

Important Blazor enhancements introduced across .NET 9 and .NET 10

Blazor .NET 9 / .NET 10 – Advanced New Features Explained

Blazor in .NET 9 and .NET 10 focuses less on “new toys” and more on **performance, predictability, scalability, and fine-grained control**. These changes matter most in **enterprise-scale apps, hybrid rendering scenarios, and production deployments**.

1. Static Asset Delivery Optimization

Problem Before .NET 9

Blazor apps (especially WASM and hybrid apps) often suffered from:

- Large `_framework` payloads
- Redundant static file requests
- Inefficient caching
- Slow cold starts (especially over mobile networks)

Static assets were treated *mostly the same* regardless of **rendering mode** or **usage context**.

What Changed in .NET 9/10

.NET 9+ introduces **intelligent static asset handling** tightly integrated with:

- Rendering mode
- Build output
- HTTP caching semantics
- Lazy loading strategies

Key Improvements

1. Smarter Asset Fingerprinting

- Static assets are now **content-hashed more aggressively**
- Enables **long-term immutable caching**

- Reduces unnecessary re-downloads after deployments

Cache-Control: public, max-age=31536000, immutable

2. Rendering-Mode-Aware Asset Loading

- Assets required **only for WebAssembly** are not loaded for Server rendering
- Interactive assets are delayed until needed

Example:

- Static SSR page → **no WASM runtime loaded**
- Interactive WASM page → runtime fetched **only when required**

3. Reduced _framework Chattiness

- Fewer small HTTP requests
 - Improved bundling and compression
 - Better HTTP/2 and HTTP/3 utilization
-

Why This Matters (Experienced Perspective)

- Faster **Time to First Paint (TTFP)**
 - Lower bandwidth usage
 - Better Lighthouse scores
 - Significant improvement for **PWA + mobile users**
-

2. Router Improvements

Problem Before .NET 9

The Blazor router:

- Was functional but rigid
 - Had limited extensibility
 - Required hacks for advanced scenarios (auth-aware routing, layouts per route, etc.)
-

What's New

.NET 9+ introduces a **more composable and extensible routing pipeline**.

Key Enhancements

1. Improved Route Matching

- More predictable matching precedence
- Better handling of overlapping routes
- Cleaner fallback behavior

2. Enhanced NotFound Handling

You can now build **first-class NotFound experiences** without hacks:

```
<Router AppAssembly="@typeof(App).Assembly">
  <Found Context="routeData">
    <RouteView RouteData="@routeData" />
  </Found>
  <NotFound>
    <NotAuthorized />
  </NotFound>
</Router>
```

3. Route-Aware Layout Selection

- Layouts can now vary more naturally based on route
- Less conditional logic inside components

Why This Matters

- Cleaner separation of **navigation, authorization, and UI composition**
- Easier to build **large modular apps**
- Router logic becomes **testable and maintainable**

3. Authentication State Serialization

The Core Problem (Pre-.NET 9)

In **Blazor Server + SSR + WASM** hybrid apps:

- Authentication state had to be recomputed
- Identity information was often re-fetched
- Caused **double auth checks** and UI flicker

What Is Authentication State Serialization?

Authentication state can now be:

- **Captured during server rendering**
- **Serialized**
- **Transferred to the client**
- **Rehydrated without re-authentication**

How It Works Conceptually

1. Server renders the page
2. User is authenticated
3. AuthenticationState is serialized
4. Client starts
5. Client **reuses the same auth state**

No extra round-trips.

Example Scenario

Before:

SSR → Client loads → AuthProvider re-evaluates → UI flicker

After:

SSR → Client loads → Auth state reused → seamless UI

Why This Is Huge

- Eliminates login flicker
 - Improves perceived performance
 - Critical for **enterprise SSO, JWT, cookie-based auth**
 - Makes Blazor feel like a **true SPA + SSR hybrid**
-

4. RendererInfo API

What Is the RendererInfo API?

The RendererInfo API allows a component to **know exactly how it is being rendered**:

- Server
 - WebAssembly
 - InteractiveServer
 - InteractiveWebAssembly
 - Static SSR
-

Why This Exists

Previously, developers had to:

- Infer rendering mode indirectly
- Use environment checks
- Duplicate components

Now rendering context is **first-class information**.

Conceptual Example

@inject RendererInfo RendererInfo

@if (RendererInfo.IsInteractive)

```
{  
    <button @onclick="DoClientThing">Click</button>  
}
```

else

```
{  
    <p>Static view</p>  
}
```

Practical Use Cases

- Disable JS interop in SSR
- Optimize components per renderer
- Conditional logic without hacks
- Cleaner shared component libraries

Why Experienced Developers Care

- Enables **renderer-aware component design**
- Reduces duplication
- Improves reliability in hybrid apps

5. Custom Rendering Modes (Per Page)

Before .NET 9

Rendering mode was mostly:

- App-wide

- Layout-wide
 - Hard to mix cleanly
-

What Changed

You can now specify **rendering mode per page or component**.

Example

@rendermode InteractiveWebAssembly

or

@rendermode InteractiveServer

Mixed Rendering Example

Page	Rendering Mode
Home	Static SSR
Dashboard	InteractiveServer
Reports	InteractiveWebAssembly
Admin	Server

Why This Is a Big Deal

- Fine-grained performance control
 - Reduce server load
 - Optimize expensive pages
 - Progressive interactivity
 - Ideal for **large enterprise portals**
-

Architectural Impact Summary

Feature	Impact
Static Asset Optimization	Faster load, lower bandwidth
Router Improvements	Cleaner navigation & layouts
Auth State Serialization	Seamless auth UX
RendererInfo API	Renderer-aware components
Custom Rendering Modes	Per-page performance tuning

Final Instructor Insight

Blazor .NET 9/10 marks a **maturity phase**:

- Less abstraction leakage
- More explicit control
- Better performance defaults
- True **SSR + SPA hybrid parity**