

welche projekte würdest du hier noch hinzufügen? Ich möchte gerne clean archtitecture verwenden. Die Blazor App soll die Api verwenden.

Sehr gutes Setup bisher 🎉

Dein aktuelles Solution-Layout zeigt eine saubere Trennung von API (BlazorERP.Api) und Web-Frontend (BlazorERP.Web + BlazorERP.Components).

Wenn du **Clean Architecture** umsetzen möchtest, würde ich folgende zusätzlichen Projekte empfehlen, um die Schichten sauber zu trennen:

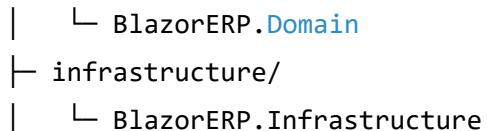
✳️ Übersicht Clean Architecture Layer

Eine typische Clean-Architecture-Struktur für Blazor + API (.NET 9, Dapper) sieht so aus:

pgsql

Code kopieren

```
src/
  |- api/
  |  |- BlazorERP.Api
  |- web/
  |  |- BlazorERP.Web
  |  |- BlazorERP.Components
  |- application/
  |  |- BlazorERP.Application
  |- domain/
```



1. BlazorERP.Domain (Core)

Zweck:

Definiert die **Kernlogik** und **Business Entities**, unabhängig von Frameworks oder Datenbanken.

Inhalt:

- Entities (z. B. Customer , Order , Product)
- Value Objects
- Enums
- Domain Events
- Domain Exceptions
- Interfaces (z. B. IUnitOfWork , IRepository<T>)

Abhängigkeiten:

 keine anderen Projekte

2. BlazorERP.Application

Zweck:

Enthält die **Use Cases** (z. B. Commands & Queries), die auf dem Domain-Modell operieren.

Inhalt:

- Application Services
- CQRS (MediatR, falls gewünscht)
- DTOs / ViewModels
- Interfaces zu externen Services (z. B. IEmailService , IUserContext)
- Validation (z. B. mit FluentValidation)

Abhängigkeiten:

 BlazorERP.Domain



3. BlazorERP.Infrastructure

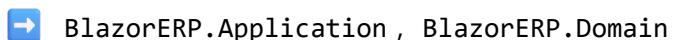
Zweck:

Implementiert alles, was mit Technik & Datenzugriff zu tun hat.

Inhalt:

- Dapper Repositories (oder EF Core, wenn du willst)
- Implementierungen der Interfaces aus Application
- Konfiguration von Logging, Mail, Auth usw.
- Migrationsscripts (falls EF Core)
- FileStorage, Cache, BackgroundJobs

Abhängigkeiten:



BlazorERP.Application , BlazorERP.Domain



4. BlazorERP.Api

Zweck:

Die REST- oder Minimal-API, die als Backend für die Blazor-App dient.

Inhalt:

- Controllers oder MinimalAPI-Endpoints
- Swagger/OpenAPI
- Dependency Injection Setup
- Middleware
- Auth & Authorization
- App-Settings

Abhängigkeiten:



BlazorERP.Application , BlazorERP.Infrastructure , BlazorERP.Domain



5. BlazorERP.Web & BlazorERP.Components

Zweck:

Blazor WebAssembly oder Blazor Server UI-Layer.

Die App ruft die API auf (über HttpClient oder gRPC).

Trennung:

- BlazorERP.Web : Haupt-Blazor-App (Routing, Layouts, Auth)
- BlazorERP.Components : Wiederverwendbare UI-Komponenten, Razor-Class-Library

Abhängigkeiten:

- BlazorERP.Components

Bonus: Optionales Shared-Projekt

BlazorERP.Shared

(oder BlazorERP.Contracts)

Zweck:

Gemeinsame DTOs / Models, die zwischen API & Blazor Web geteilt werden.

Abhängigkeiten:

- nur einfache Datentypen
- Referenziert von BlazorERP.Api und BlazorERP.Web

Empfehlung für dich

In deinem Fall würde ich zusätzlich anlegen:

css

Code kopieren

```
src/
  └── BlazorERP.Domain/
  └── BlazorERP.Application/
  └── BlazorERP.Infrastructure/
    └── BlazorERP.Shared/
```

Danach:

- BlazorERP.Api referenziert Infrastructure & Application

- BlazorERP API referenziert Infrastructure & Application

- BlazorERP.Application referenziert Domain
- BlazorERP.Web referenziert Shared

Wenn du möchtest, kann ich dir ein Beispiel- `Directory.Build.props` und die Projekt-Referenzen (`.csproj`-Verknüpfungen) zeigen, wie sie in einer sauberen Clean-Architecture-Lösung aussehen würden.

Soll ich das für dich generieren?



die api soll die domains haben. die request zum erstellen eines kunden ist eine andere als die zum updaten. ich möchte auch keine weiteren ordner hinzufügen, nur web und api. In api sollen alle datenbank aufrufe passieren und das blazorerp projekt soll mit contracts die api aufrufen

Perfekt — du willst also eine **vereinfachte Clean Architecture**, aber ohne zusätzliche Projekte oder Ebenen.

Nur api und web , wobei:

- die **API** das gesamte Backend (inkl. Datenbankzugriff, Domain-Logik etc.) enthält,
- die **Blazor-WebApp** (Client) ausschließlich per **Contracts/DTOs** mit der API kommuniziert.

Das ist ein sehr pragmatischer Ansatz – besonders geeignet, wenn du Dapper nutzt und kein überkomplexes Projekt-Setup willst.



Empfohlene Projektstruktur (Minimal Clean Architecture)

css

Code kopieren

```
src/
  |- api/
    |   |- DOMAINES.DNI
```

```
|   -- BlazorERP.Api  
└ web/  
    └ BlazorERP.Web  
      └ BlazorERP.Contracts
```

💡 1. BlazorERP.Api

Zweck:

Backend mit Domainlogik, Repositories, Services und API-Endpunkten.

Schichten intern (nur als Ordner, keine neuen Projekte):

mathematica Code kopieren

```
BlazorERP.Api/  
  └ Domain/  
    |   └ Entities/  
    |   |   └ Customer.cs  
    |   └ Enums/  
    |   └ ValueObjects/  
    |  
    |  
  └ Application/  
    |   └ Customers/  
    |   |   └ Commands/  
    |   |   |   └ CreateCustomerCommand.cs  
    |   |   |   └ UpdateCustomerCommand.cs  
    |   |   └ Queries/  
    |   |   |   └ GetCustomerByIdQuery.cs  
    |   |   └ GetCustomersQuery.cs  
    |   └ Handlers/  
    |       └ CreateCustomerHandler.cs  
    |       └ UpdateCustomerHandler.cs  
    |       └ GetCustomerByIdHandler.cs  
  └ Common/  
    |   └ Interfaces/  
    |   |   └ ICustomerRepository.cs  
    |   |   └ IUnitOfWork.cs  
    |   └ Behaviors/
```

```
└─ Infrastructure/
    |   └─ Persistence/
    |       |   └─ DapperConnectionFactory.cs
    |       |   └─ CustomerRepository.cs
    |       └─ DbInitializer.cs
    |   └─ Services/
    └─ Config/
```



```
└─ Api/
    |   └─ Controllers/
    |       |   └─ CustomersController.cs
    |   └─ Extensions/
    |       |   └─ DependencyInjection.cs
    |       └─ Endpoints.cs
```

◆ Domain

Hier definierst du deine Kernobjekte:

csharp Code kopieren

```
namespace BlazorERP.Api.Domain.Entities;

public class Customer
{
    public Guid Id { get; set; }
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```

◆ Commands & Queries

Du kannst einfache Request-Klassen definieren:

csharp Code kopieren

```
public record CreateCustomerCommand(string Name, string Email);
public record UpdateCustomerCommand(Guid Id, string Name, string Email);
```

Handlers führen dann die Logik aus – mit Repository-Aufrufen über Dapper.

🌐 2. BlazorERP.Contracts

Zweck:

Enthält **DTOs, Enums und Request-/Response-Modelle**, die zwischen API und Web geteilt werden.

Beispielstruktur:

markdown

[Code kopieren](#)

```
BlazorERP.Contracts/
├── Customers/
│   ├── CustomerDto.cs
│   ├── CreateCustomerRequest.cs
│   ├── UpdateCustomerRequest.cs
│   └── CustomerResponse.cs
└── Common/
    └── ApiResult.cs
```

Beispiel DTO:

csharp

[Code kopieren](#)

```
public class CustomerDto
{
    public Guid Id { get; set; }
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```

Beispiel Request:

csharp

[Code kopieren](#)

```
public class CreateCustomerRequest
{
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```



3. BlazorERP.Web

Zweck:

Blazor-Frontend, das die API über HttpClient aufruft.

Struktur:

markdown

Code kopieren

```
BlazorERP.Web/
├─ Pages/
│  ├─ Customers.razor
│  ├─ CustomerEdit.razor
├─ Services/
│  ├─ CustomerApiClient.cs
│  └─ ApiClientBase.cs
└─ Shared/
    ├─ MainLayout.razor
    └─ NavMenu.razor
```

Beispiel API-Client:

csharp

Code kopieren

```
public class CustomerApiClient
{
    private readonly HttpClient _http;

    public CustomerApiClient(HttpClient http)
    {
        _http = http;
    }

    public async Task<List<CustomerDto>> GetAllAsync()
        => await _http.GetFromJsonAsync<List<CustomerDto>>("api/customers") ?? []

    public async Task CreateAsync(CreateCustomerRequest request)
        => await _http.PostAsJsonAsync("api/customers", request);
```

}

Fazit

Du brauchst keine weiteren Projekte, aber du **kannst intern in der API zwischen Domain / Application / Infrastructure per Ordner trennen** – das bleibt modular und wartbar.

Kernpunkte deines Setups:

- Nur `BlazorERP.Api`, `BlazorERP.Web`, `BlazorERP.Contracts`
- `Api` enthält Domain + Datenbankzugriff (Dapper)
- `Contracts` enthält DTOs, Requests und Responses
- `Web` konsumiert `Contracts` und ruft `Api` via HttpClient auf

Möchtest du, dass ich dir auf Basis dieses Modells ein vollständiges Beispiel (inkl. csproj-Referenzen und Beispielcode für einen Customer-Endpoint) generiere?



die api sollte schon in projekte getrennt sein

Perfekt — das ist die **saubere Clean-Architecture-Variante mit echten Projekt-Trennungen innerhalb der API**.

Das ist die beste Lösung, wenn du langfristig wartbare und testbare Logik möchtest, aber trotzdem klar trennen willst, was wo passiert.

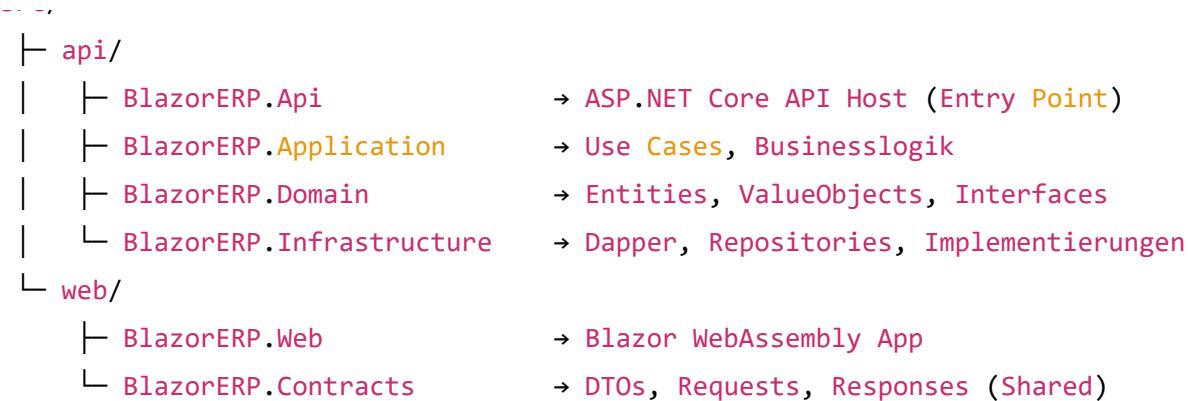
Hier ist der empfohlene Aufbau speziell für **.NET 9 + Dapper + Blazor WASM**:

Projektstruktur

mathematica

Code kopieren

`src/`



🧠 Projektrollen im Detail

1. BlazorERP.Domain

- Enthält:
Entities, Value Objects, Enums, Domain Events, Interfaces (z. B. ICustomerRepository)
- Keine Frameworkabhängigkeit (kein EF, kein Dapper, kein Web-Zeug)

Beispiel:

csharp

Code kopieren

```
namespace BlazorERP.Domain.Entities;

public class Customer
{
    public Guid Id { get; set; }
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```

2. BlazorERP.Application

- Enthält:
Business-Use-Cases (z. B. CreateCustomerHandler , UpdateCustomerHandler)
- Nutzt **Interfaces** aus Domain
- Verwendet ggf. MediatR oder eigene Command/Query-Struktur
- Keine Abhängigkeit auf Datenbank oder Infrastruktur

Beispiel:

csharp

Code kopieren

```
namespace BlazorERP.Application.Customers.Commands;

public record CreateCustomerCommand(string Name, string Email);

public class CreateCustomerHandler
{
    private readonly ICustomerRepository _repo;

    public CreateCustomerHandler(ICustomerRepository repo)
        => _repo = repo;

    public async Task<Guid> Handle(CreateCustomerCommand command)
    {
        var customer = new Customer { Id = Guid.NewGuid(), Name = command.Name, E
        await _repo.AddAsync(customer);
        return customer.Id;
    }
}
```

3. BlazorERP.Infrastructure

- Enthält:
Implementierungen der Interfaces (z. B. CustomerRepository mit Dapper)
- DB-Zugriff, Konfiguration, externe Services
- Keine Businesslogik

Beispiel:

csharp

Code kopieren

```
public class CustomerRepository : ICustomerRepository
{
    private readonly IDbConnection _db;

    public CustomerRepository(IDbConnection db) => db = db;
```

```
public async Task AddAsync(Customer customer)
{
    const string sql = "INSERT INTO Customers (Id, Name, Email) VALUES (@Id, @Name, @Email)";
    await _db.ExecuteAsync(sql, customer);
}
```

4. BlazorERP.Api

- Entry Point für das Backend
- Enthält nur:
 - Controller/Endpoints
 - DI Setup
 - Middleware
 - API-Routen
- Referenziert:
 - Application
 - Infrastructure
 - Domain
 - Contracts

Beispiel:

csharp

Code kopieren

```
[ApiController]
[Route("api/[controller]")]
public class CustomersController : ControllerBase
{
    private readonly CreateCustomerHandler _createHandler;

    public CustomersController(CreateCustomerHandler createHandler)
        => _createHandler = createHandler;

    [HttpPost]
    public async Task<IActionResult> Create(CreateCustomerRequest request)
```

```
    }
    var id = await _createHandler.Handle(new CreateCustomerCommand(request.Name));
    return Ok(new { Id = id });
}
}
```

5. BlazorERP.Contracts

- Shared DTOs und Requests zwischen Web und API

Beispiel:

csharp Code kopieren

```
public class CreateCustomerRequest
{
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}

public class CustomerDto
{
    public Guid Id { get; set; }
    public string Name { get; set; } = default!;
    public string Email { get; set; } = default!;
}
```

6. BlazorERP.Web

- Blazor WASM Frontend
- Nutzt HttpClient + Contracts
- Ruft API auf (z. B. /api/customers)

Beispiel:

csharp Code kopieren

```
public class CustomerApiClient
{
```

```
private readonly HttpClient _http;

public CustomerApiClient(HttpClient http) => _http = http;

public async Task CreateAsync(CreateCustomerRequest request)
=> await _http.PostAsJsonAsync("api/customers", request);
}
```

🔗 Projektverweise (.csproj)

Projekt	Referenzen
BlazorERP.Domain	–
BlazorERP.Application	→ Domain
BlazorERP.Infrastructure	→ Application, Domain
BlazorERP.Api	→ Application, Infrastructure, Domain, Contracts
BlazorERP.Contracts	–
BlazorERP.Web	→ Contracts

✓ Vorteile dieses Ansatzes

- Klare Trennung von Verantwortlichkeiten
- Kein Framework-Leak in Domain/Application
- Wartbar, testbar und trotzdem kompakt
- Web-Frontend und API sauber getrennt über Contracts
- Du kannst Dapper, MediatR, FluentValidation usw. sauber einbinden

Möchtest du, dass ich dir die **.csproj-Dateien (mit Referenzen)** und die **Verzeichnisstruktur** als Vorlage generiere, die du direkt in Visual Studio anlegen kannst (inkl. MinimalAPI-Setup für .NET 9)?



wo würdest du jetzt logiken für das front end rein machen für die dtos? z.B. sowas wie getcustomeraddress

Sehr gute (und wichtige) Frage — das ist einer der **häufigsten Stolpersteine** bei Blazor + Clean Architecture.

Du hast ja im Prinzip zwei getrennte Welten:

- **Backend (API)** → Geschäftslogik, Datenbankzugriff, Domain
- **Frontend (Blazor Web)** → UI, Darstellung, Interaktion

Die Frage ist also: *Wo gehört Logik hin, die auf DTOs im Frontend arbeitet?*

Zum Beispiel: `GetCustomerAddress()` oder Formatierungen wie "Herr Mustermann, Berlin".

1. Grundregel

Frontend-Logik, die nur für Darstellung oder UI-Zwecke relevant ist, gehört ins Blazor-Web-Projekt.

Also:

Wenn du aus einem DTO (`CustomerDto`) etwas zusammensetzt, formatierst oder berechnest,

gehört das **nicht in die API**, sondern **in die Web-Schicht** – weil das reines Anzeigeverhalten ist.

Beispiel: CustomerDto im Contracts-Projekt

Im `BlazorERP.Contracts`-Projekt bleibt das DTO rein **datentransferorientiert** – keine Logik, keine Formatierung:

csharp

Code kopieren

```
namespace BlazorERP.Contracts.Customers;

public class CustomerDto
{
    public Guid Id { get; set; }
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
    public string Street { get; set; } = default!;
    public string City { get; set; } = default!;
    public string PostalCode { get; set; } = default!;
}
```

💻 2. Logik für Darstellung im Frontend (Extension Methods oder Partial Classes)

Im BlazorERP.Web -Projekt kannst du **DTO-spezifische Erweiterungen** hinzufügen.
Dafür gibt es zwei elegante Wege:

✳️ Variante A — Extension Methods (empfohlen)

Erstelle z. B. in BlazorERP.Web/Extensions/CustomerDtoExtensions.cs :

csharp Code kopieren

```
using BlazorERP.Contracts.Customers;

namespace BlazorERP.Web.Extensions;

public static class CustomerDtoExtensions
{
    public static string GetFullAddress(this CustomerDto customer)
        => $"{customer.Street}, {customer.PostalCode} {customer.City}";

    public static string GetDisplayName(this CustomerDto customer)
        => $"{customer.FirstName} {customer.LastName}";
}
```

Diese Varianten sind in diesem Projekt von mir ausgewählt.

Dann kannst du in deinen Razor-Komponenten einfach schreiben:

razor

[Code kopieren](#)

```
@using BlazorERP.Web.Extensions  
<p>@customer.GetDisplayName()</p>  
<p>@customer.GetFullAddress()</p>
```

 **Vorteile:**

- Trennt DTO und UI-Logik klar
- Wiederverwendbar in allen Komponenten
- Keine Vermischung von Backend und UI-Verhalten

 **Variante B — Partial Classes (auch okay)**

Wenn du lieber direkt am DTO arbeitest, kannst du im Web-Projekt eine **Partial Class** anlegen, die das DTO aus `Contracts` erweitert.

csharp

[Code kopieren](#)

```
// Datei: BlazorERP.Web/Models/CustomerDto.Extensions.cs  
namespace BlazorERP.Contracts.Customers;  
  
public partial class CustomerDto  
{  
    public string FullAddress => $"{Street}, {PostalCode} {City}";  
    public string DisplayName => $"{FirstName} {LastName}";  
}
```

Damit kannst du in Razor schreiben:

razor

[Code kopieren](#)

```
<p>@customer.DisplayName</p>  
<p>@customer.FullAddress</p>
```

 **Vorteil:**

Sauber integriert, wirkt wie "echte" Eigenschaften am DTO.

 **Nachteil:**

Kann später verwirren, wenn man nicht weiß, woher die Logik kommt — also gut dokumentieren.

🚫 Was du vermeiden solltest

✗ Keine UI-Logik in:

- Contracts (die sind rein für Datentransfer da)
- Api (Backend darf von UI nichts wissen)
- Domain oder Application (rein Business)

✓ Empfehlung für dich

Wenn du saubere Trennung behalten willst, mach's so:

[Code kopieren](#)

```
BlazorERP.Web/  
|   └── Extensions/  
|       |   └── CustomerDtoExtensions.cs  
|       |   └── OrderDtoExtensions.cs  
|       └── ...
```

So hast du alle UI-spezifischen Formatierungen zentral gesammelt und klar getrennt von deinen reinen Datenmodellen aus Contracts .

Wenn du willst, kann ich dir ein **kurzes Codebeispiel zeigen**, wie du das dann in einer Blazor-Komponente (z. B. `CustomerDetails.razor`) sauber verwendest – mit Extension Method, Binding und API-Aufruf.
Möchtest du das sehen?



wo würde in dem projekt ui logik platziert sein?

Sehr gute Anschlussfrage — das ist der entscheidende Punkt bei **Clean Architecture + Blazor**.

Also: *Wo genau lebt die UI-Logik (nicht Business-Logik, sondern z. B. State, Validierung, Interaktion, Anzeige)?*

Ich erkläre dir das konkret für deine Struktur (mit `BlazorERP.Web` als Blazor-Frontend und `BlazorERP.Api` als Backend).

✳️ Grundprinzipien

- **Business-Logik** → gehört in `BlazorERP.Application` (und wird über die API bereitgestellt)
- **UI-Logik** → gehört ausschließlich ins `BlazorERP.Web -Projekt`
- **UI-spezifische Models (ViewModels, State, Commands)** → leben im Web-Projekt, nicht in Contracts oder Api

📁 Struktur im `BlazorERP.Web -Projekt`

Eine saubere, modulare Struktur sieht z. B. so aus:

vbnnet

[Code kopieren](#)

```
BlazorERP.Web/
├── Pages/
│   ├── Customers/
│   │   ├── CustomerList.razor
│   │   ├── CustomerDetail.razor
│   │   └── CustomerEdit.razor
│   └── Orders/
│       └── ...
└── Components/
    ├── Shared/
    │   └── CustomControls/
    └── Services/
        └── Api/
            └── CustomerApiClient.cs
```

```
|   |   └ OrderApiClient.cs
|   └ State/
|       └ CustomerState.cs
|           └ OrderState.cs
└ ViewModels/
    └ CustomerViewModel.cs
    └ OrderViewModel.cs
└ Extensions/
    └ CustomerDtoExtensions.cs
    └ FormatHelpers.cs
└ Validators/
    └ CustomerValidator.cs
    └ OrderValidator.cs
└ Program.cs
```



1. UI-Logik = State, ViewModels, Validierung

Das ist **reine UI-spezifische Logik**, die beschreibt, *wie die Oberfläche sich verhält* – z. B.:

- Wann wird ein Formular validiert?
- Welche Buttons sind aktiv?
- Wie werden Daten im UI kombiniert oder sortiert?

Diese Logik lebt in **ViewModels** oder **State-Klassen**, nicht in den DTOs oder Komponenten selbst.



Beispiel: CustomerViewModel

csharp

Code kopieren

```
using BlazorERP.Contracts.Customers;

namespace BlazorERP.Web.ViewModels;

public class CustomerViewModel
{
    public CustomerDto Data { get; set; } = new();
    public bool IsNew => Data.Id == Guid.Empty;
```

```
public string DisplayTitle => IsNew  
    ? "Neuer Kunde"  
    : $"{Data.FirstName} {Data.LastName}";  
  
public string GetAddressDisplay()  
=> $"{Data.Street}, {Data.PostalCode} {Data.City}";  
}
```

💡 → Das ist UI-Logik: *Wie soll der Kunde auf der Seite angezeigt werden?*
(Das gehört **nicht** in API oder Contracts!)

✳️ Beispiel: CustomerState (z. B. mit Flux-ähnlichem Pattern oder einfachem Singleton)

csharp Code kopieren

```
public class CustomerState  
{  
    public List<CustomerDto> Customers { get; private set; } = new();  
  
    public event Action? OnChange;  
  
    public void SetCustomers(List<CustomerDto> customers)  
    {  
        Customers = customers;  
        NotifyStateChanged();  
    }  
  
    private void NotifyStateChanged() => OnChange?.Invoke();  
}
```

💡 → Das ist "UI-State" — *Wie ist der aktuelle Zustand der Anwendung im Browser?*

✳️ Beispiel: CustomerValidator

Optional mit FluentValidation oder eigener Logik:

csharp

Code kopieren

```
using FluentValidation;
using BlazorERP.Contracts.Customers;

public class CustomerValidator : AbstractValidator<CustomerDto>
{
    public CustomerValidator()
    {
        RuleFor(x => x.FirstName).NotEmpty();
        RuleFor(x => x.Email).EmailAddress();
    }
}
```

🔌 2. Wo liegt der API-Aufruf?

In **Service-Klassen** unter `/Services/Api/`.

Diese kapseln den `HttpClient` und geben DTOs zurück.

csharp

Code kopieren

```
public class CustomerApiClient
{
    private readonly HttpClient _http;

    public CustomerApiClient(HttpClient http) => _http = http;

    public async Task<List<CustomerDto>> GetAllAsync()
        => await _http.GetFromJsonAsync<List<CustomerDto>>("api/customers") ?? []
}
```

Dann nutzt du sie z. B. in deiner Razor-Komponente:

razor

Code kopieren

```
@inject CustomerApiClient Api
@inject CustomerState State

@code {
```

```
protected override async Task OnInitializedAsync()
{
    var customers = await Api.GetAllAsync();
    State.SetCustomers(customers);
}
```

3. Was gehört nicht in die UI-Logik?

- Kein Datenbankzugriff
- Keine Business-Entscheidungen ("darf Kunde gelöscht werden?" → gehört ins Backend)
- Keine Validierungen, die Geschäftsregeln abbilden (nur UI-seitig, z. B. "Feld darf nicht leer sein")

Zusammenfassung

Bereich	Speicherort	Beschreibung
DTOs, Requests, Responses	BlazorERP.Contracts	Reine Datenstrukturen (ohne Logik)
**Business-Logik (z. B. „Kunde darf nur gelöscht werden, wenn keine Bestellung existiert“) **	BlazorERP.Application	Geschäftsregeln
**UI-Logik (z. B. „Button nur aktiv, wenn Formular valide“) **	BlazorERP.Web	ViewModels, State, Validatoren
Datenzugriff (Dapper, SQL)	BlazorERP.Infrastructure	Implementierung von Repositories
Darstellung (Razor-Komponenten)	BlazorERP.Web/Pages	Views, Bindings, UI-Events

Wenn du willst, kann ich dir im nächsten Schritt ein Beispiel zeigen, wie das Ganze konkret zusammenspielt:

z. B. `CustomerPage.razor` → ruft `CustomerApiClient` auf → schreibt in `CustomerState` → rendert per `CustomerViewModel`.

Möchtest du das Beispiel sehen (komplett durchverkabelt)?



ok wie würde ich jetzt z.B. den löschen button im frontend disablen, wenn diese logik nur ins backend darf?

Sehr gute Frage — das ist **genau** der Punkt, an dem viele Projekte zwischen „sauberer Architektur“ und „brauchbarer UX“ kippen.

Du willst:

- die **Business-Regel** („Darf Kunde gelöscht werden?“) im **Backend** haben
- aber trotzdem den **Löschen-Button im Frontend** deaktivieren, wenn das nicht erlaubt ist

Das lässt sich **vollständig regelkonform** mit Clean Architecture lösen — ohne Logik ins UI zu schieben.

Hier ist die saubere Vorgehensweise

1. Trennung verstehen

Ebene	Aufgabe
Backend (Application / Domain)	Entscheidet <i>was erlaubt ist</i>
Frontend (Web)	Reagiert nur auf die vom Backend gelieferten Informationen

Das Frontend darf **nicht selbst entscheiden**, ob etwas „erlaubt“ ist — es darf aber **anzeigen, was das Backend entschieden hat**.

Also: das Frontend **zeigt** den Zustand, **entscheidet** aber nicht darüber.

✳️ 2. Beispiel – “Kunde darf nur gelöscht werden, wenn keine Bestellungen existieren”

☰ Backend (BlazorERP.Application)

Du fügst im Backend eine Methode hinzu, die **diese Regel prüft**:

csharp

Code kopieren

```
public class CustomerRulesService
{
    private readonly IOrderRepository _orders;

    public CustomerRulesService(IOrderRepository orders)
    {
        _orders = orders;
    }

    public async Task<bool> CanDeleteAsync(Guid customerId)
    {
        var hasOrders = await _orders.ExistsByCustomerIdAsync(customerId);
        return !hasOrders;
    }
}
```

🌐 API-Endpoint

Die API bietet dann explizit einen Endpunkt an, um die Regel abzufragen:

csharp

Code kopieren

```
[HttpGet("{id}/can-delete")]
public async Task<ActionResult<bool>> CanDelete(Guid id)
{
    var result = await _rulesService.CanDeleteAsync(id);
    return Ok(result);
}
```

Oder — eleganter — du gibst es direkt im **CustomerDto** zurück.

Contracts

In `BlazorERP.Contracts` erweiterst du das DTO um ein entsprechendes Feld:

csharp

[Code kopieren](#)

```
public class CustomerDto
{
    public Guid Id { get; set; }
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
    public bool CanDelete { get; set; } // 👍 vom Backend gesetzt
}
```

Das Backend setzt es beim Mapping:

csharp

[Code kopieren](#)

```
dto.CanDelete = await _rulesService.CanDeleteAsync(customer.Id);
```

3. Frontend-Integration

Im Blazor-Frontend (`BlazorERP.Web`) bekommst du dann einfach das DTO mit dieser Info:

razor

[Code kopieren](#)

```
@inject CustomerApiClient Api

@if (customer != null)
{
    <h3>@customer.FirstName @customer.LastName</h3>
    <button class="btn btn-danger"
            disabled="@(!customer.CanDelete)"
            @onclick="DeleteCustomer">
        Löschen
    </button>
```

}

```
@code {
    private CustomerDto? customer;

    protected override async Task OnInitializedAsync()
        => customer = await Api.GetByIdAsync(customerId);

    private async Task DeleteCustomer()
    {
        await Api.DeleteAsync(customer!.Id);
    }
}
```

💡 Das Frontend entscheidet nicht selbst.

Es zeigt nur das an, was das Backend vorgibt (`CanDelete = false` → Button disabled).

🧱 4. Alternative: API gibt AllowedActions zurück

Wenn du viele Aktionen prüfen musst (Löschen, Bearbeiten, Deaktivieren ...), kannst du eine kleine Policy-Struktur zurückgeben:

csharp

Code kopieren

```
public class CustomerDto
{
    public Guid Id { get; set; }
    public string FullName { get; set; } = default!;
    public CustomerActionsDto AllowedActions { get; set; } = new();
}

public class CustomerActionsDto
{
    public bool CanDelete { get; set; }
    public bool CanEdit { get; set; }
    public bool CanActivate { get; set; }
}
```

Dann kannst du im UI einfach schreiben:

razor

[Code kopieren](#)

```
<button disabled="@(!customer.AllowedActions.CanDelete)">Löschen</button>
```

Vorteile dieses Ansatzes

Vorteil	Beschreibung
 Saubere Trennung	Backend entscheidet über Business-Regeln
 UI bleibt dumm	Es zeigt nur das an, was erlaubt ist
 Wiederverwendbar	Die gleiche Logik kann auch im API-Client, Tests, CLI etc. genutzt werden
 Erweiterbar	Du kannst später Rollen, Berechtigungen, Kontext hinzufügen (z. B. UserRole)

Fazit

Regel im Backend, Reaktion im Frontend.

- Das Backend berechnet Flags (CanDelete , CanEdit , ...).
- Das Frontend liest sie aus und passt das UI an.
- Keine Businesslogik im UI, aber dennoch hervorragende UX.

Wenn du willst, kann ich dir den kompletten Ablauf (API + Dapper + Blazor-Button + DTO) als Beispiel in Codeform zusammenstellen, damit du siehst, wie es von DB bis Button durchläuft.

Willst du das Beispiel komplett?



