotifyStateChanged()
```

```
        => OnChange?.Invoke();  
    }
```

## Why this works

- Centralized state
- Explicit mutation methods
- Controlled updates

## 3. Injecting Application State as a Service

### Register in DI

#### *Program.cs*

```
builder.Services.AddScoped<AppState>();
```

Scoped is correct for Blazor Server and Blazor Web App interactive sessions.

---

### Consume in Component

```
@inject AppState AppState
```

```
<h3>Welcome @AppState.UserName</h3>
```

#### **Subscribing to State Changes**

```
@implements IDisposable
```

```
@code {
```

```
    protected override void OnInitialized()
```

```
    {
```

```
        AppState.OnChange += StateHasChanged;
```

```
    }
```

```
    public void Dispose()
```

```
    {
```

```
        AppState.OnChange -= StateHasChanged;
```

```
    }
```

```
}
```

### Why this is mandatory

Without subscription:

- UI will not re-render when state changes

## 4. Invoking State Changes from Anywhere

### Concept

Because the state container is DI-based:

- Any component
- Any service
- Any event handler

can update state.

---

### Example: Login Component

```
@inject AppState AppState
@inject NavigationManager Nav

<button @onclick="Login">Login</button>

@code {
    void Login()
    {
        AppState.SetUser("San", "Admin");
        Nav.NavigateTo("/dashboard");
    }
}
```

### Example: Logout from NavMenu

```
<button @onclick="Logout">Logout</button>

@code {
    void Logout()
    {
        AppState.Clear();
    }
}
```

---

## 5. Persisting State

### Problem

In-memory state is lost on:

- Browser refresh
- Circuit reconnect
- App restart

### Solution Options

Storage	Use Case
ProtectedSessionStorage	Per session
ProtectedLocalStorage	Persistent
Database	Server-side

### Example: Persisting to ProtectedSessionStorage

#### *Register*

```
builder.Services.AddScoped<ProtectedSessionStorage>();
```

## 6. Resolving Persisted State

### Restore on App Start

#### *App.razor or MainLayout.razor*

```
@inject AppState AppState
@inject ProtectedSessionStorage Storage

@code {
    protected override async Task OnInitializedAsync()
    {
        var saved = await Storage.GetAsync<AppState>("app_state");
        if (saved.Success && saved.Value != null)
        {
            AppState.SetUser(saved.Value.UserName!, saved.Value.Role!);
        }
    }
}
```

## Best Practice

- Restore state **before rendering secured UI**
  - Avoid async logic inside constructors
- 

## 7. Sharing State Across Interactive Render Mode Boundaries

### Context (.NET 8 Blazor Web App)

Render modes:

- `InteractiveServer`
- `InteractiveWebAssembly`
- `Static`

## Problem

**State containers do NOT automatically flow across render mode boundaries**

## Correct Patterns

### *Pattern 1: URL-based State*

Use query string or route params

```
/dashboard?role=admin
```

---

### *Pattern 2: Browser Storage (Recommended)*

Shared across render modes:

- `LocalStorage`
- `SessionStorage`

```
await storage.SetAsync("role", "Admin");
```

---

### *Pattern 3: Server Persistence*

Use database or cache

Flow:

- WASM → API → Database
- Server → Reload state

---

## Rule of Thumb

Scenario	Solution
Same render mode	DI State Container
Across render modes	URL or Browser Storage
Across sessions	Database

## Summary Architecture

pgsql

UI Components



AppState (**In**-Memory)



Persistence Layer

(**S**ession / **L**ocal / DB)



Restore **on** Startup