



# Blazor State Management

Managing User Data Across Client and Server

Tim Purdum

DevUp Conf

August, 2025

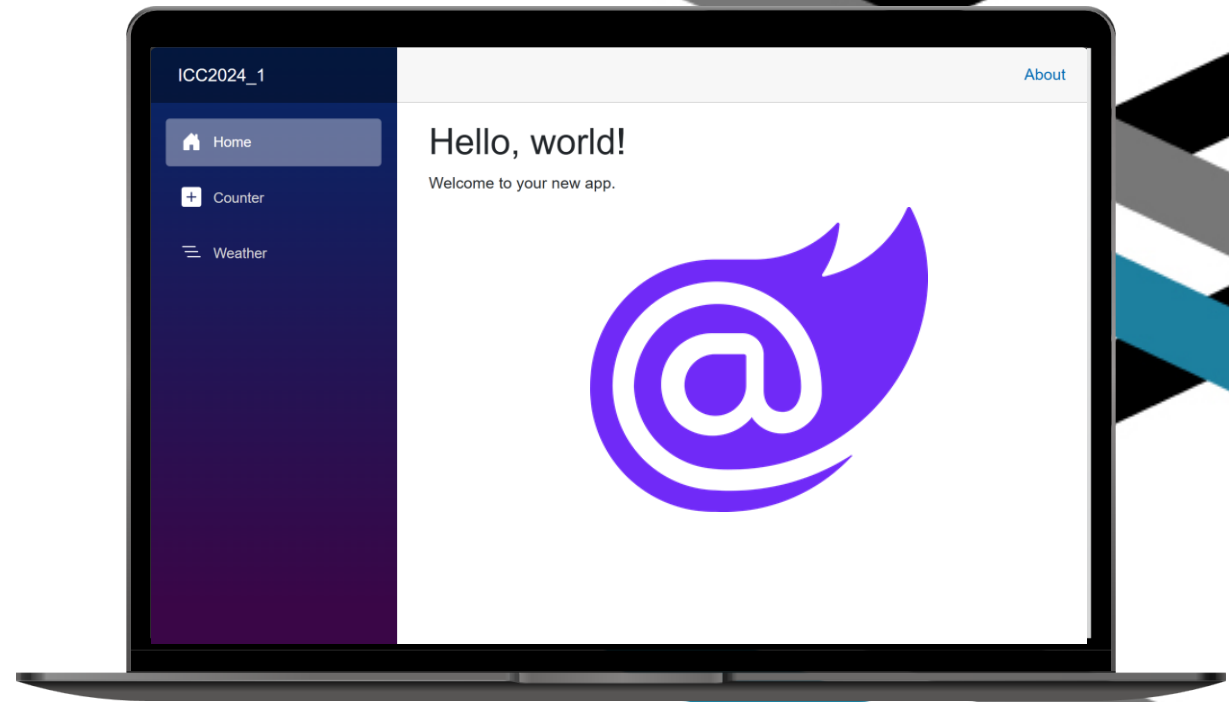


# Goals of the Session

- Identify types of state management in Blazor and the tools and patterns used
- Learn about how the Blazor rendering modes and render cycles impact state management
- Identify larger architectural patterns and practical examples for managing state in a Blazor application

# What is Blazor?

- Modern full-stack web framework
- Built on Asp.NET Core and Modern .NET
- Released with .NET Core 3.1 in 2018
- Component-based reactive framework
- Static and dynamic Server-Side rendering
- Client WebAssembly SPA applications or individual components
- High productivity with a single unifying language and framework



# dev up **Bl@zing Shipments**

As we look at this web app, consider the following questions:

- Where are the pages being rendered?
- How does it know what data to load?
- Are the pages comprised of a single component, or many?
- How does the site respond to user interaction?
- If we needed to store data, where would we store it?





# What is State Management?

- “State management refers to the management of the state of one or more user interface controls such as text fields, submit buttons, radio buttons, etc. in a graphical user interface.”
  - *from Wikipedia (based on [redux.js.org](https://redux.js.org))*





# Types of State in Web Development

- **Component State**
- **Application State**
- **User/Session State**
- **Persistent State**





# Component State

- Stored in component fields/properties or a model object
- Bound to HTML input and display elements
- Unsaved changes are lost on navigation/refresh

```
<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary"
    @onclick="IncrementCount">Click me</button>
@code {
    private int currentCount = 0;
    private void IncrementCount() => currentCount++;
}
```



# Component State

- /

```
<input type="text" @bind="fieldOrProp" />
```

- fires with the event

- Change the event with `@bind:event="oninput"`

- Add a change handler method with `@bind:after="HandlerMethod"`

- C w /

```
<TestComponent @bind-ParameterName="fieldOrProp" />
```



# Application State

- State shared across components using
  - **Parameters**
  - **CascadingValues**
  - **EventCallbacks**
  - **Service Classes**



# Application State: Parameters

- C# public properties with [Parameter] attribute on a child component

```
MapView.razor

[Parameter]
public double? Latitude { get; set; }

[Parameter]
public double? Longitude { get; set; }
```

- In consuming (parent) class markup, parameters display like HTML attributes with capital letters

```
<MapView Latitude="@shipment.Latitude" Longitude="@shipment.Longitude">
  <Map>
    <Basemap>
      <BasemapStyle Name="BasemapStyleName.ArcgisStreets"/>
    </Basemap>
  </Map>
</MapView>
```



# Application State: Cascading Values

- Wrap child components with markup tags

```
<CascadingValue Value="@User" Name="CurrentUser">
  <ProfileSelector />
</CascadingValue>
```

- !

/ t

```
[CascadingParameter(Name="CurrentUser")]
public ApplicationUser? CurrentUser { get; set; }
```

# dev up Application State: EventCallbacks

- A type of Parameter
- Async-supporting Event triggers

```
[Parameter]
public EventCallback<LayerViewCreateEvent> OnLayerViewCreate { get; set; }
```

- Bind to a parent component method instead of field or property

```
<MapView OnLayerViewCreate="OnLayerViewCreate">
  <Map>
    <FeatureLayer OutFields="@(["*"])">
      <PortalItem PortalItemId="234d2e3f6f554e0e84757662469c26d3" />
    </FeatureLayer>
  </Map>
</Extent>
</ MapView>
```

```
private async Task OnLayerViewCreate(LayerViewCreateEvent createEvent)
{
    if (createEvent.Layer is FeatureLayer)
    {
        // query the feature service
    }
}
```



# Application State: Service Classes

- Any C# Class can be injected via Property Injection

- In Razor Markup

```
@page "/order"
```

```
@inject StateManagementService StateManagementService
```

- Or in C#

```
@code {  
    [Inject]  
    private StateManagementService? StateManager { get; set; }  
}
```

- !
  - {
  - Ü
- { a t /
- /
- b9Ç 9 l



# User/Session State

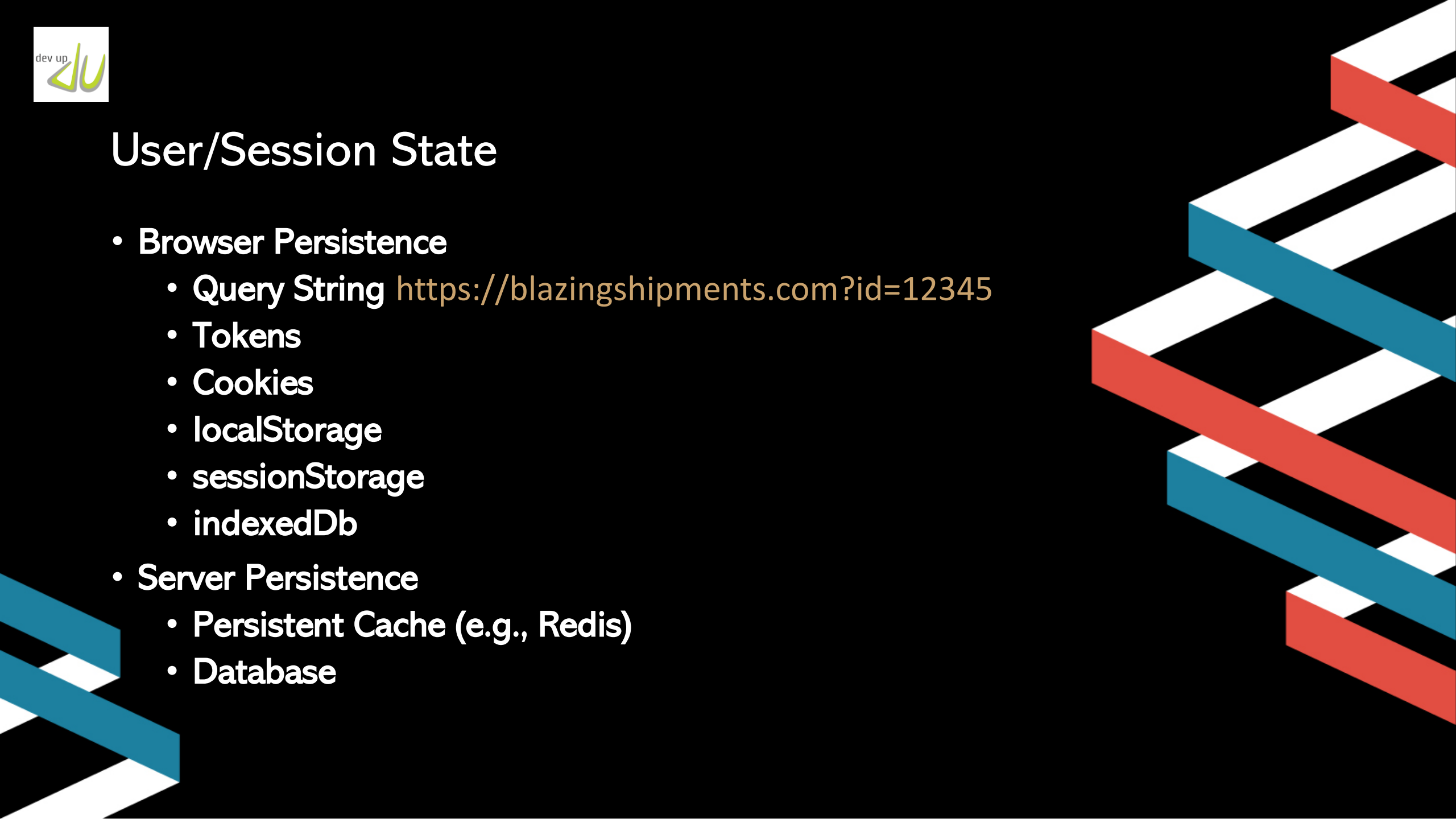
- Authentication
- Authorization
- Profile
- Records
- Work Progress





# User/Session State

- **Browser Persistence**
  - **Query String** <https://blazingshipments.com?id=12345>
  - **Tokens**
  - **Cookies**
  - **localStorage**
  - **sessionStorage**
  - **indexedDb**
- **Server Persistence**
  - **Persistent Cache (e.g., Redis)**
  - **Database**





# Persistent State: Browser Storage

- **localStorage**
  - persists when tab/browser is closed, across multiple tabs
- **sessionStorage**
  - isolates data between tabs to prevent issues, data also is lost when tab is closed
- **IndexedDb**
  - Object-store structured database
  - Create an object store with a key path (aka ID) or a key generator
  - Also supports indexes
  - Transaction-scoped access: add, put (update), get, delete
- All require JavaScript or NuGet JS wrappers to interact.
- Available in “Interactive Render Modes”





# Persistent State: Server Storage

- **MemoryCache**
- **Redis cache**
- **HybridCache**
- **Database**
- **Only available from “Interactive Server” or via web API calls.**

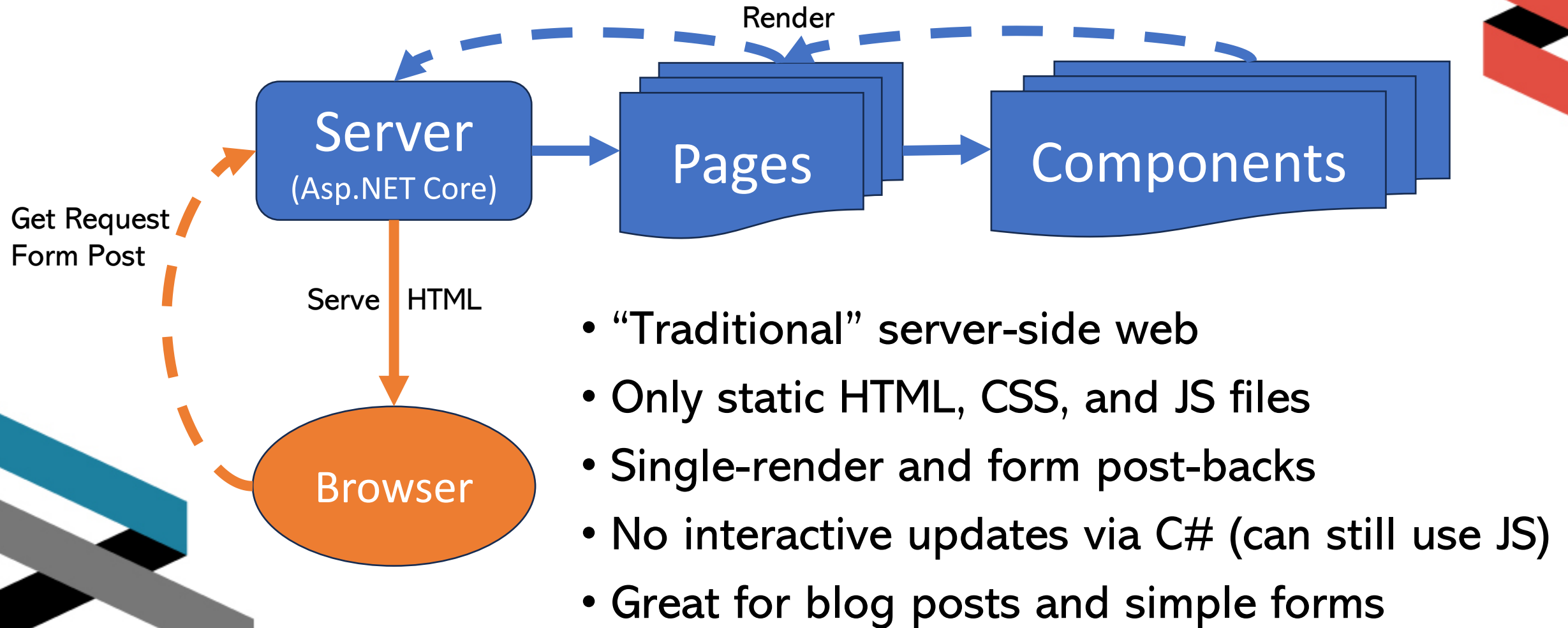


# Blazor Render Modes

- { { a
- L { a
- L í ! a
- L ! a
- . |

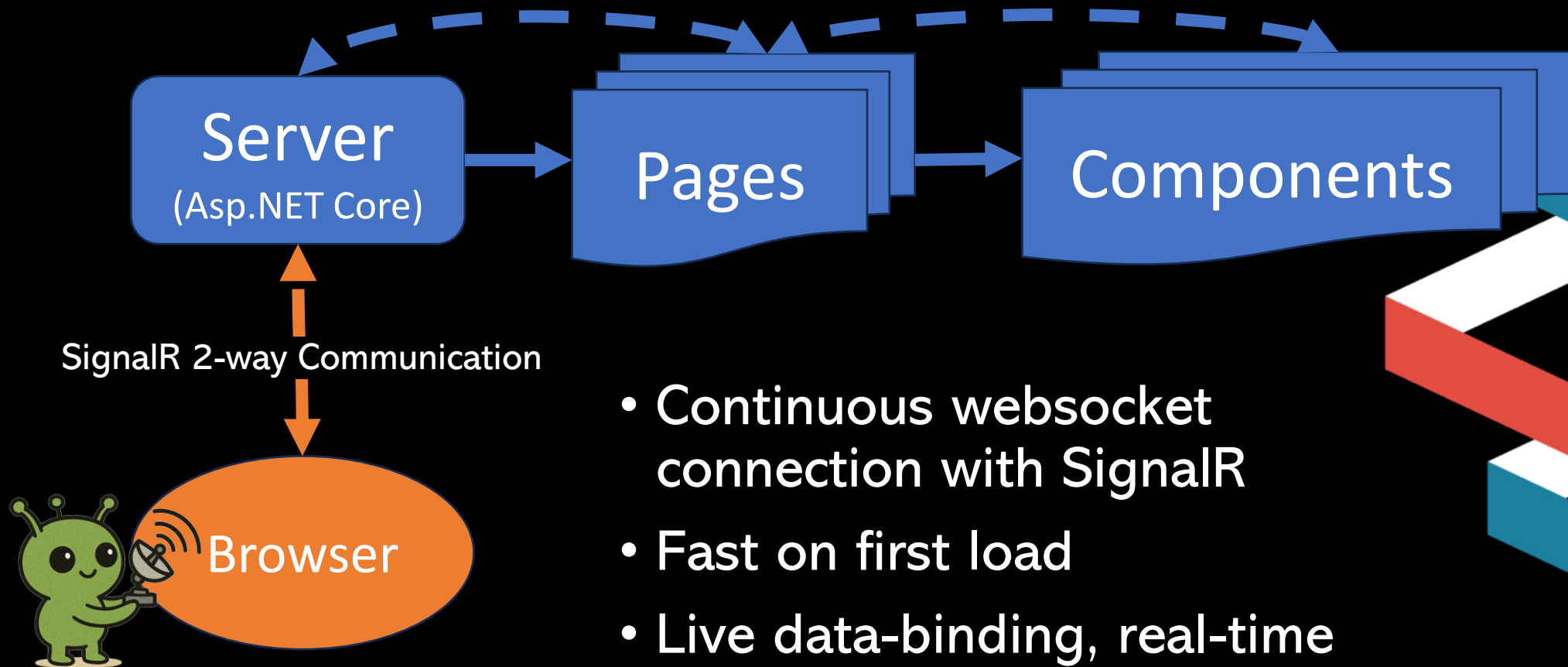


# Blazor Render Modes: Static Server





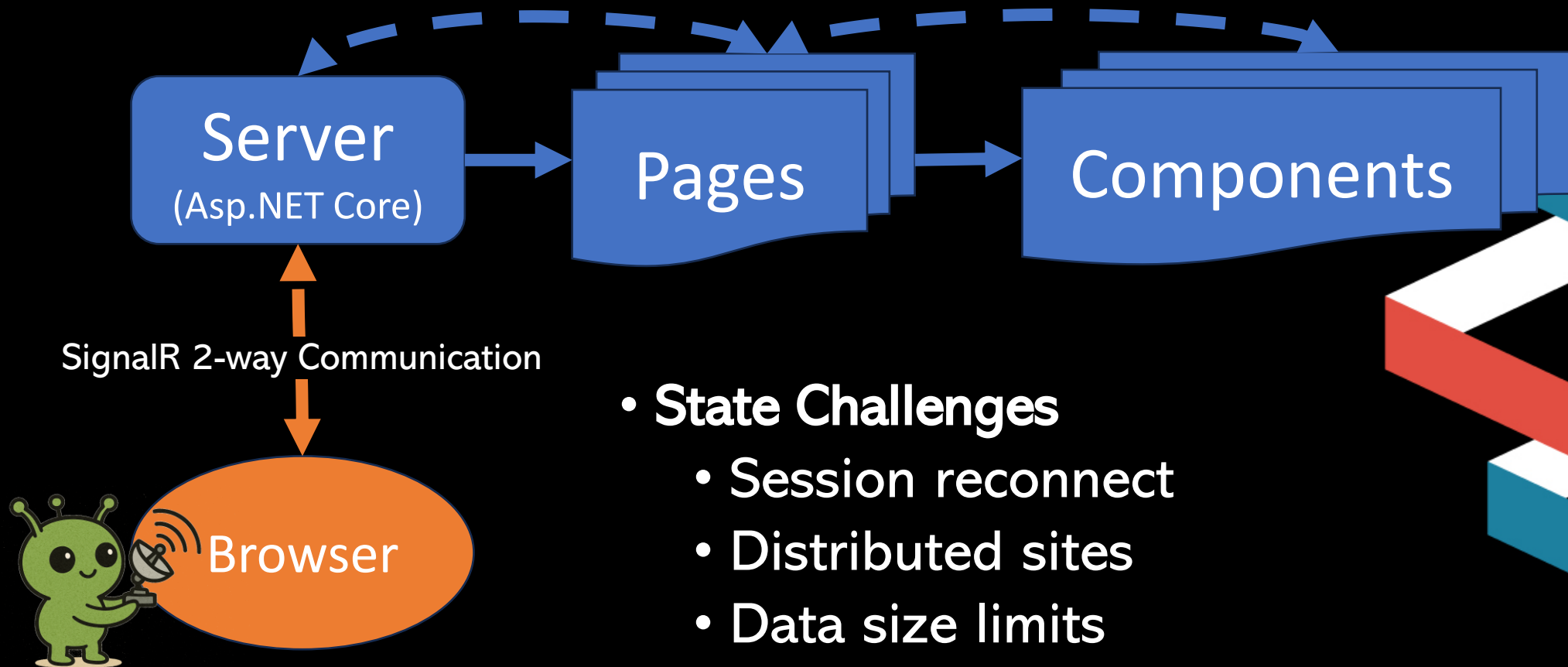
# Blazor Render Modes: Interactive Server



- Continuous websocket connection with SignalR
- Fast on first load
- Live data-binding, real-time updates, JavaScript interop
- Direct access to server data store



# Blazor Render Modes: Interactive Server

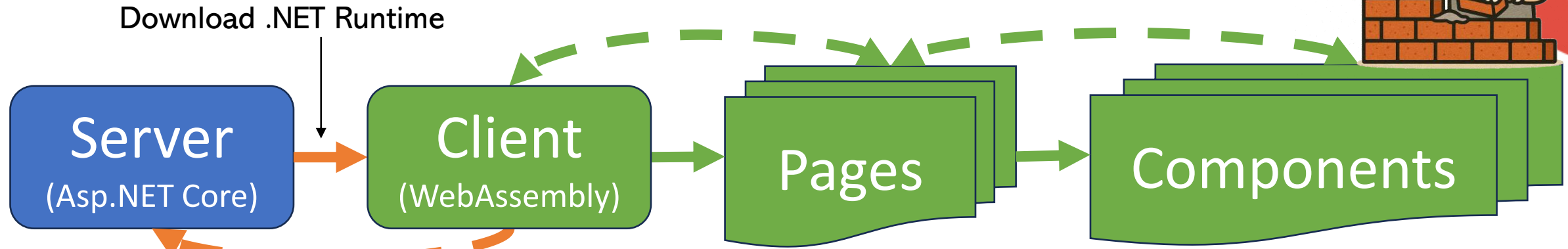


- **State Challenges**

- Session reconnect
- Distributed sites
- Data size limits
- Network lag



# Blazor Render Modes: Interactive WebAssembly



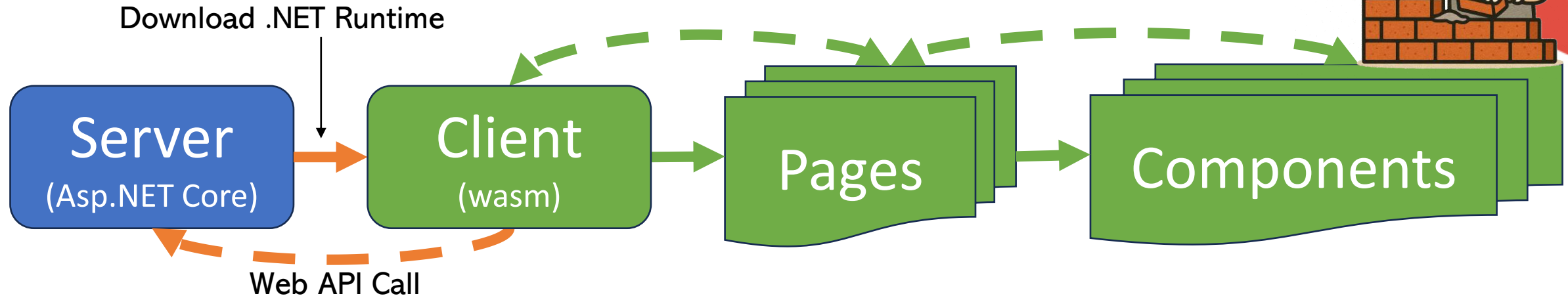
HttpClient Web API Calls  
SignalR, gRPC

- Runs in the client browser
- Large/slow first load
- Live data-binding, real-time updates, JavaScript interop
- HttpClient calls to communicate with server web API

- Single-threaded
- Fast interactions after load
- Closest in approach to most JS SPA frameworks



# Blazor Render Modes: Interactive WebAssembly



- **State Challenges**
  - Data sync
  - New data object IDs before syncing
    - (Use Guid)



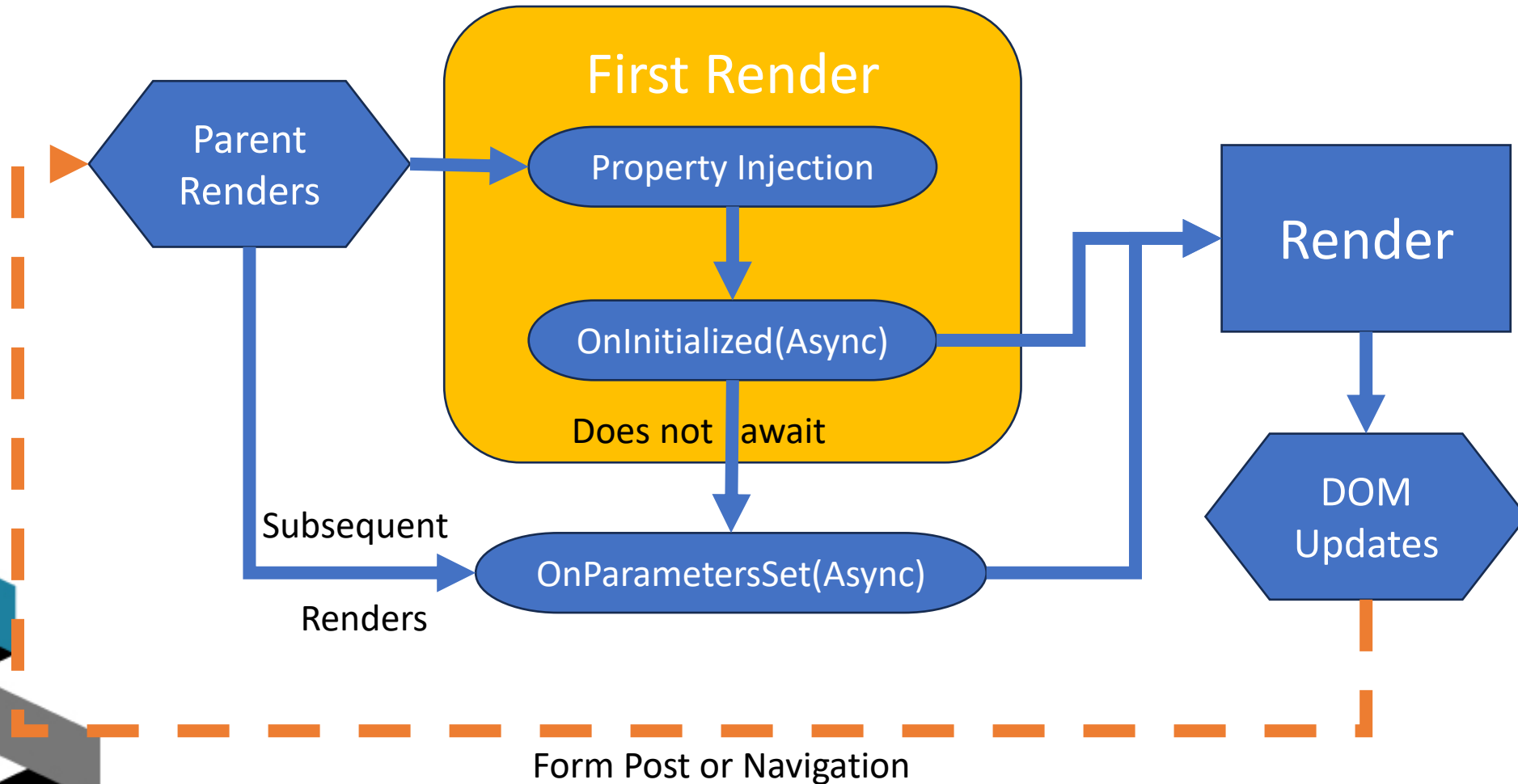
# Blazor Render Modes: Interactive Auto



- On first load, runs Interactive Server (SignalR)
- Downloads .NET runtime and client Wasm code in the background
- On next load, switches to running from WebAssembly
- “Best of both worlds”
  - Fast start on first load (server)
  - More responsive and robust interactions (client)
- Requires flexible data handling/abstraction to handle both client and server modes

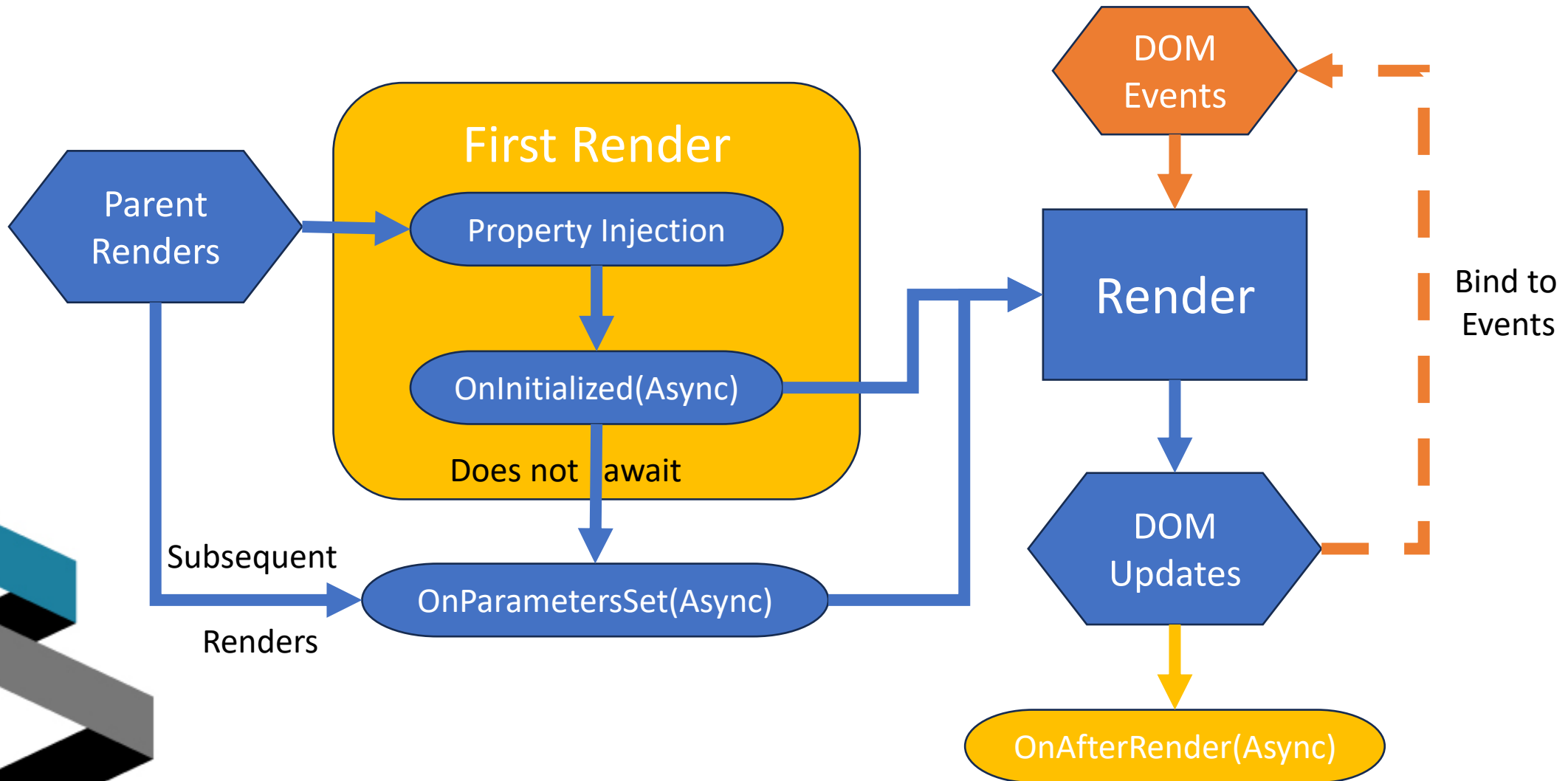


# Razor Component Lifecycle: Static Server Mode



*State set in `OnInitialized` and `OnParametersSet` should be Idempotent*

# Razor Component Lifecycle: Interactive Modes



***Don't set state that will cause a render cycle in OnAfterRender!***



# Architectural Patterns for State Management

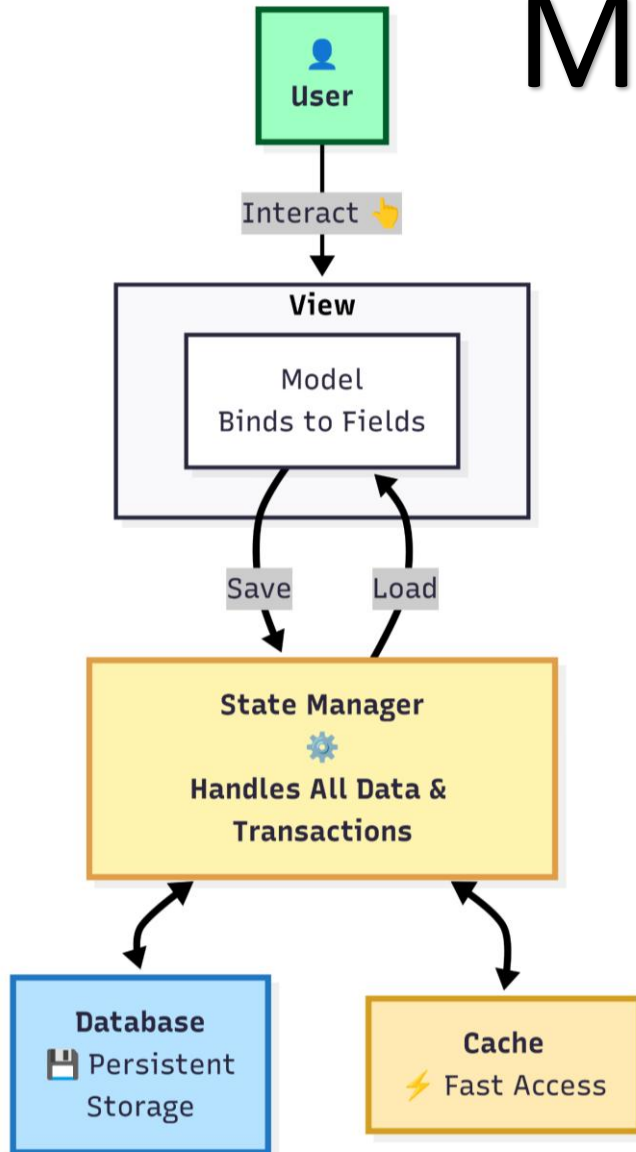
- {
  - w C w a ëÜ
  - ó! a [ C a ëëa
  - ! b9Ç/ a ë/ a ë/
- .



# Architectural Patterns for State Management

- **Goals for Blazor State Management**
  - **Flexible components that will work in both Interactive Server and Interactive WebAssembly modes**
  - **Reduced boilerplate logic like pass-through methods**
    - *(e.g., `clientComponent => clientService => webApi => webService => dataRepository`)*
  - **Consistent patterns for communication between components**
  - **Abstract away communication from WebAssembly client to Server**
  - **Keep pages and components lightweight and easy to read**
  - **Allow generic implementations for simple use cases**

## MVSM™



- **Model**
- **View**
- **State Manager**
- Model and View designed to work together with two-way binding
- Model can live in either the View or the State Manager class
- State Manager is responsible for abstracting transport and any data transformation

SERVER

API

SignalR Client

RazorComponentPage  
StateComponentBase

Fields & Properties

PageModel  
: StateRecord

RedoStack

UndoStack

Methods

Redo

Undo

Delete

Save

Load

Track Changes

ServerStateManager

Load

Delete

Save

Data  
Repository

BROWSER

WebAssembly  
Client

RazorComponentPage  
StateComponentBase

Fields & Properties

PageModel  
: StateRecord

RedoStack

UndoStack

Methods

Redo

Undo

Save

Load

Track Changes

Delete

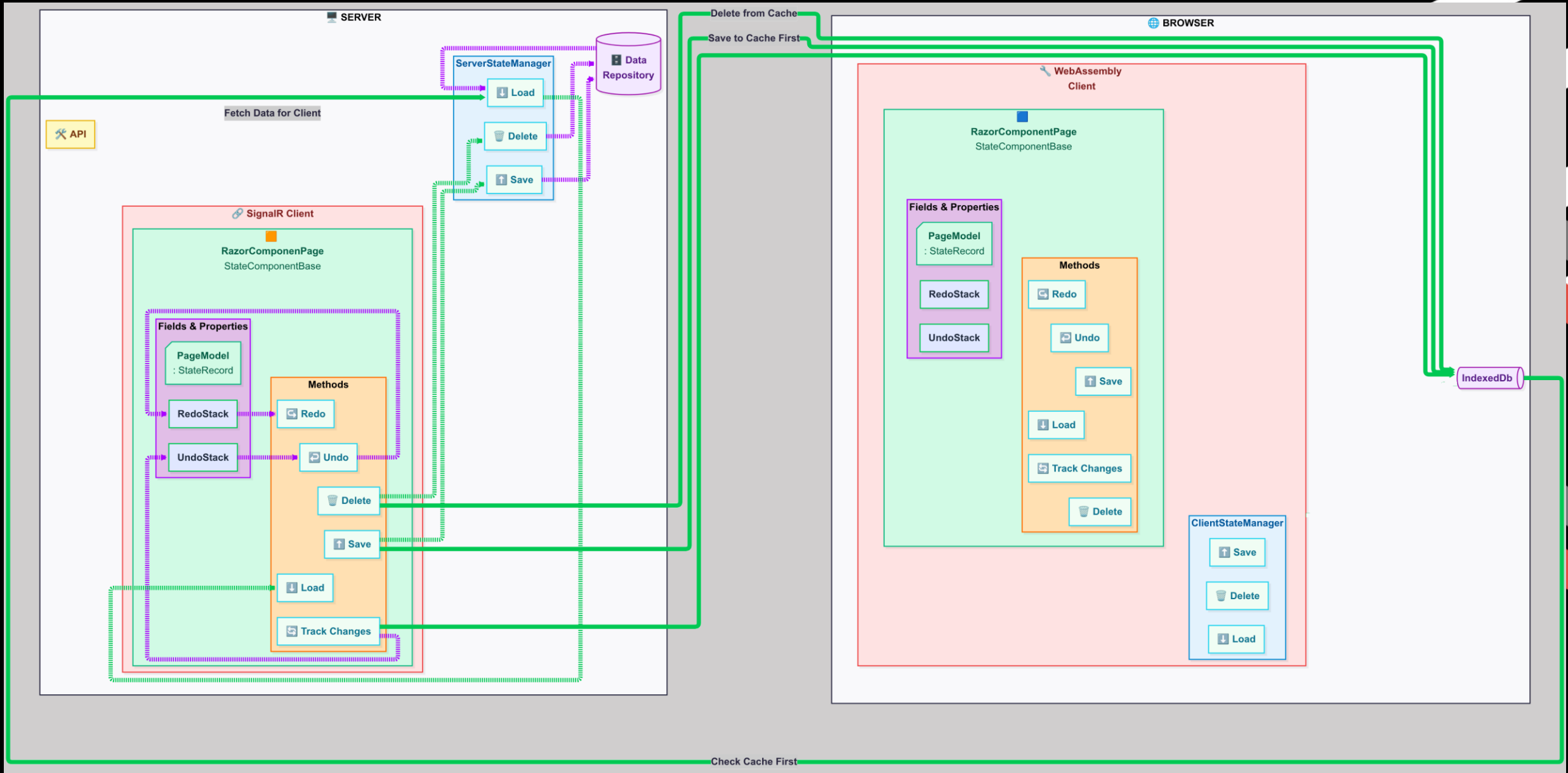
ClientStateManager

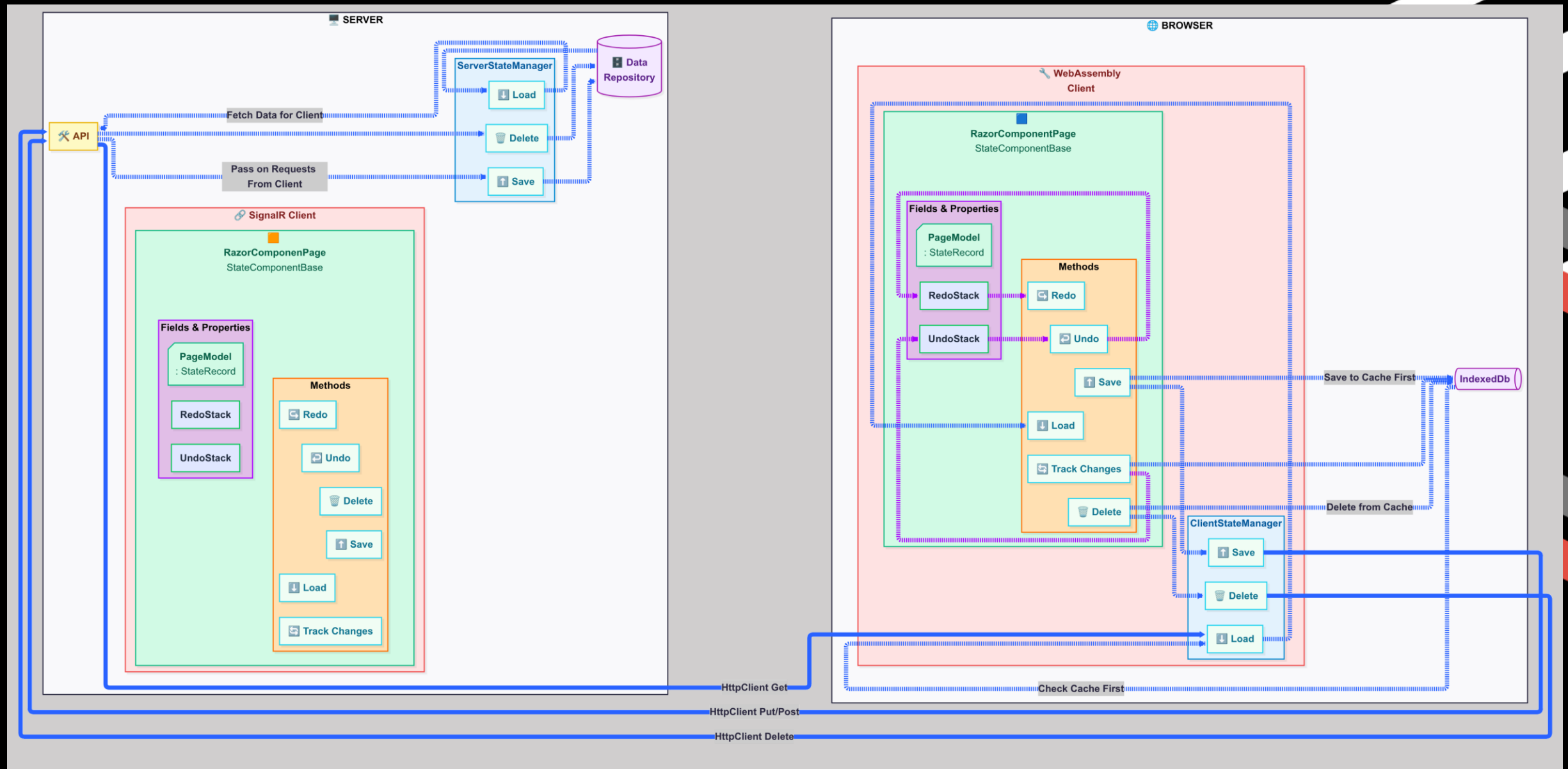
Save

Delete

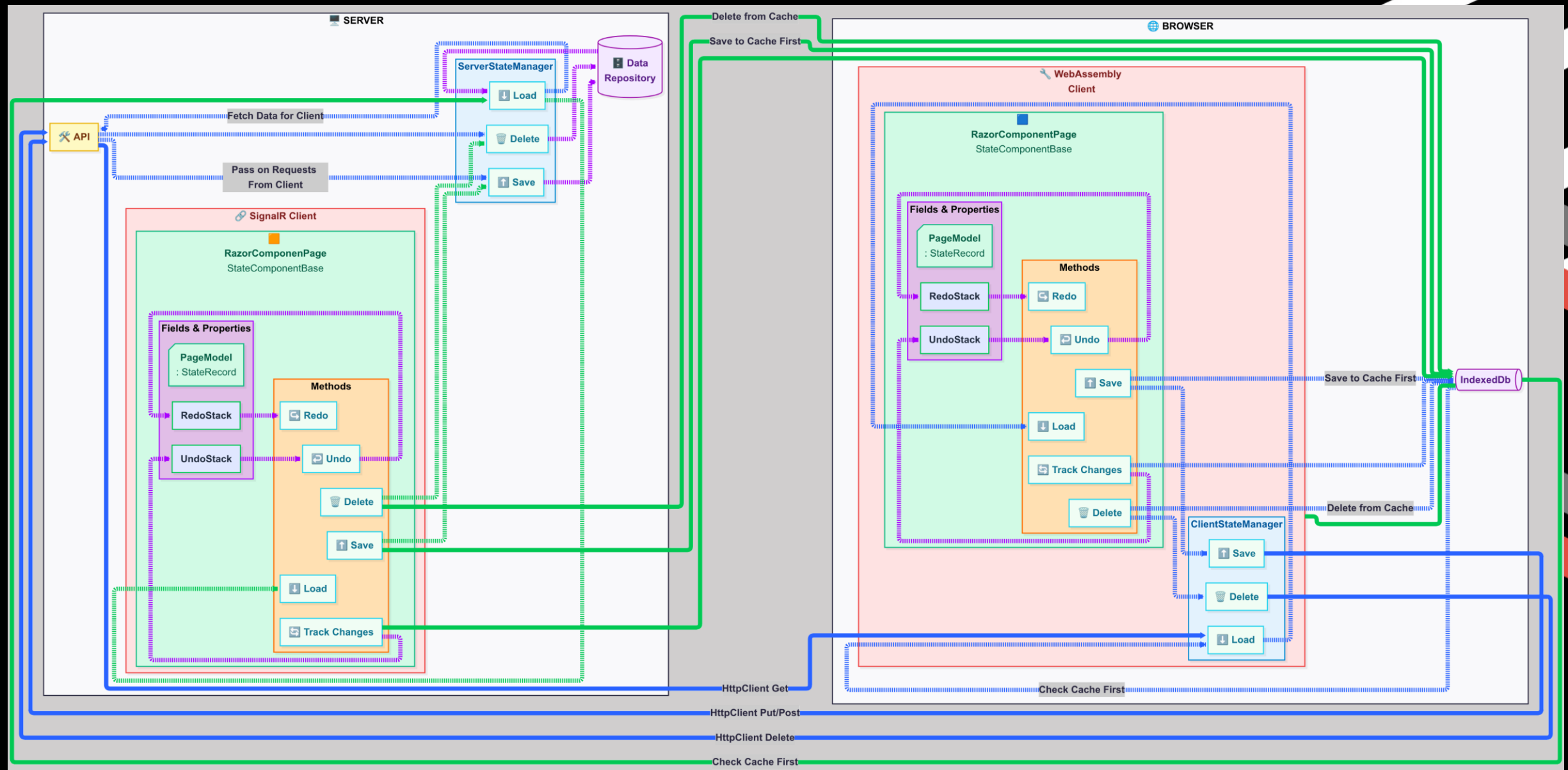
Load

IndexedDb











# Review



## Core Concepts of State Management

- **Definition:** State management involves tracking the dynamic data of a user interface—across components, sessions, and storage layers.
- **Types of State:**
  - *Component State:* Temporary, lost on refresh or navigation.
  - *Application State:* Shared across components using cascading values, DI services, etc.
  - *User/Session State:* Stored in browser memory (e.g., localStorage, sessionStorage, indexedDb), usually not synced with the server.
  - *Persistent State:* Long-term data stored in a database or API.



# Review

## Blazor Render Modes & Their State Implications

- **Static Server Mode:**
  - Simple form submission and HTML rendering.
  - Limited interactivity and no real-time state updates.
  - Persistence tools: cookies, tokens, query strings.
- **Interactive Server Mode:**
  - Real-time two-way binding using SignalR.
  - Enables in-memory server-side tracking and real-time updates.
  - Challenges: reconnection handling, distributed server sync.
- **Interactive WebAssembly Mode:**
  - Fully client-side execution.
  - Rich interactivity with flexible state control
  - .Risks of state desynchronization and ID conflicts for new data.
- **Interactive Auto Mode:**
  - Hybrid approach: server-rendered first load, client-side on reload.
  - Balances fast startup with responsive interactivity.



# Review

## ✦ Patterns for Binding & Application State Sharing

- *Binding:*
  - @bind, @bind:event, @bind:get/set, and @bind:after allow seamless two-way data binding in Razor.
- *Component Communication:*
  - Parameters, CascadingValues, EventCallbacks, and DI Services are used to maintain shared state and coordination.

## 📁 Browser Storage Techniques

- *localStorage and sessionStorage:*
  - Simple key-value stores for persistence.
- *IndexedDb:*
  - Structured object store with indexing and transaction support. Can be wrapped with JS + C# logic or NuGet packages.



# Review

## 🧠 Architectural Patterns for Blazor

- MVU: Immutable, Redux-style, but not ideal for Blazor's reactive capabilities.
- MVVM: Familiar in .NET but verbose; Blazor doesn't require `INotifyPropertyChanged`.
- MVC: Suited for non-interactive, server-rendered apps—less effective in Blazor.

## 🧩 MVSM™ – A Blazor-Centric Pattern

- **Model-View-State Manager:**
  - Two-way binding between View and Model
  - .State Manager handles all data transport, persistence, and API abstraction.
  - Designed for extensibility using generics, reflection, and browser storage.

<https://nation-finder.geoblazor.com>

Find the country based on its outline



# Thank you to our Sponsors!



Thank You!



dymaptic

Notes & Links @  
<https://timpurdum.dev>

