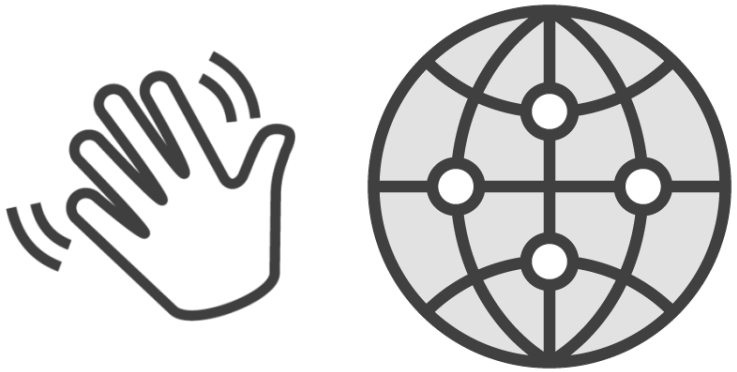
Designing and Building Enterprise Blazor Applications

ESSENTIAL LIFE CYCLE CONCEPTS

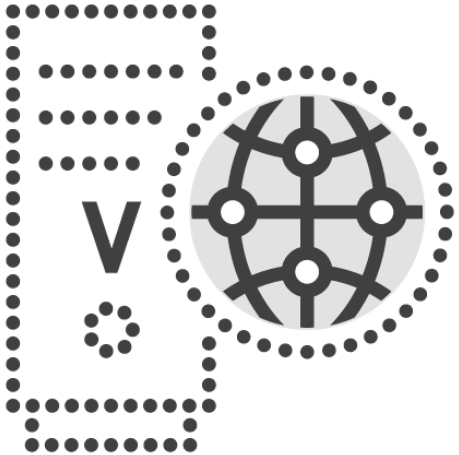
Alex Wolf

www.crywolfcode.com

Going Further with Blazor



Hello World!



Hello Real World

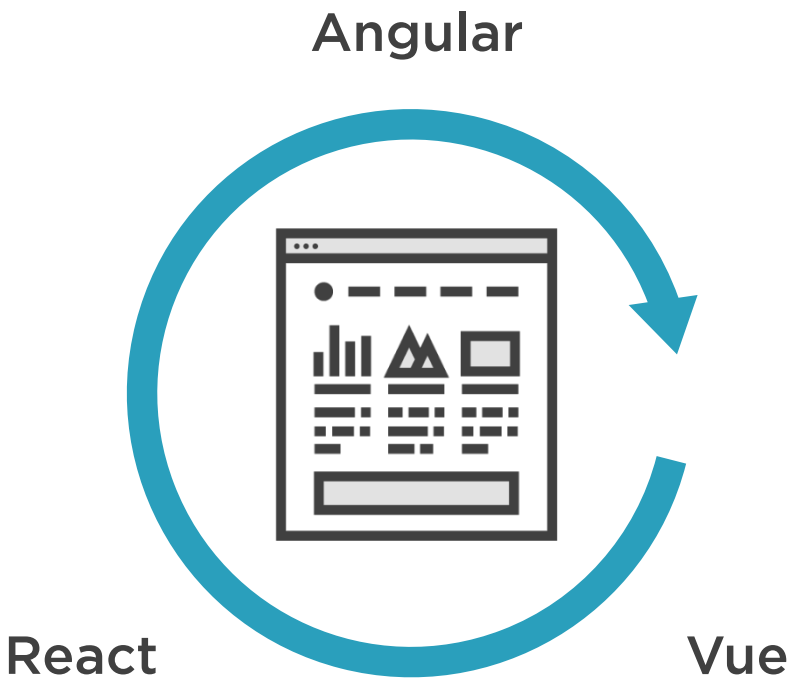
Our Goal:

Increase our understanding of Blazor to build more meaningful applications.

Why?

Why Blazor in the Enterprise?

JavaScript and Blazor



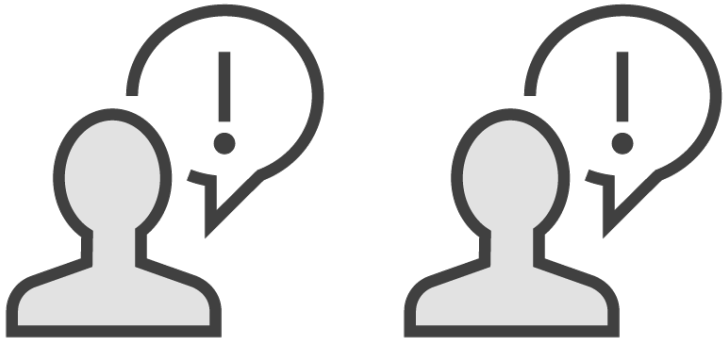
JavaScript frameworks

Blazor?

.NET framework

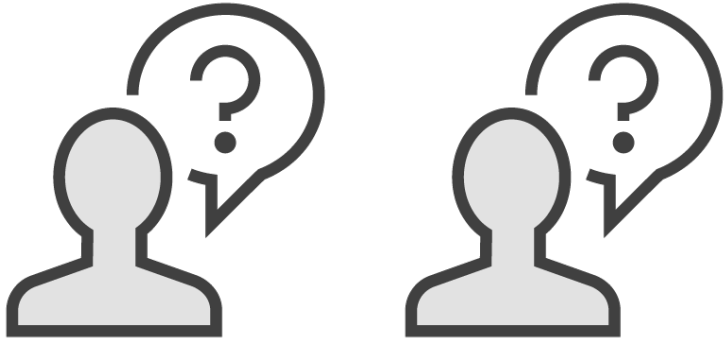
Blazor and JavaScript

C#



.NET Developers

JavaScript



.NET Developers

Advantages of Blazor

.NET packages

NuGet Packages built on .NET Standard

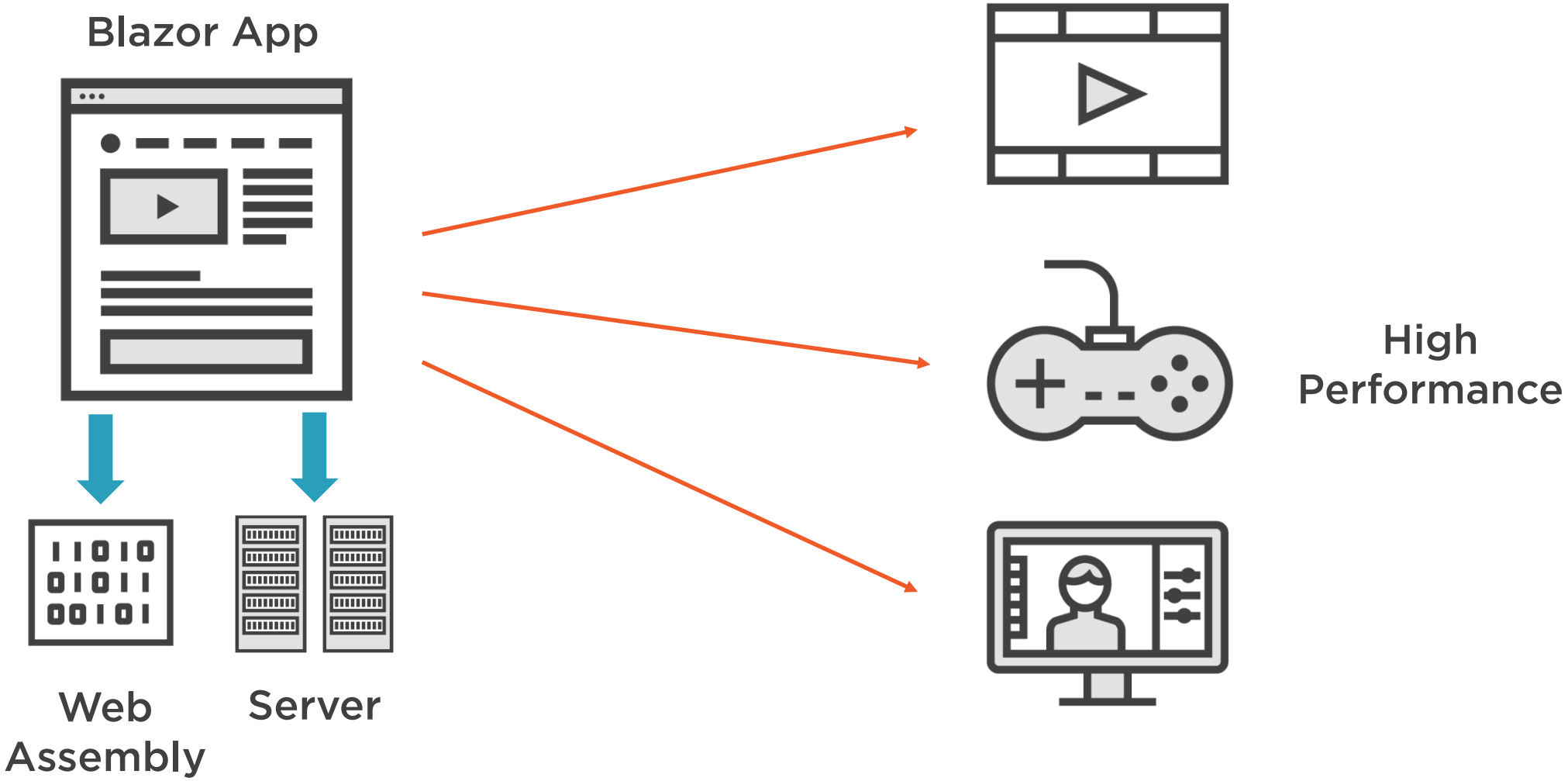
Powerful tooling

Visual Studio, VS Code, command line and more

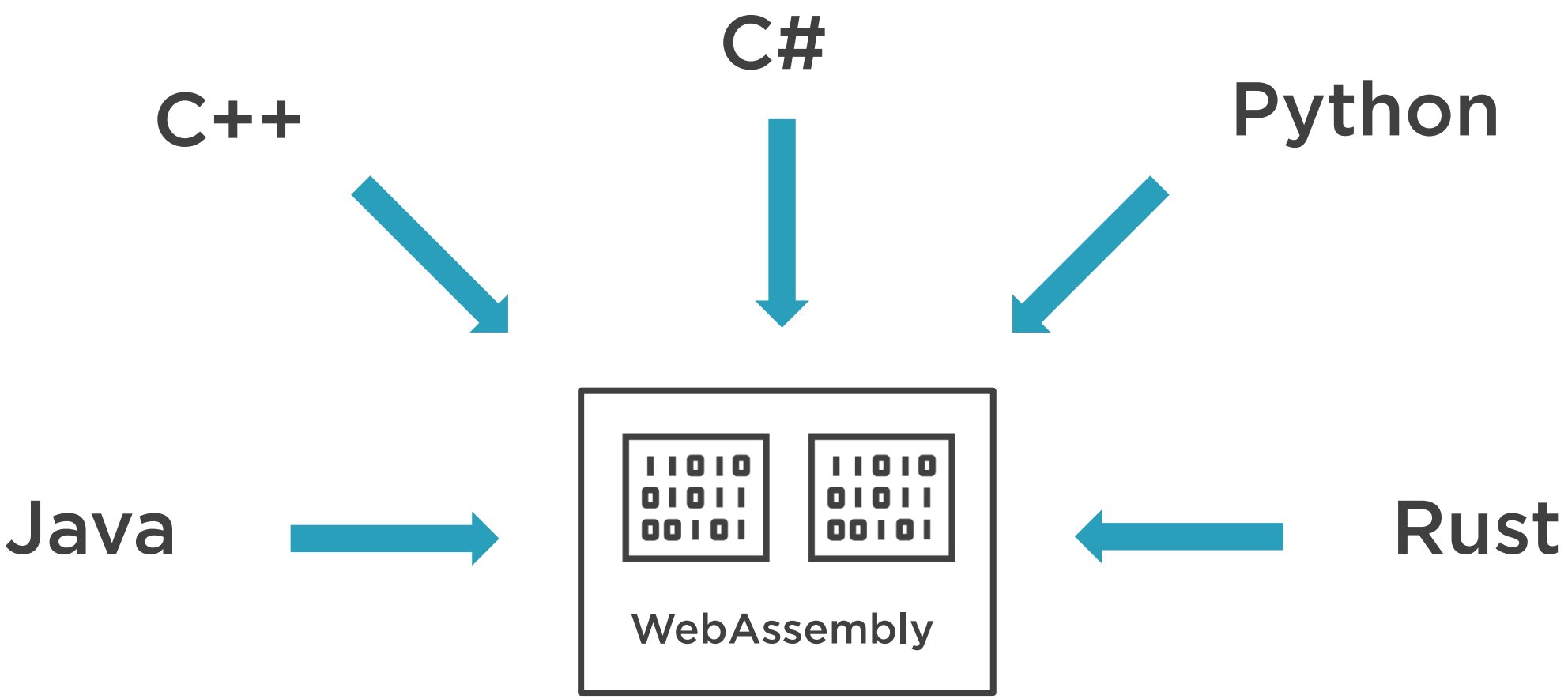
Microsoft support

An official, supported part of .NET Core

Performance Gains



Expanding Browser Languages



Blazor is also great just because....it's great.

Understanding Blazor Hosting Models

Blazor is one framework with two hosting models.

Additional Resources for Hosting Models

Blazor: Getting Started

Blazor: The Big Picture

Understanding the Two Hosting Models

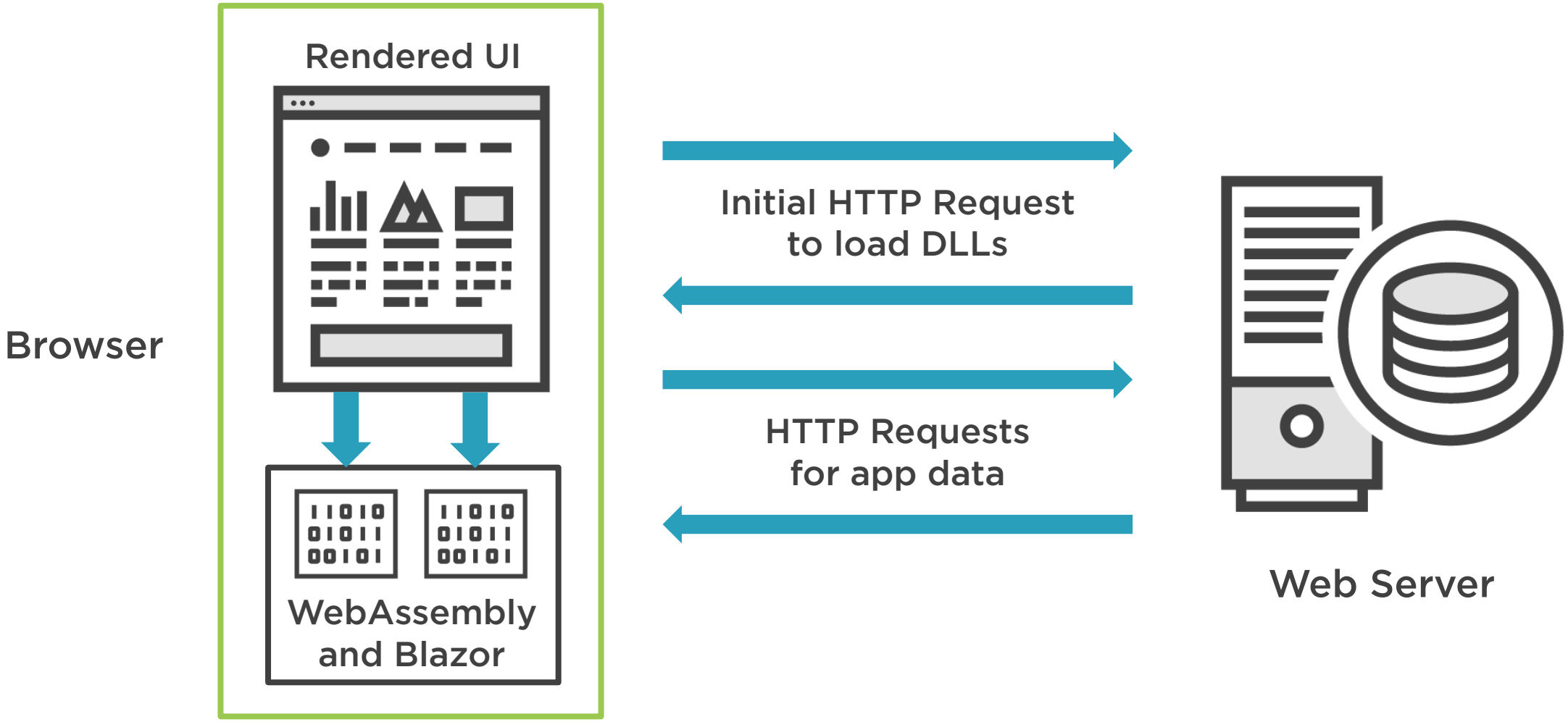
Blazor WebAssembly

Runs directly in the browser on top of WebAssembly

Blazor Server

Processed server side with updates sent to the browser

Understanding Blazor WebAssembly



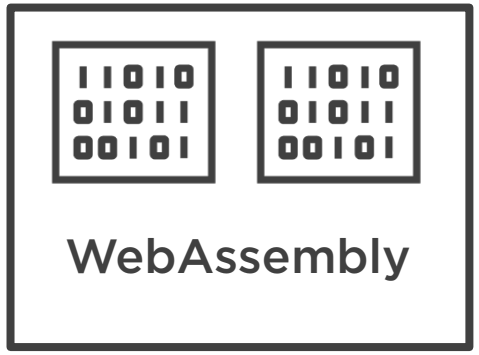
Understanding Blazor Server

One Application, Two Hosting Models

Blazor Application

Services
Components
Markup
Logic

Same app code
regardless of hosting



Browser

Server

Choosing a Hosting Model

Blazor WebAssembly

Released in May 2020

Blazor Server

Released in September 2019

Blazor Design Patterns

Component Driven Design

Bethany's Pie Shop

Bethany's Pie Shop HRM

Home
Tasks
Employees
Opportunities
Expenses
Feedback
Staff Directory

Welcome to Bethany's Pie Shop HRM!

Component

Notices

Description	Priority	Date Posted
Today is boss's appreciation day! Remind employees to thank their upline for all their support and hard work.	Low	10/10/2019
Open enrollment for benefits is coming - make sure a reminder is communicated to all staff.	High	10/05/2019
We won an award! Our pie shops were recognized for their continued cleanliness by the American Pie Association.	Low	10/01/2019

Component

Tasks

Add Task

Title	Description	Status
Employee Onboarding	Joe is having an issue with his account login, please look into it.	Open
Kitchen Duty	The fridge needs to be cleaned out and people are ignoring the weekly rotation.	Open
Welcome Lunch	Plan a welcome lunch for our new employees	Open
asef	asefase	Open
asef	asef	InProgress

Component

Component

New Employees

We're constantly expanding our team. Make sure to say hello to our latest friends!

Bethany Smith Bob Smith

Remmber, we offer referral bonuses!

Report a Concern

It's simple, safe and anonymous.

Title:

Enter title

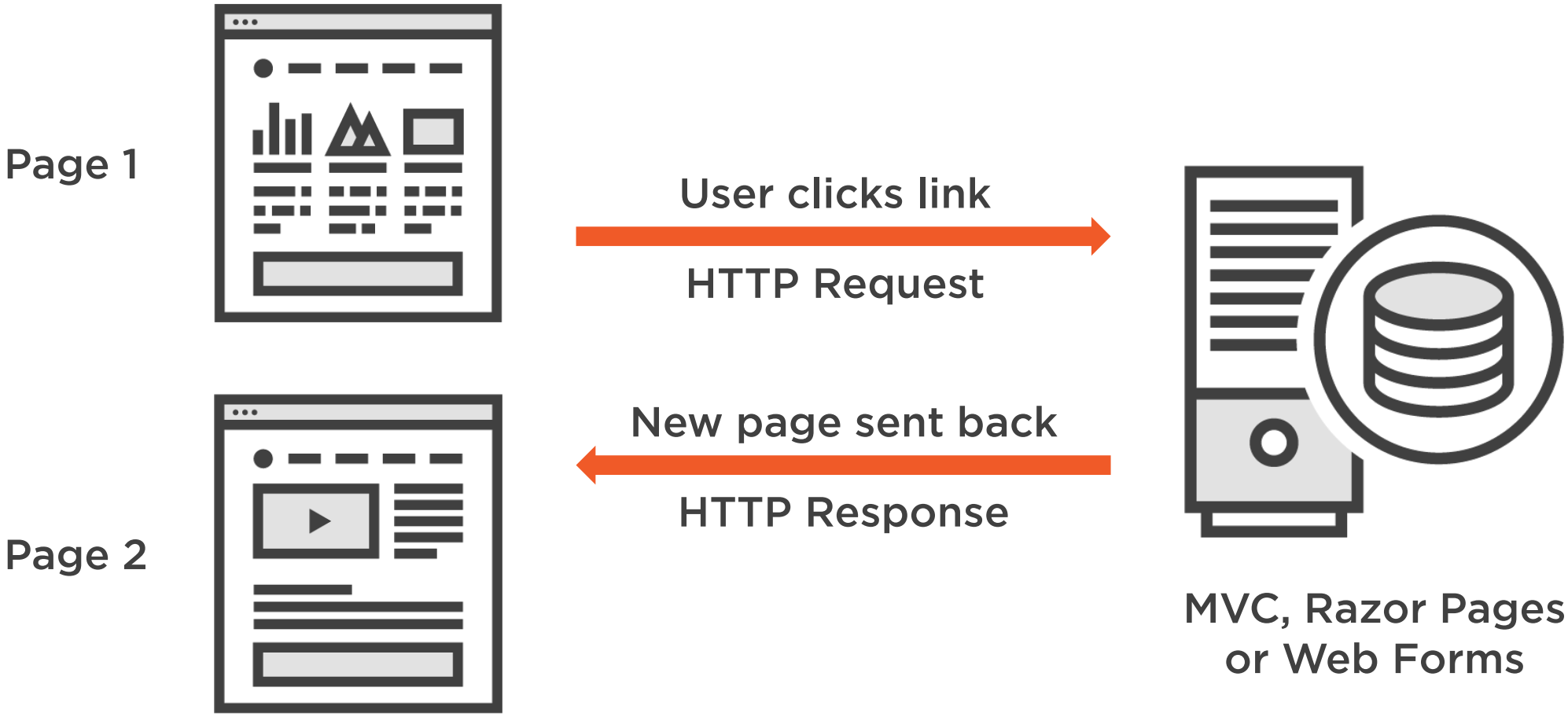
Description:

Enter title

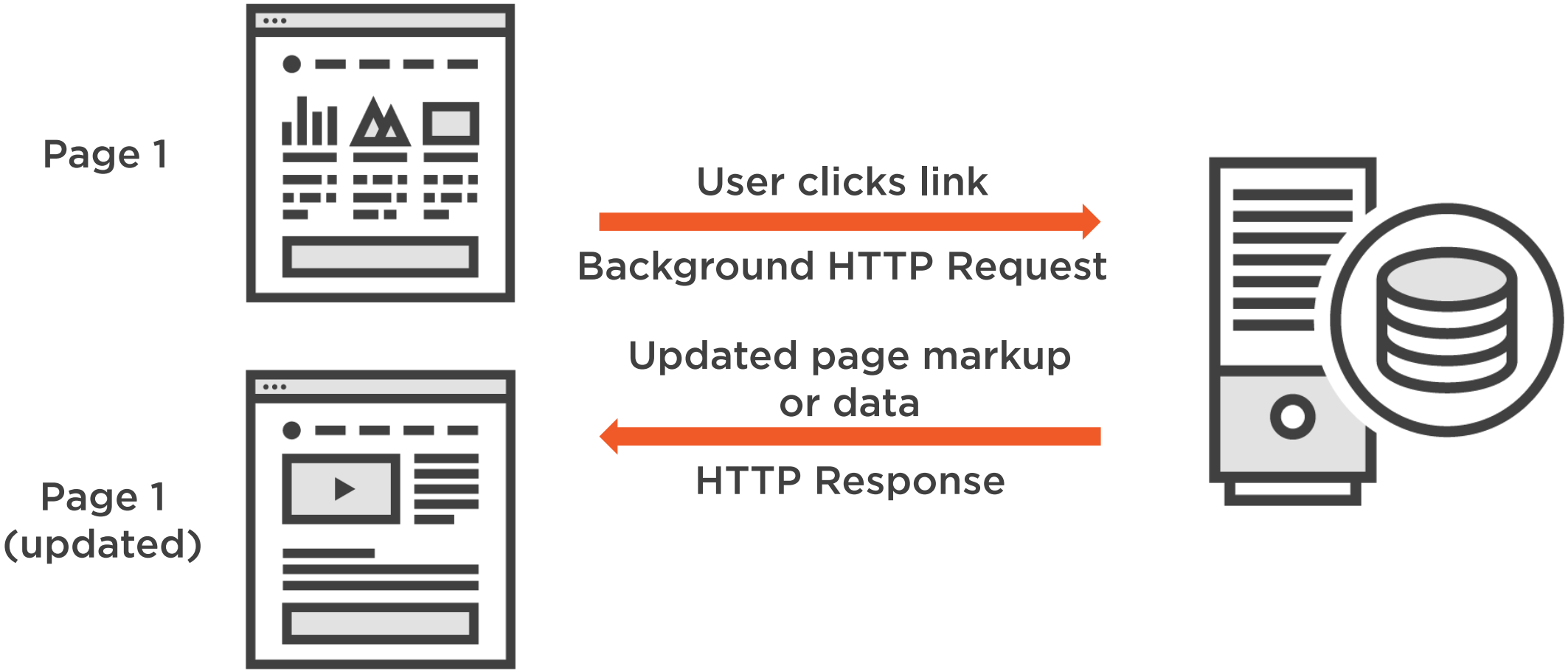
Submit

Component

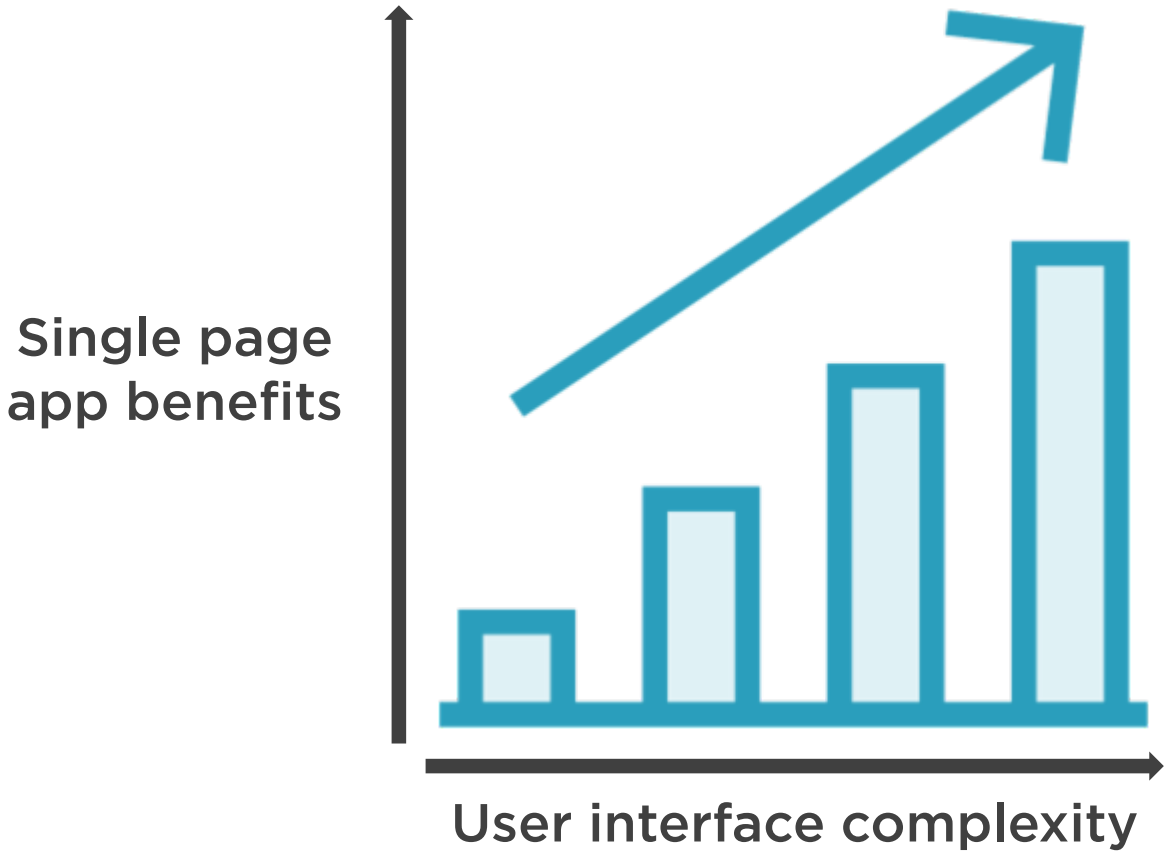
Traditional .NET Page Rendering



Blazor Page Rendering

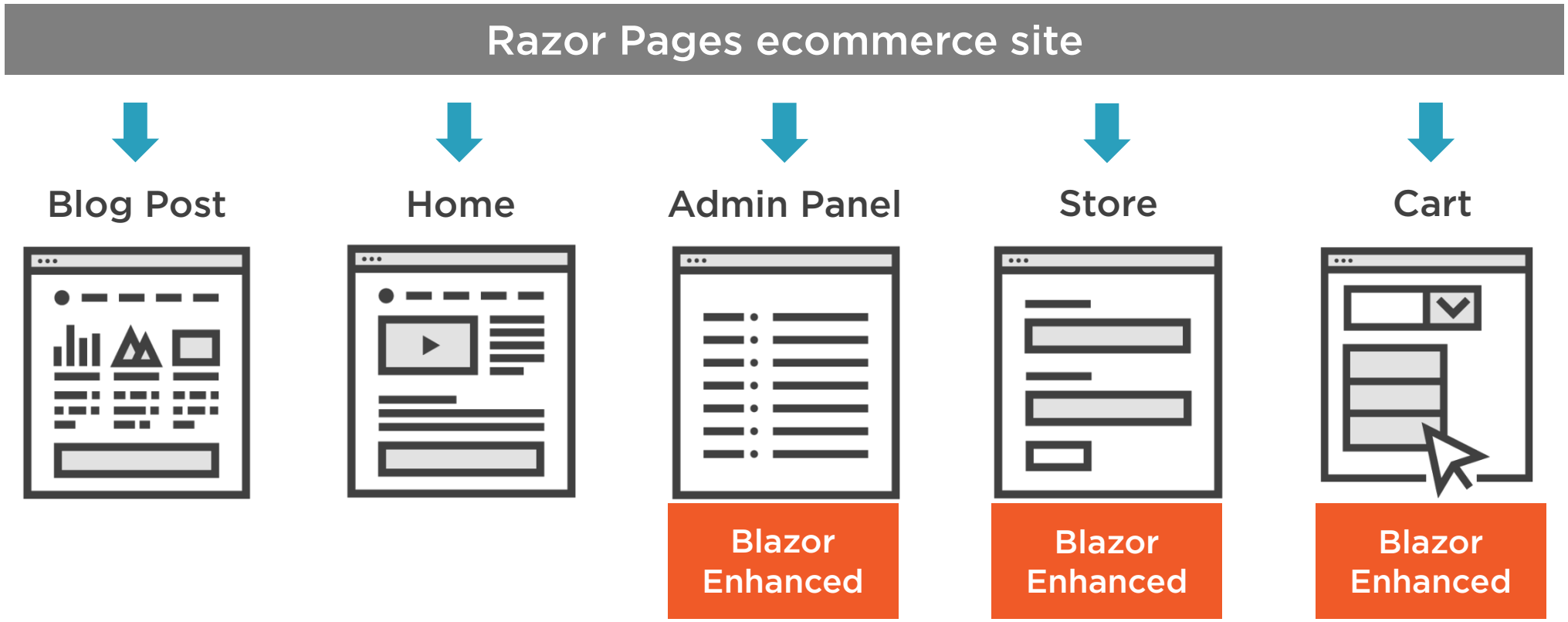


The Value of Single Page Apps



Complex user interfaces benefit from single page architecture.

Blazor Hybrid Applications



Browser Support for Client-Side .NET

Legacy .NET

Required plugins, poor mobile performance and integration

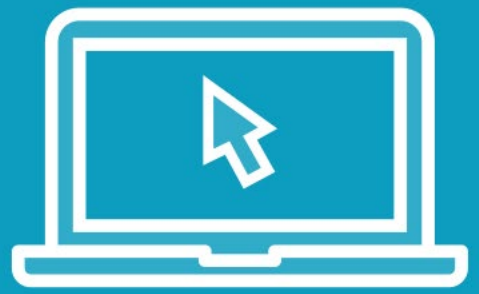
Blazor

Performant and supported natively in all major browsers

Benefits of Blazor

- Dependency injection
- Hosting options
- Standardized packages
- JavaScript interop

Demo



Setting up the sample application

Bethany's Pie Shop HRM

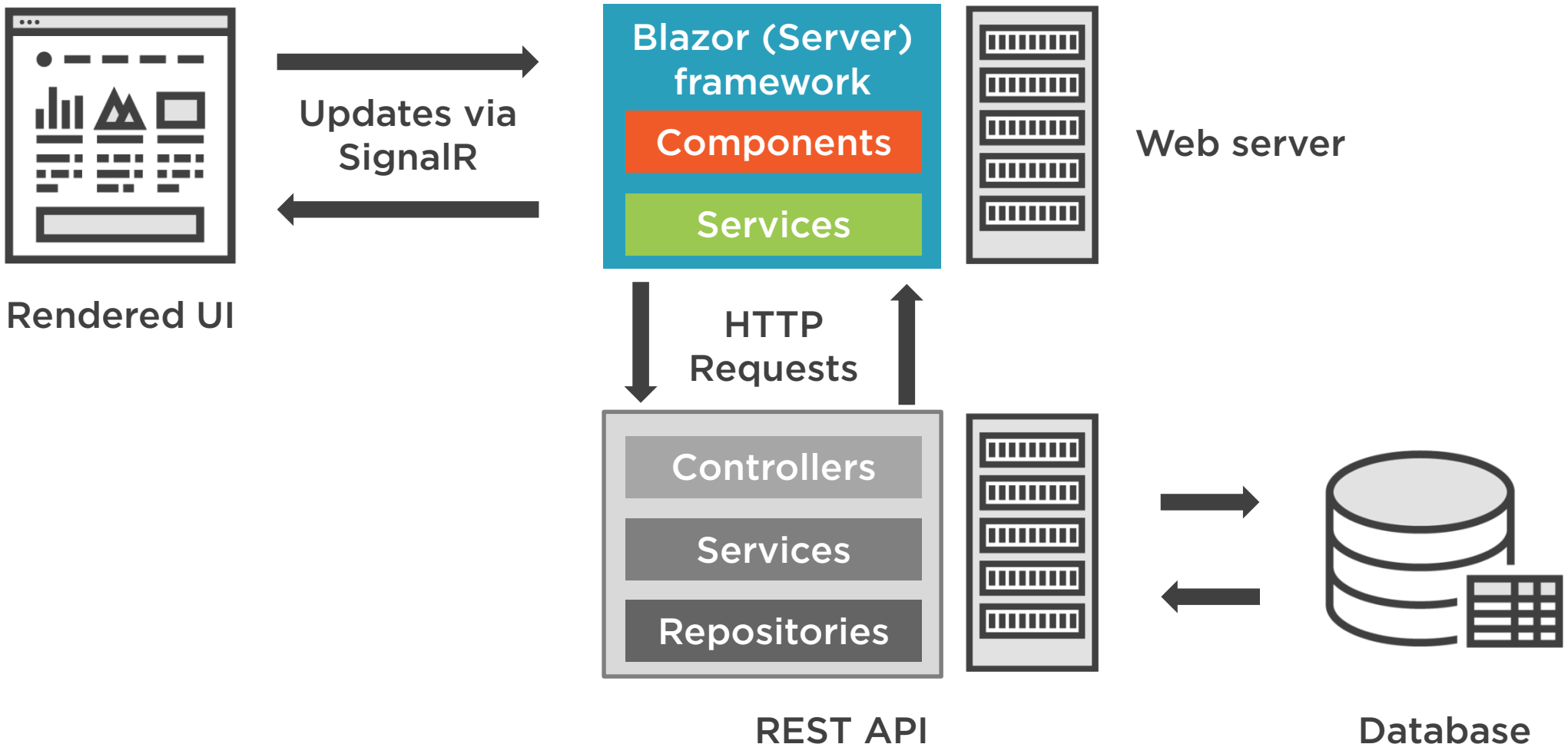
- Onboard new employees
- Submit expense reports
- Manage job openings
- Conduct reviews
- Much more

GitHub Link

https://github.com/alex-wolf-ps/blazor-enterprise

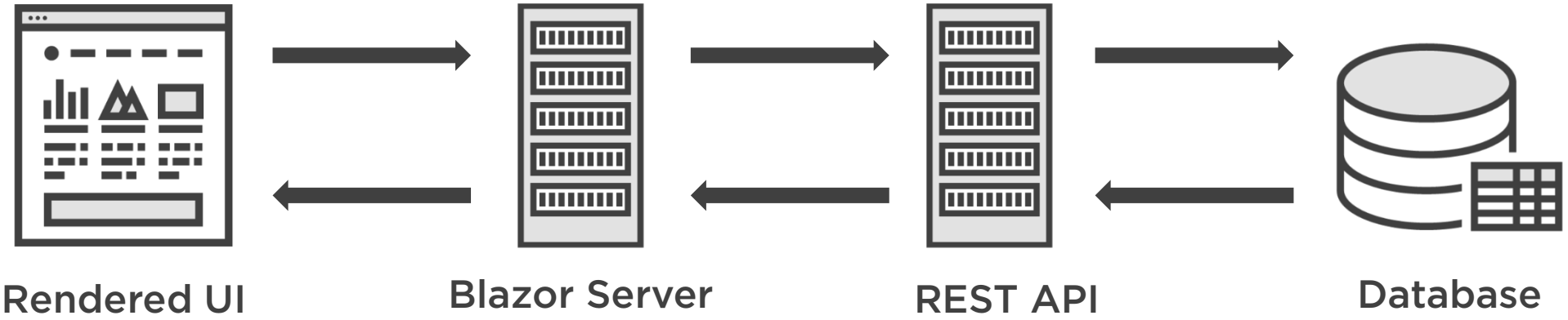
Understanding the Application Architecture

A Sample Blazor Server Architecture

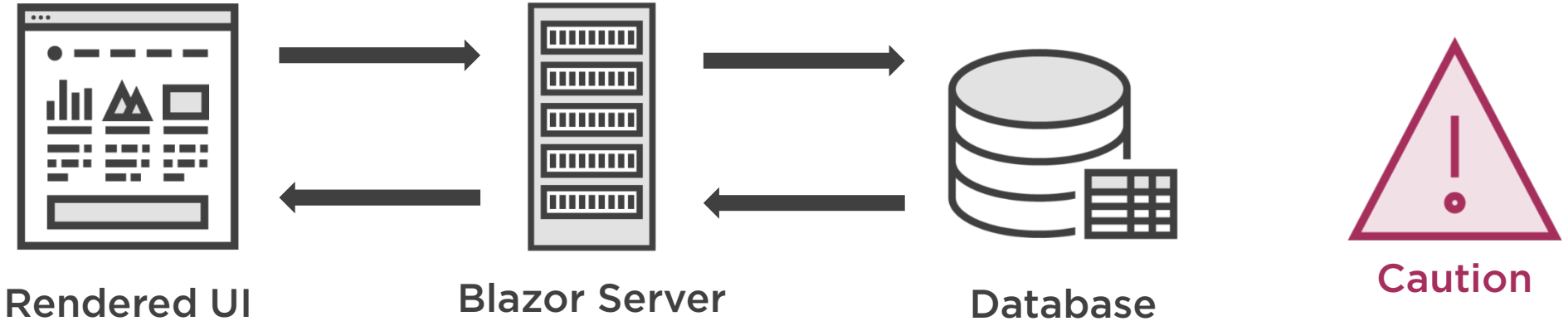


Exploring Blazor Server Options

Option 1: Separation of UI and data logic

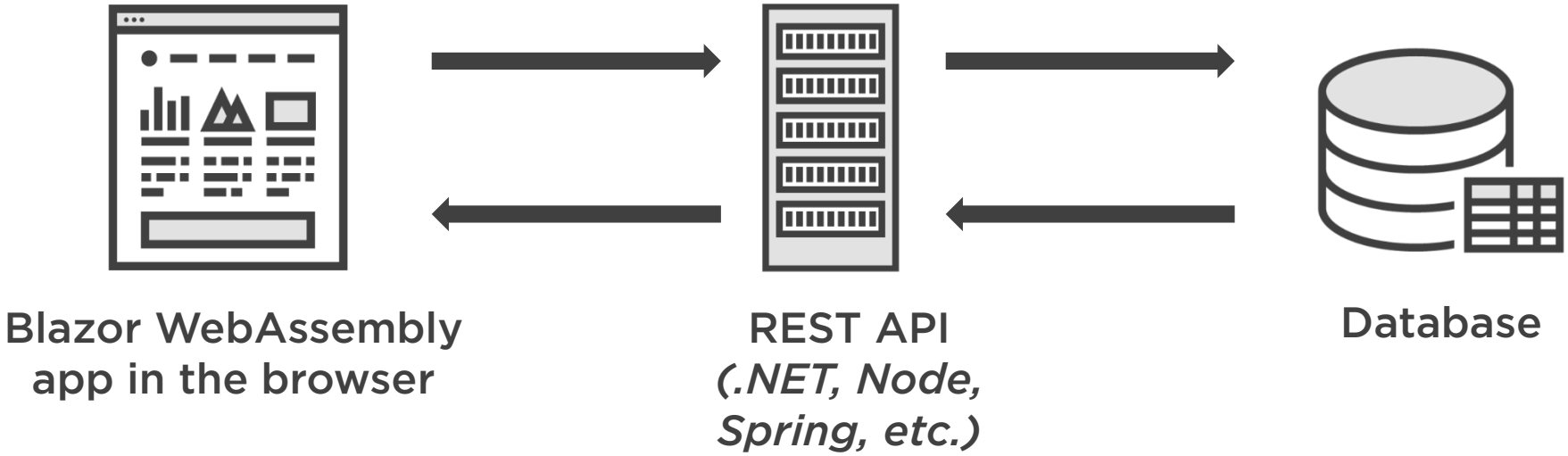


Option 2: Combined UI and data logic

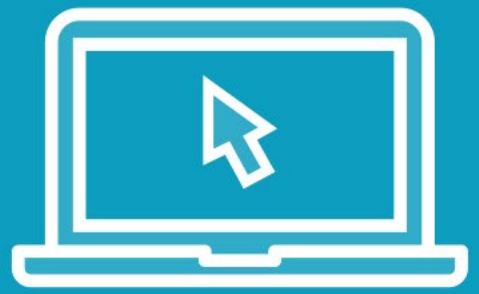


Blazor is a UI framework - use caution when architecting server apps.

Exploring Blazor WebAssembly Architecture

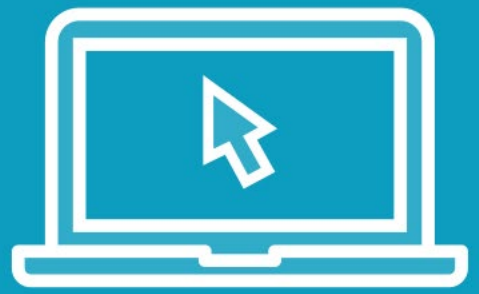


Demo



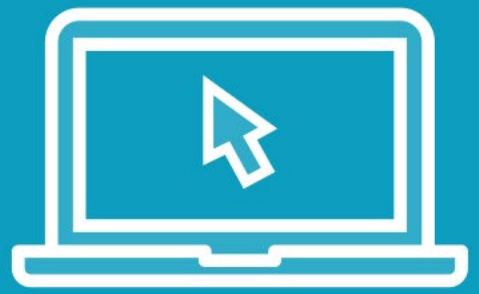
Exploring the sample application

Demo



Refactoring a page using Components

Demo



Improving the Component's reusability

Working with Dependency Injection and Application State

Alex Wolf

www.crywolfcode.com

A simple approach to Dependency Injection.

Dependency Injection

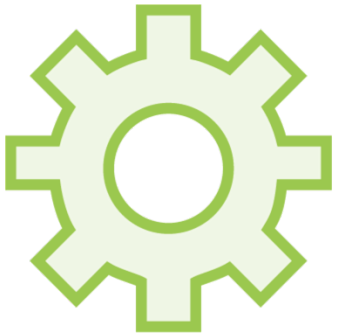
A design pattern that implements Inversion of Control principles to create loosely coupled code.

Abstractions vs Implementations

Abstractions:
Interfaces

IEmailService

Implementations:
Classes



OutlookMailService

A Simple Dependency Injection Example

Decoupling Components

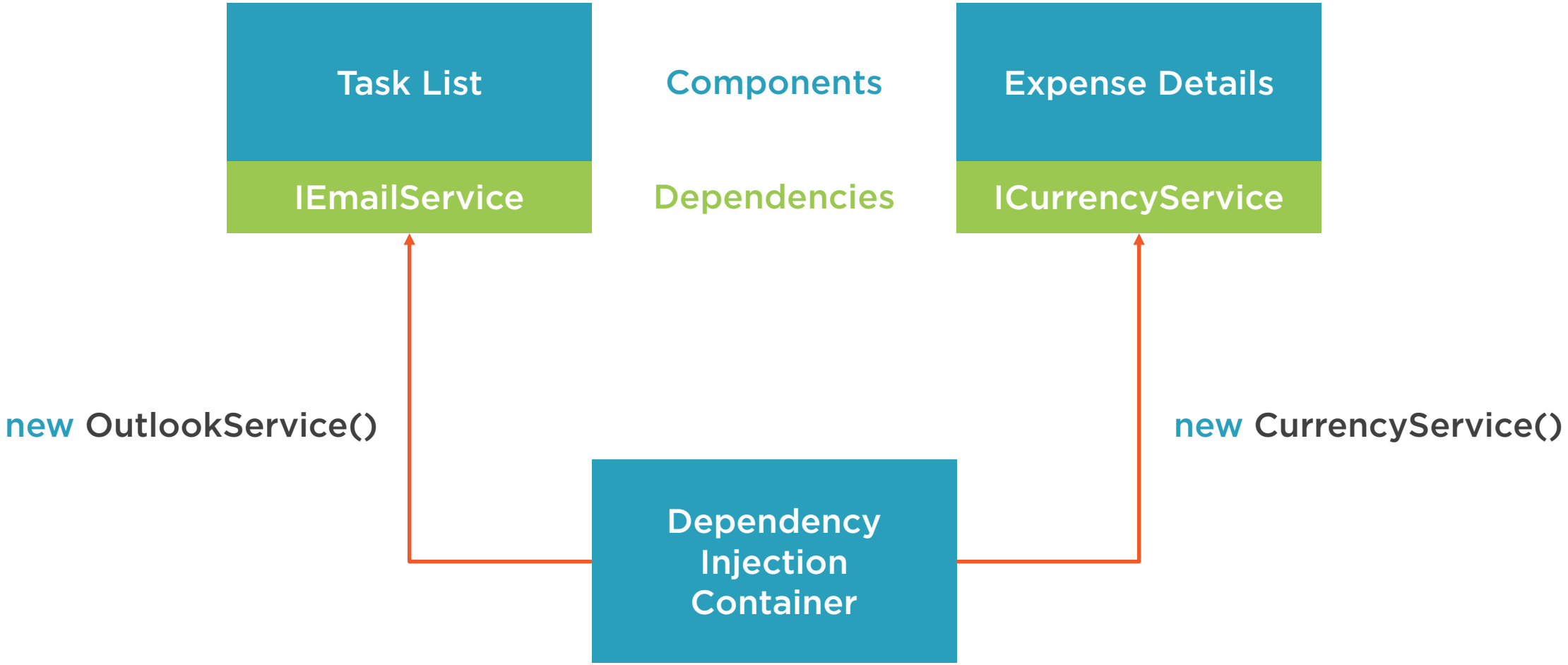
Tightly Coupled Component

```
<h1>Hello World</h1>

@code {

var mailer = new OutlookMailService()

mailer.SendEmail();

// Component Code

}
```

Loosely Coupled Component

```
[Inject]

public IEmailService EmailService
{ get; set; }

public class TaskComponent

{

EmailService.SendEmail();

// Component code

}
```

The Dependency Injection Container



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IEmailService, OutlookService>();
    services.AddTransient<IExpenseService, ExpenseService>();
}
```

Configuring Dependency Injection Containers

Dependencies are registered in Startup.cs using the service collection

We generally bind an interface type to a class implementation type

Providing Dependencies

Task List Component

```
[Inject]
public IEmailService EmailService { get; set; }
```

Hey, I depend on this Email contract

Okay, here is an implementation

Dependency Injection Container

```
services.AddTransient<IEmailService, OutlookEmailService>();
```

Dependency Injection with Blazor

Registering Dependencies in Blazor Server

startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMailService, OutlookService>();
}
```

Registering Dependencies in Blazor WebAssembly

program.cs

```
public static async Task Main(string[] args)
{
    var builder =  WebAssemblyHostBuilder.CreateDefault(args)
    builder.services.AddTransient<IMailService, OutlookService>();
}
```

Injecting Services into Components

Use Attributes and Directives

Single Markup File

```
@inject IEmailService EmailService

<p>Hello</p>

@code {

    EmailService.SendEmail();

    // Component Code

}
```

Base Class File

```
[Inject]
public IEmailService EmailService
{ get; set; }

public class TaskComponent

{

    EmailService.SendEmail();

    // Component code

}
```

```
public class ExpenseService : IExpenseService
{
    private IEmployeeService _employeeService;

    public ExpenseService(IEmployeeService employeeService)
    {
        _ employeeService = employeeService;
    }
}
```

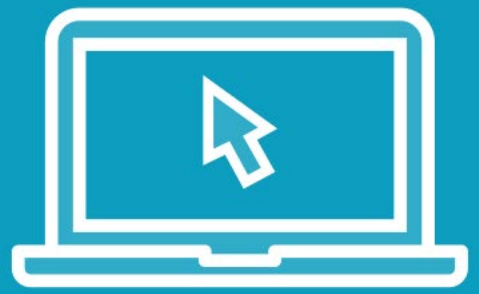
Constructor Injection

Constructor injection is common for complex classes that require dependencies

The [Inject] attribute does not work with services, only components

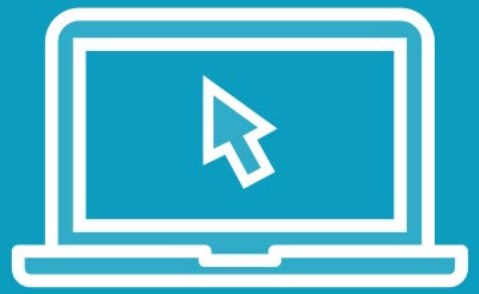
Why dependency injection?

Demo



Improving Components using dependency injection

Demo

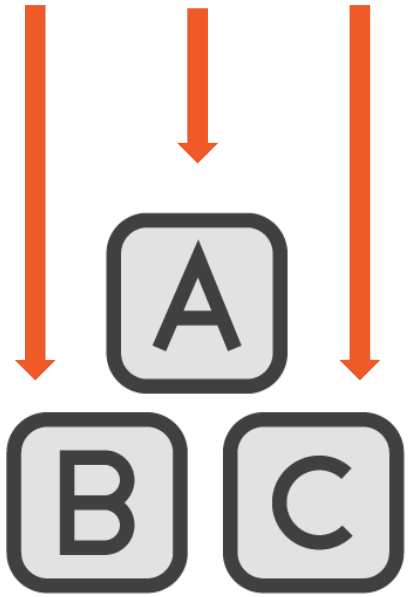


Using Dependency Injection with services

Benefits of Dependency Injection

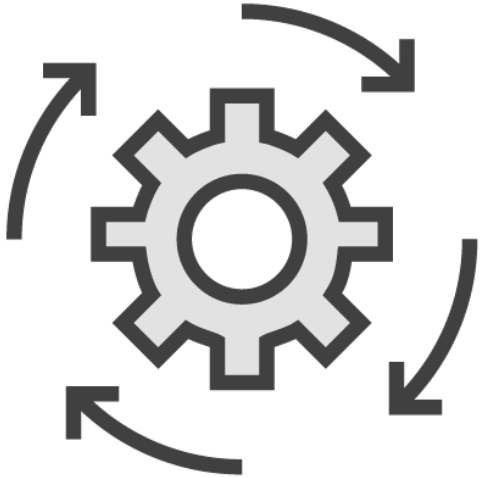
Benefits of Dependency Injection

Loose Coupling



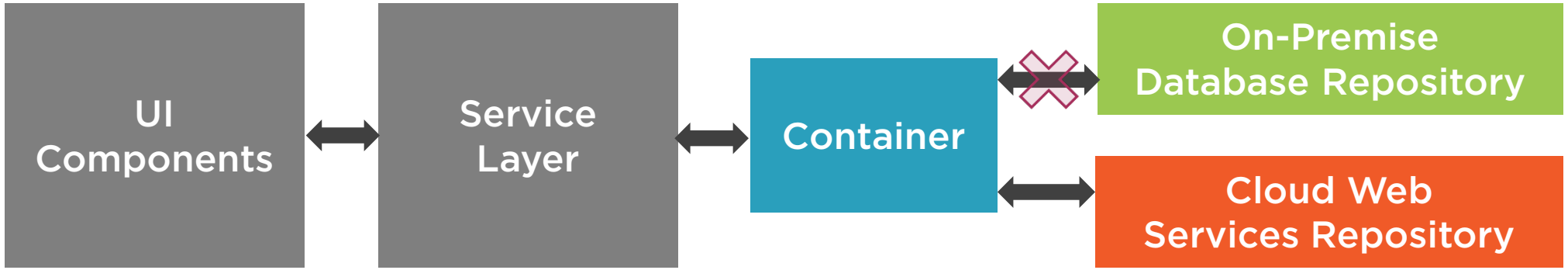
Improved Testing

Service Lifetime Management

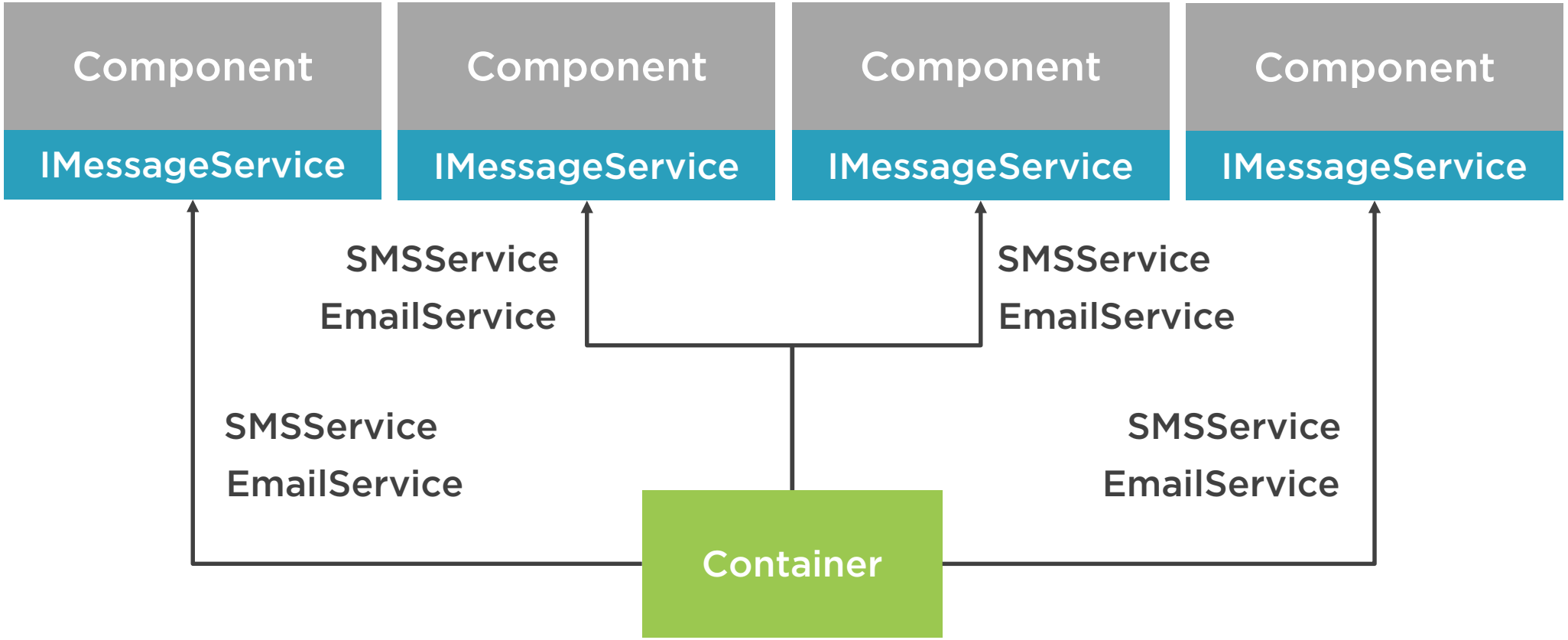


Changing Dependency Implementations

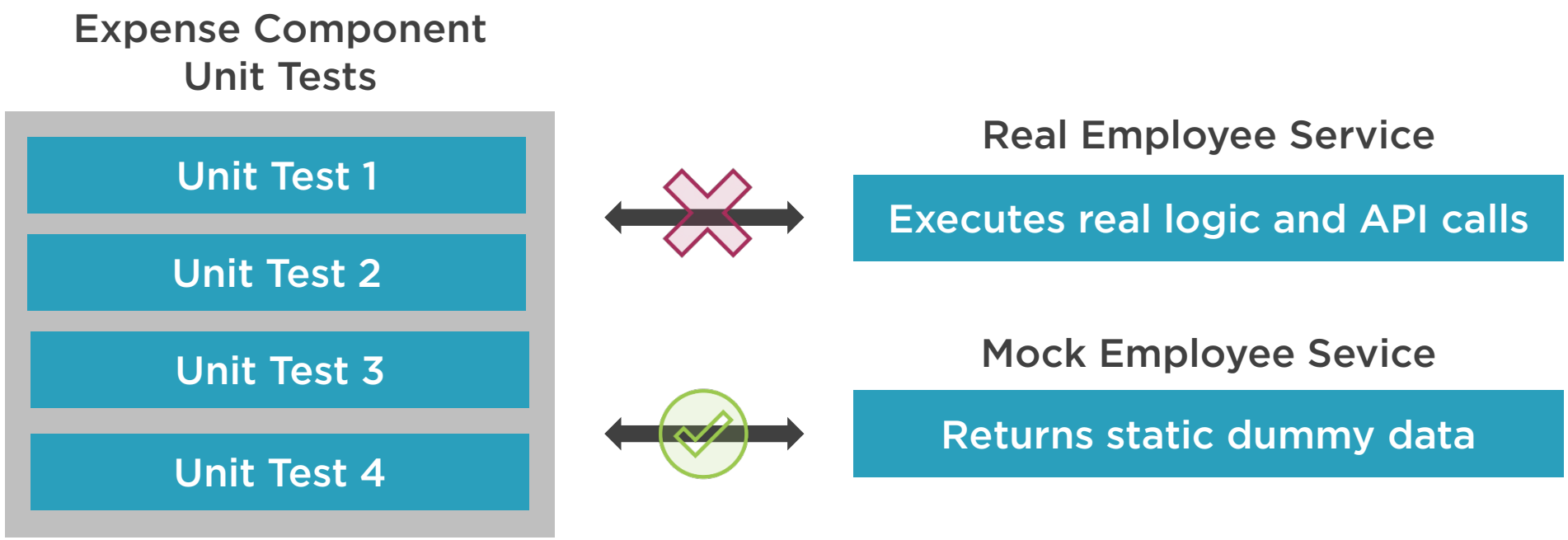
Loosely Coupled Application Layers



Centralized Dependency Management



Unit Testing with Dependency Injection



A note about dependency life times.

Framework Dependencies

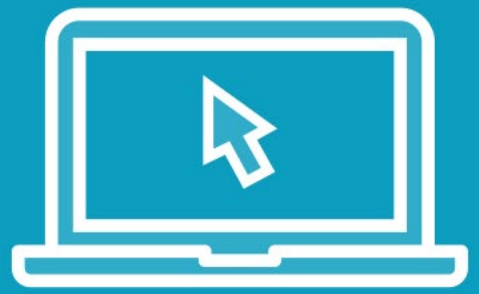
Navigation

Logging

HTTP

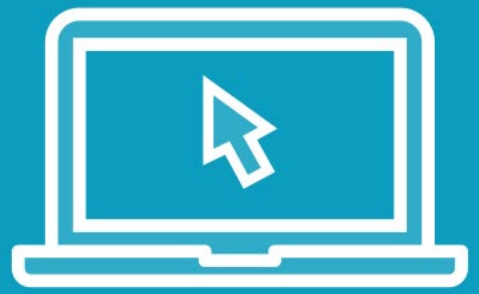
Much more!

Demo



Managing dependency implementations

Demo



Leveraging provided services

Understanding Service Lifetimes

Service Lifetime

Determines when a new instance of a dependency is created and helps to manage its scope.

Service Lifetimes in Blazor Server

AddTransient()

A new instance is created every single time it is needed

AddSingleton()

One instance is shared across all classes and HTTP requests

AddScoped()

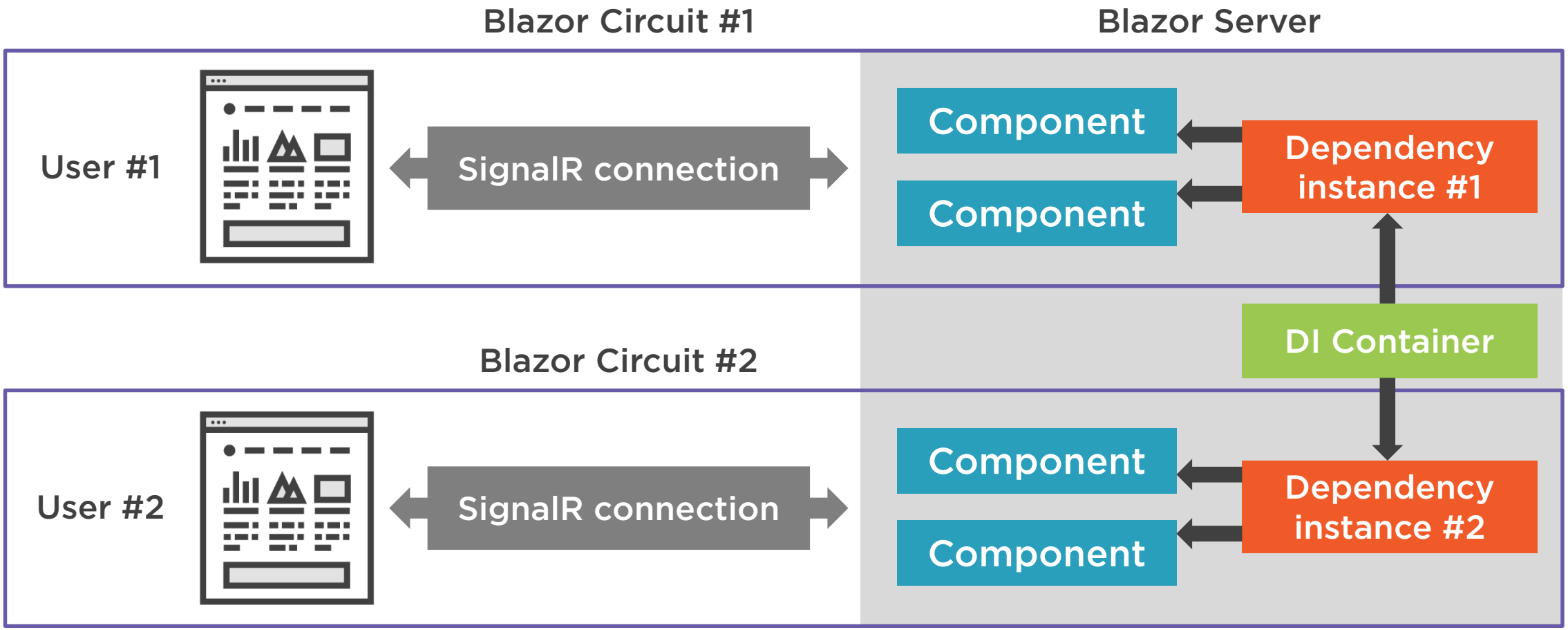
A new instance is created and reused for each Blazor Server circuit

Blazor Circuit

An abstraction over the SignalR Connection between the browser and server to manage state and scope.

(Blazor Server only)

Understanding Blazor Circuits



Service Lifetimes in Blazor WebAssembly

AddTransient()

A new instance is created every single time it is needed

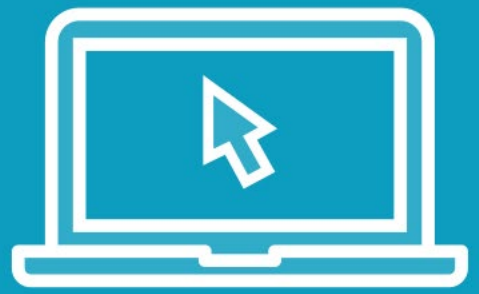
AddSingleton()

One instance is shared across all classes and HTTP requests

AddScoped()

Behaves the same as AddSingleton – one instance is reused

Demo



Working with Service Lifetimes

Summary

Dependency Injection allows us to build loosely coupled components

Interface abstractions allow for more testable, sustainable code

Blazor allows injection into components using Inject directives and attributes

We can also use standard constructor injection in services

Dependencies are registered in startup.cs

Dependency Injection allows us to easily swap or access service implementations

We can control service instantiation through registration methods

Enhancing the Application for the Enterprise

Alex Wolf

www.crywolfcode.com

The Agenda

Advanced routing concepts

Designing for reusability

Exploring application state

Understanding Blazor Server Routing

Routing in Blazor Server and WebAssembly

Blazor Server

Concepts and Code

==

Blazor WebAssembly

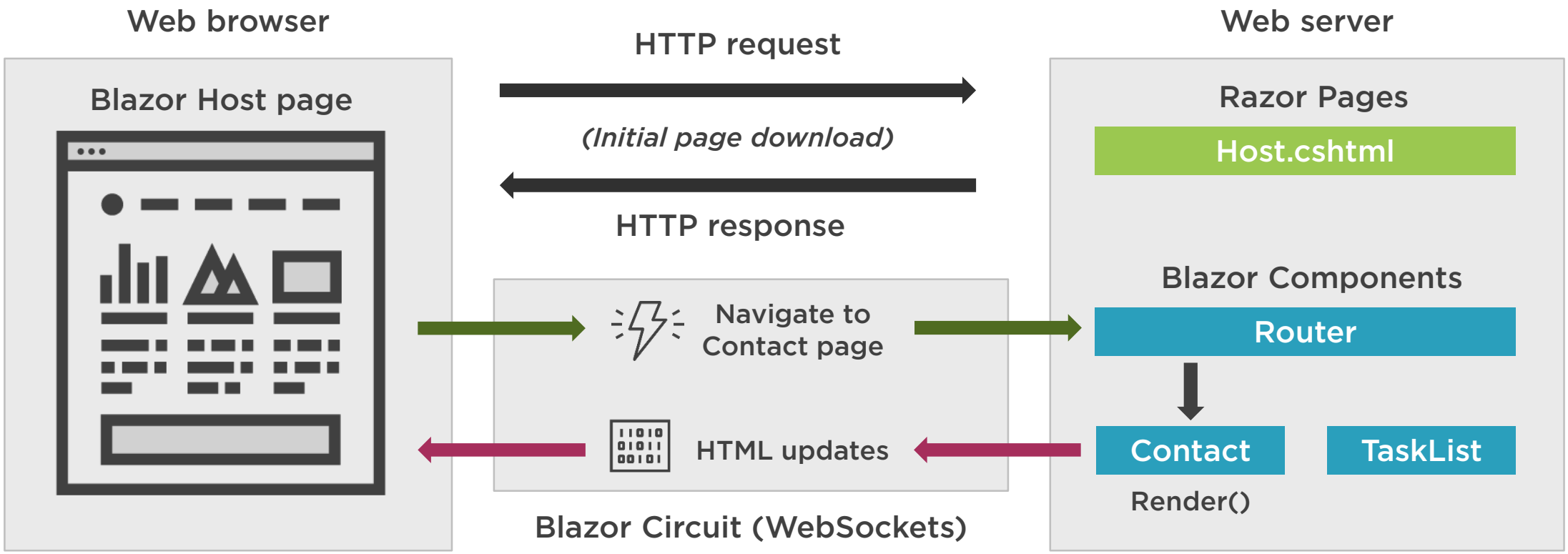
Blazor Server

Underlying Implementation

!=

Blazor WebAssembly

The Blazor Server Routing System



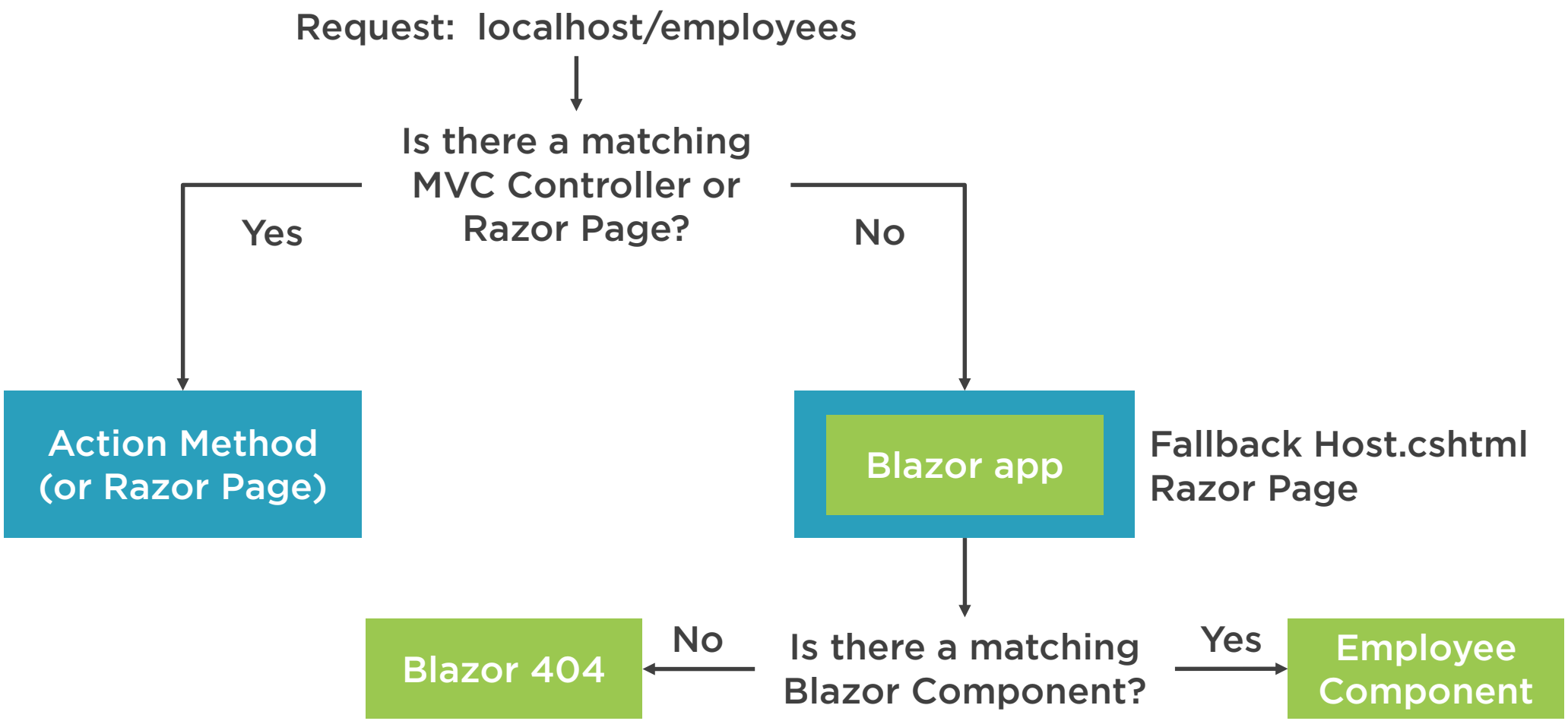
```
public void ConfigureServices(IApplicationBuilder app)
{
    app.UseEndpoints(endpoints => {
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
    });
}
```

Blazor Server Routing Configurations

MapBlazorHub sets the path for SignalR communication

MapFallbackToPage routes incoming traffic to our container host page with low priority

Understanding Server Routing Compatibility



```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
         <p>Sorry, there's nothing at this address.</p>
     </NotFound>
</Router>
```

The Router Component

Includes templates for the “Found” and “Not Found” routing scenarios

The RouteView component renders the selected component

Understanding Routable Components

Blazor Components

@page "/employee"

EmployeeComponent

ModalComponent

@page "/contact-us"

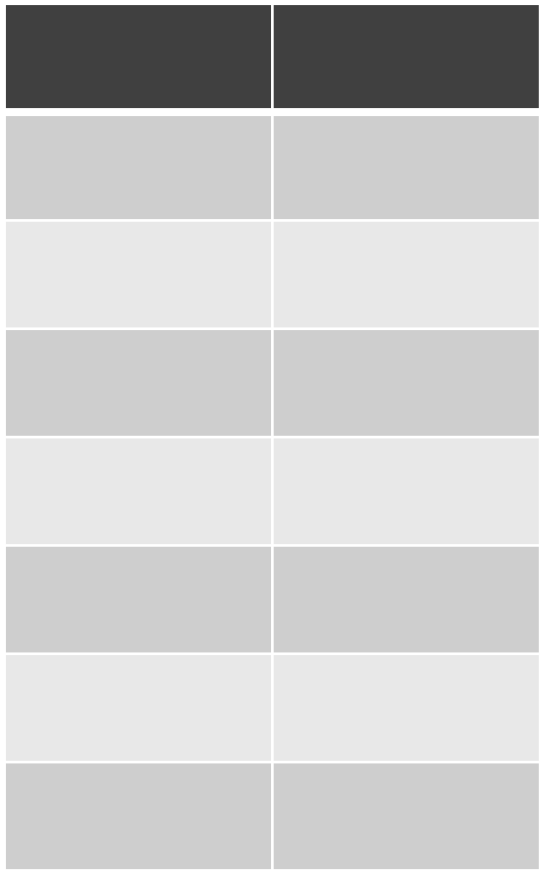
ContactComponent

MapComponent

Added to table

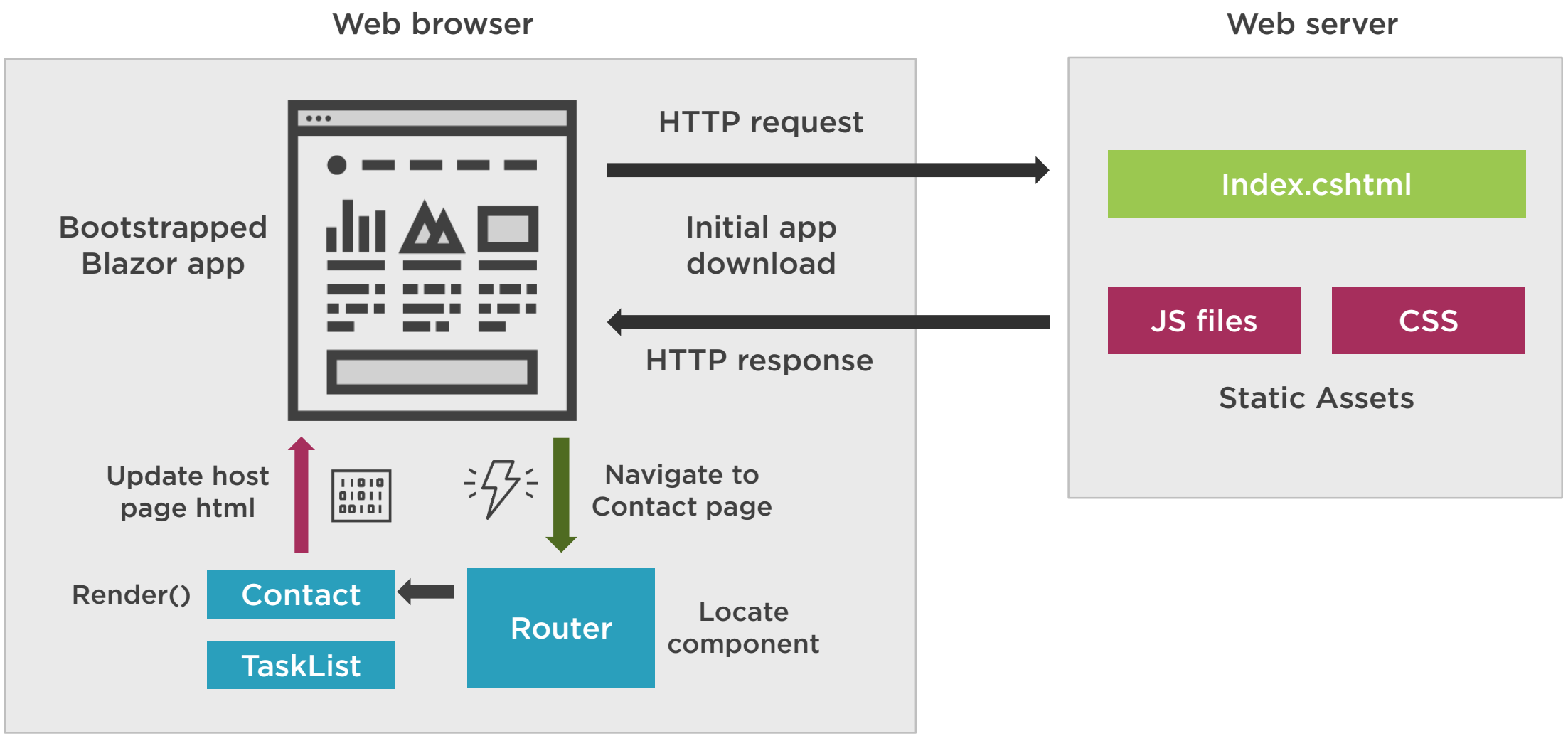
Added to table

Route Table



Understanding Blazor WebAssembly Routing

The Blazor WebAssembly Routing System



WebAssembly Configuration Differences

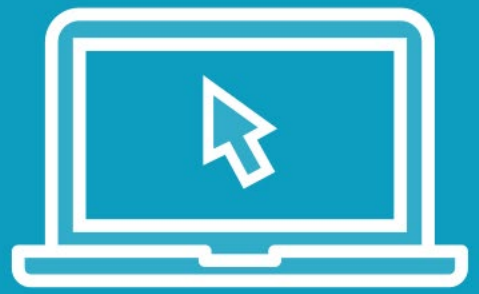
Filename.code

```
ConfigureServices(IApplicationBuilder app)
{
    // None of this is needed!
    app.UseEndpoints(endpoints => {
        endpoints.MapBlazorHub();

        endpoints.MapFallbackToPage("/_Host");
    });
}
```

No SignalR hub is needed – no circuits

No fallback is needed for MVC or Razor Pages

Demo



Touring the Blazor routing system

Exploring Additional Routing Features

```
@page "/editemployee"
@page "/editemployee/{id}"
@inherits EmployeeEditBase

// Component content
```

Component Route Templates

Blazor route templates allow parameters, but not optional parameters

We can work around this by using multiple templates

Introducing Blazor Route Constraints

Constraints enforce the data types of Route segments.

EmployeeEdit.razor

```
@page “/editemployee/{id:int}”
// Enforces the ID is an integer

<div class=“component">
    // Component content
</div>
```

BlogPost.razor

```
@page “/post/{filter:datetime}”
// Enforces the filter is date

<div class=“component">
    // Component content
</div>
```

Available Route Constraints

Constraint Type	Example Match
:bool	true, false
:datetime	2019-11-30
:decimal	24.99
:double	1.55
:float	1.60
:guid	FD3E1333-2341-35F6-3344-LIVEKLKJ5421
:int	123
:long	13234
Custom	unsupported

app.razor

```
<Router AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    // Enable routing to components in another assembly
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout">
            // Friendly layout styling
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

```
<NavLink href="/tasks" Match="NavLinkMatch.All">All Tasks</NavLink>

<NavLink href="/task/edit/1" Match="NavLinkMatch.Prefix">Edit Task</NavLink>
```

Revisiting the NavLink

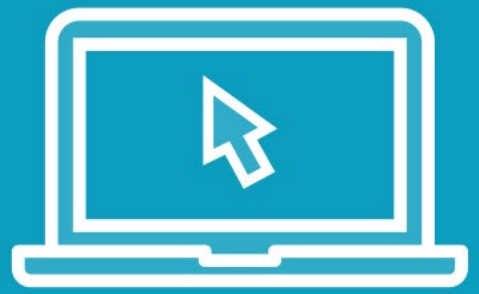
Automatically assigns active CSS class to links that match the current URL

Provides a wrapper around anchor tags that allow us to pass in attributes

A Closer Look at the NavigationManager

Class Member	Description
Uri	Gets the absolute URI
BaseUri	Gets the base URI that relative URI paths can be appended to
NavigateTo()	Navigates to the provided URL
LocationChanged	An event that fires when the URL changes
ToAbsoluteUri()	Converts a relative URI to an absolute URI
ToBaseRelativePath()	Converts an absolute URI to a relative URI

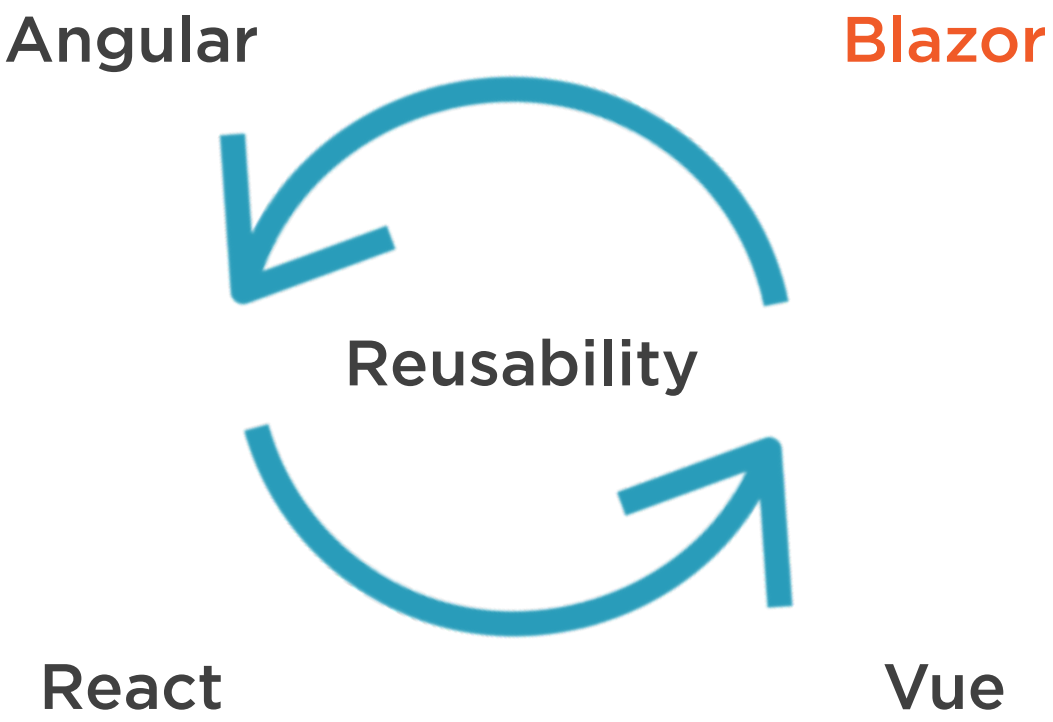
Demo



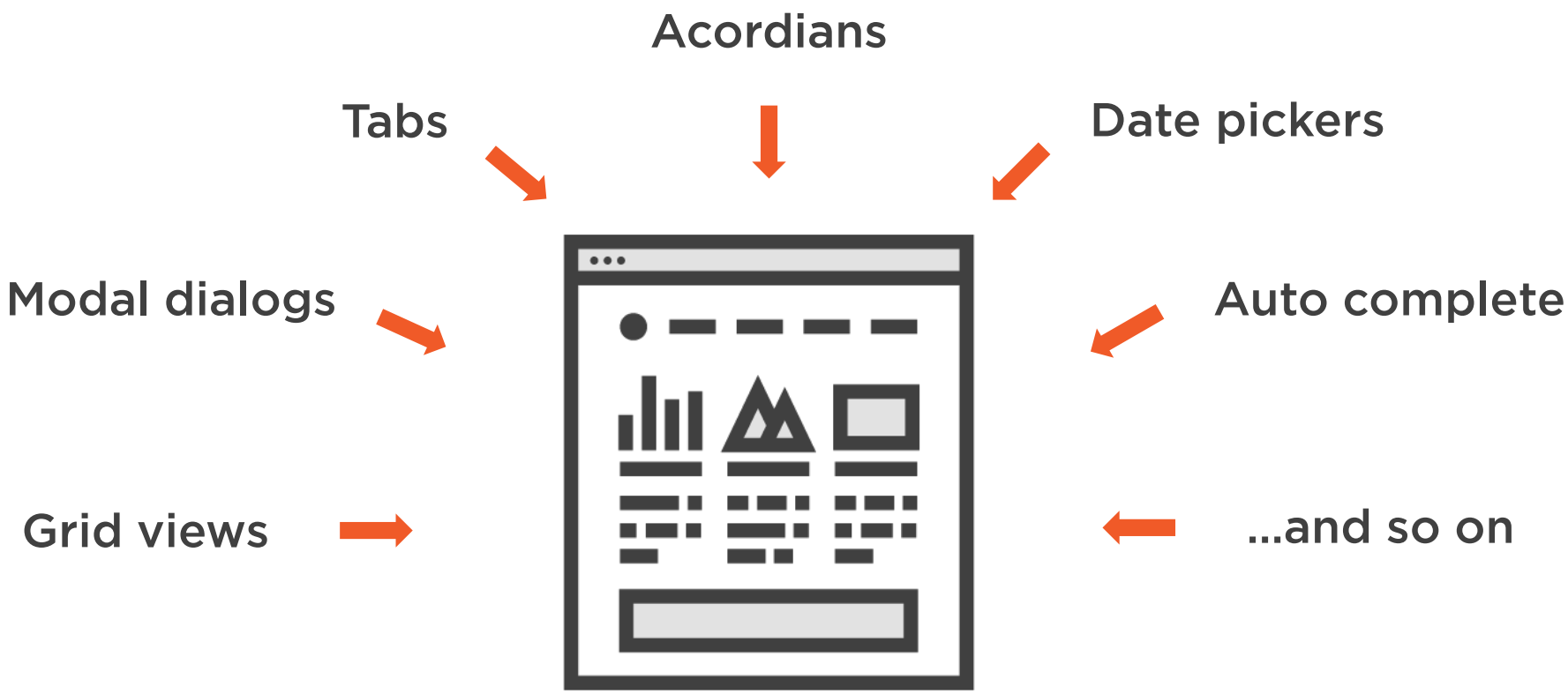
Enhancing the application's routing features

Understanding Component Reusability

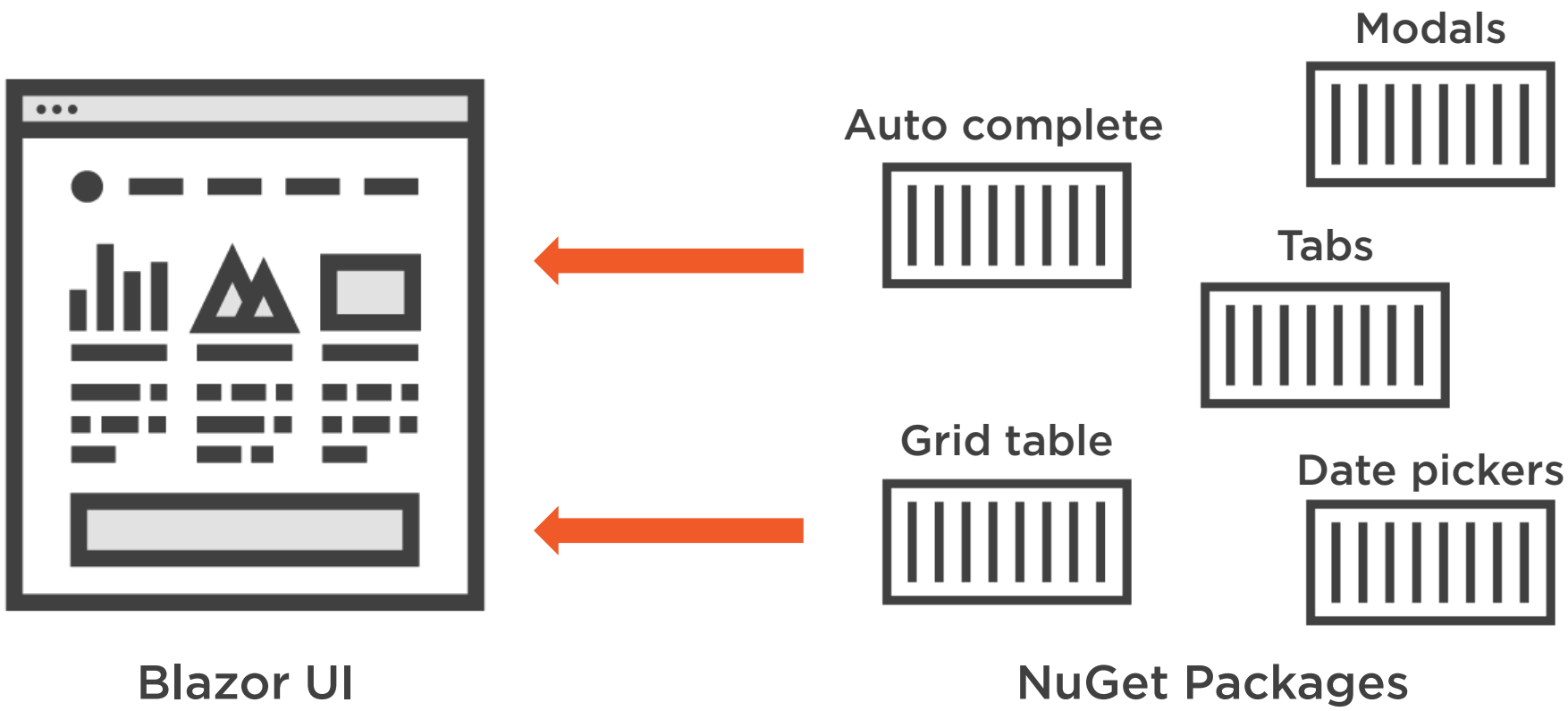
Designing for Reusability



Many apps...similar components.



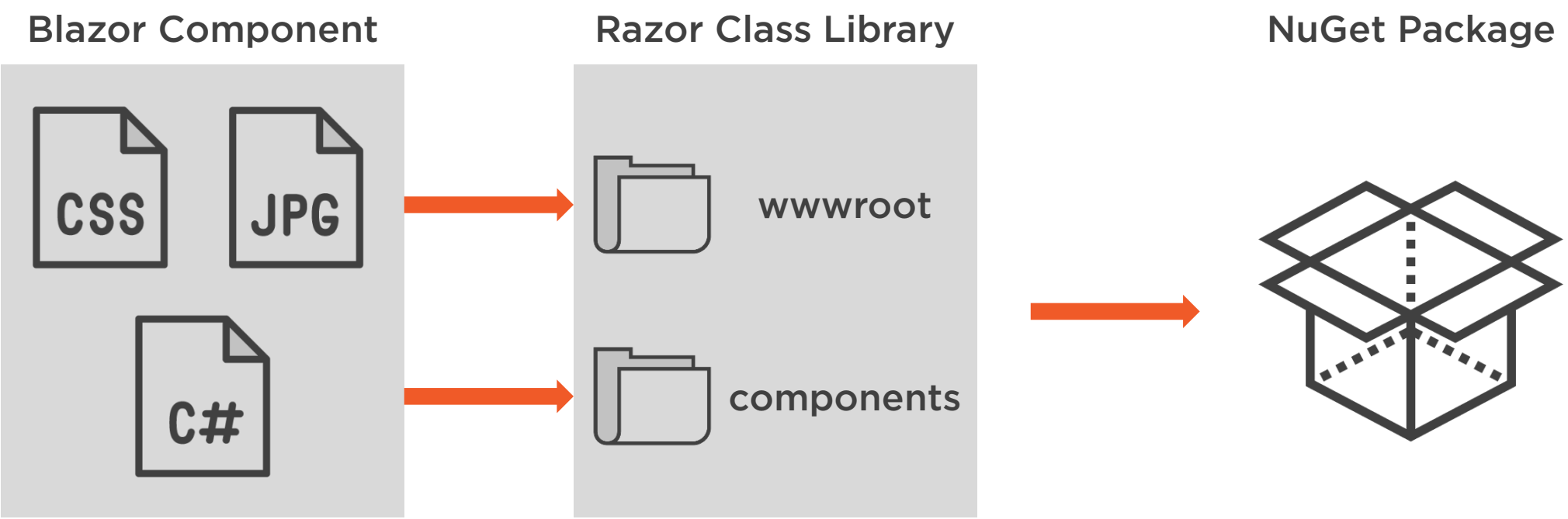
Revisiting NuGet Packages



Razor Class Libraries

A project template designed for easily distributing Blazor and Razor based UI components.

Working with Razor Class Libraries



Referencing Components from a Nuget Package

_host.cshtml

```
<link rel="stylesheet"
href="_content/PieShopLib/styles.css"/>

<!-- CSS sheet for the map component in
our NuGet package -->
```

EmployeeDetail.razor

```
@page "/employee/{id:int}"

@using PieShopLib


<div class="component">

    <map zoom="10"></map>

     <!-- Map Component -->

</div>
```

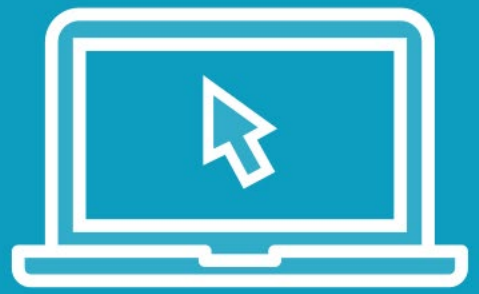
Package Use Cases

Create Component packages

Consume UI Component packages

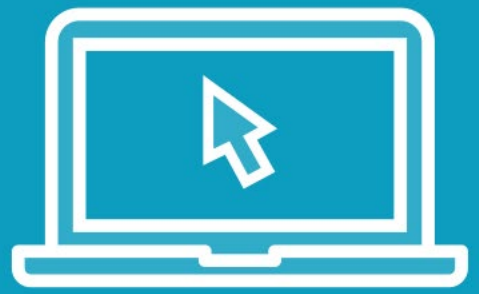
Consume functional packages

Demo



Distributing our components as NuGet packages

Demo



Enhancing our UI using NuGet

Exploring Application State

Application State Options

In-memory services

URL

Database

Browser storage

Storing State in the URL

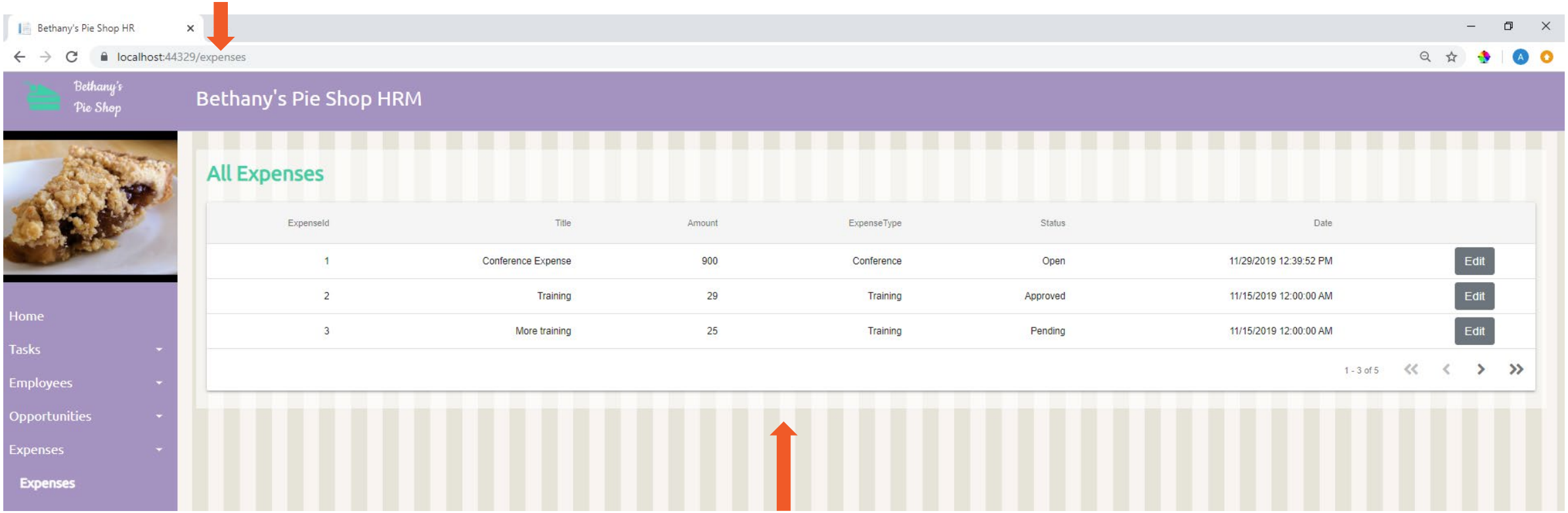
localhost/search?count=25&filterby=date&category=new

The page to load

Result filtering

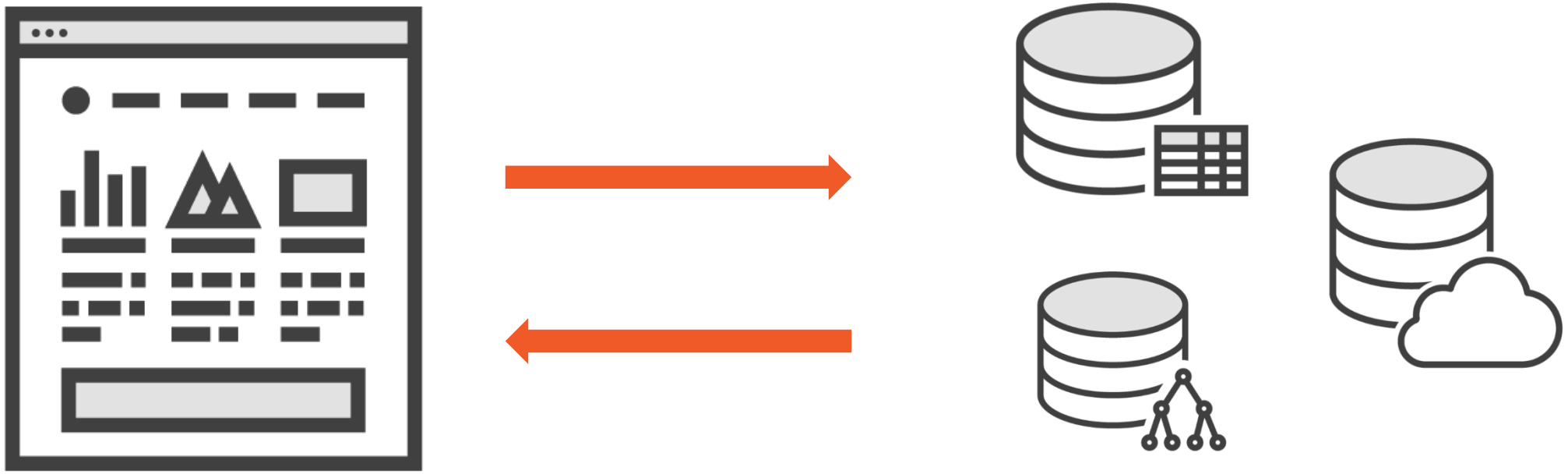
Single Page App State Considerations

... but URL stays the same



Grid updates...

Managing State with Databases



Blazor UI

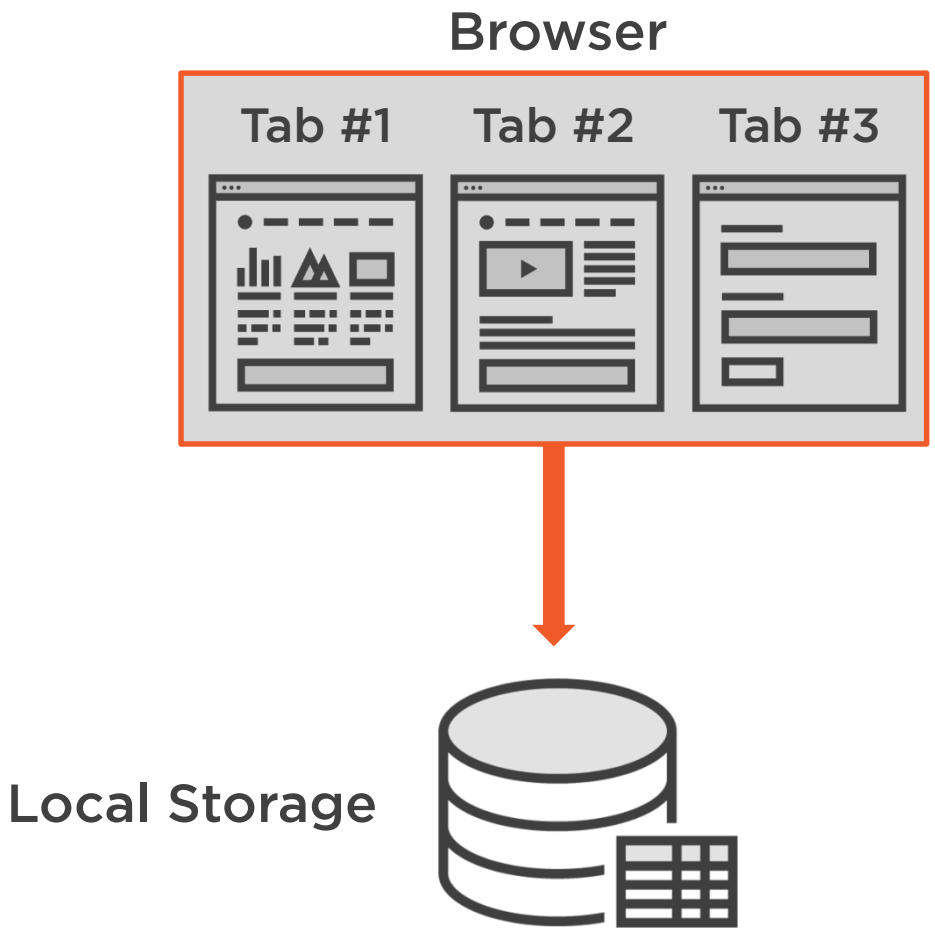
Database technologies

Managing State in the Browser

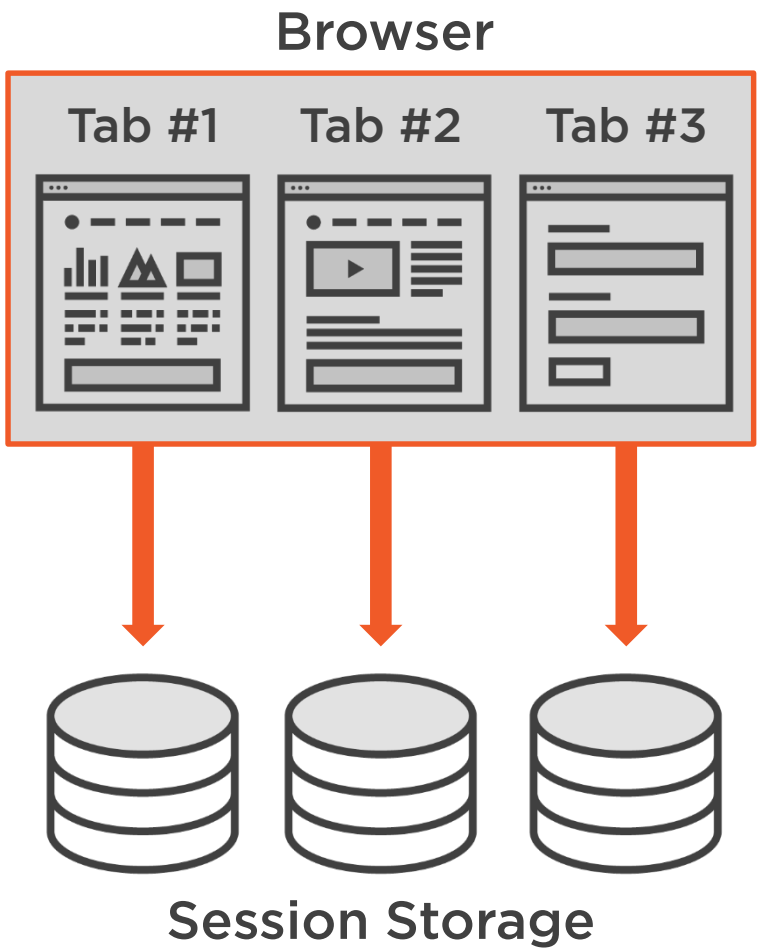
Local Storage

Session Storage

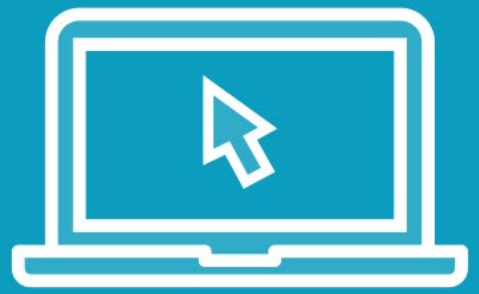
Understanding Browser Local Storage



Understanding Browser Session Storage



Demo



Managing application state using browser storage

Summary

The Blazor Router component handles mapping URLs to components

Route Constraints allow us to enforce the data types of URL segments

The Navigation Manager can programmatically influence navigation

We can use NuGet packages to enhance the UI of our application

Razor Class Libraries are useful templates for distributing Blazor components

We can use Browser Storage to manage small amounts of state with Blazor

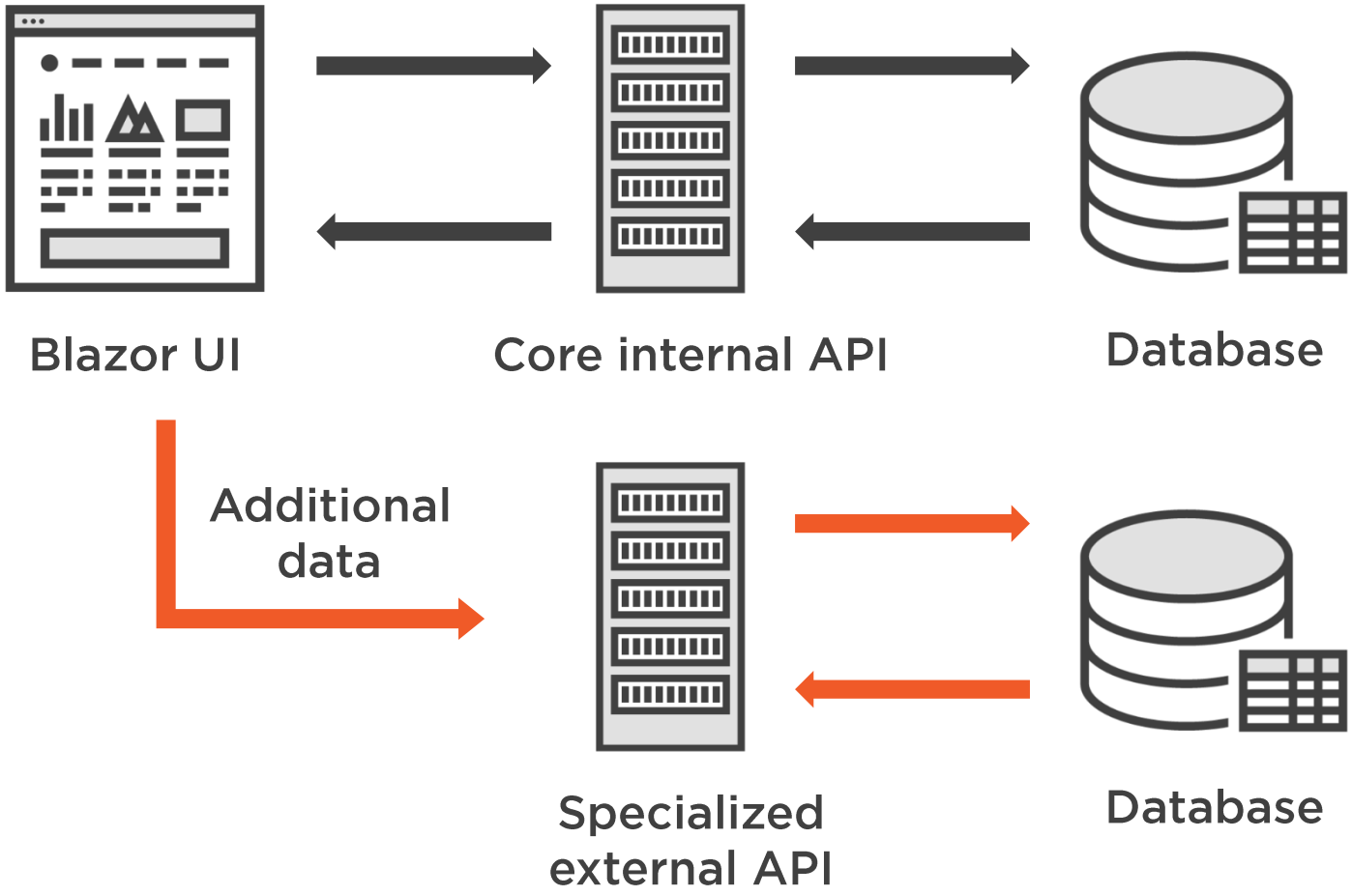
HTTP Communication with Blazor

Alex Wolf

www.crywolfcode.com

A note about HTTP and RESTful Web Services.

One Application, Multiple APIs



Calling a Web API with Blazor

HttpClient

The main class used by Blazor to send HTTP Requests.

Understanding HTTPClient Options

Blazor WebAssembly

HttpClient is ready to go with default app settings.

Blazor Server

HttpClient requires some basic configuration.

A note about HttpClient.

HttpClientFactory

A class that manages HttpClient instances and addresses common problems associated with them.

HttpClientFactory Options

- Basic clients
- Named clients
- Typed clients
- Generated clients

Working with HttpClient and Typed Services

JobsDataService.cs

```
private HttpClient _httpClient;


public JobsDataService(HttpClient
client)

{

    _httpClient = client;

}
```

startup.cs

```
var baseUrl = new Uri("localhost:5001")

services.AddHttpClient<IJobsDataService,
JobsDataService>(client =>

{

    client.BaseAddress = baseUrl;

});
```

Working with HttpClientFactory Directly

An option for isolated scenarios that don't require their own service.

JobEditBase.cs

```
[Inject]
public IHttpClientFactory _httpClientFactory;

// Use this client to send requests
var client = _httpClientFactory.CreateClient();
```

Making Requests with HttpClient

Default Options

Make requests using the default methods of the class.

Helper Libraries

Make requests using helper extension methods provided by libraries.

Demo

Improving the setup of HttpClient

Demo

Retrieving data from the API

Demo

Sending data to the API

Demo

Customizing HTTP Requests

Summary

Blazor utilizes the HttpClient class to communicate with web services

HttpClientFactory provides an improved way of managing HttpClient instances

.NET provides various techniques for working with HttpClientFactory

Typed Clients are application services that depend on HttpClient

HttpClient provides various methods to make requests with different HTTP verbs

Blazor provides a helper library to streamline working with JSON requests

We can also send custom HTTP requests

Building Advanced Form Workflows

Alex Wolf

www.crywolfcode.com

The Agenda

Working with complex form data

Advanced form validation

Custom form inputs

Demo

Revisiting the new employee form

Demo

Working with complex form data

Going Further with Form Validation

Blazor Validation Essentials

Employee.cs

```
[Required]
[MaxLength(100)]
public string Notes { get; set; }

[Required]
public string Name { get; set; }
```

EmployeeEdit.cs

```
<ValidationSummary />
<!-- List of all validation errors -->

<InputText @bind-
Value="@Employee.Name"></InputText>
<ValidationMessage For="@(() =>
Employee.Name)" />
<!-- Field validation error -->
```

Types of Validation

Model validation

Validation rules that apply across multiple properties on the model

Property validation

Validation rules that apply to a single property

Understanding Model Level Validation

Employee.cs

```
public double Latitude { get; set; }

public double Longitude { get; set; }

public DateTime StartDate { get; set; }

public bool IsFTE { get; set; }
```

Longitude is required if Latitude has a value.

StartDate must be a Monday if the employee is full time.

Understanding Property Level Validation

Employee.cs

```
[Required]
public double Salary { get; set; }
```

Property must have a value.

```
[Required]
[Email]
public string Email { get; set; }
```

Property must have a value formatted as an email.

Custom Property Validation using Attributes

CustomValidator.cs

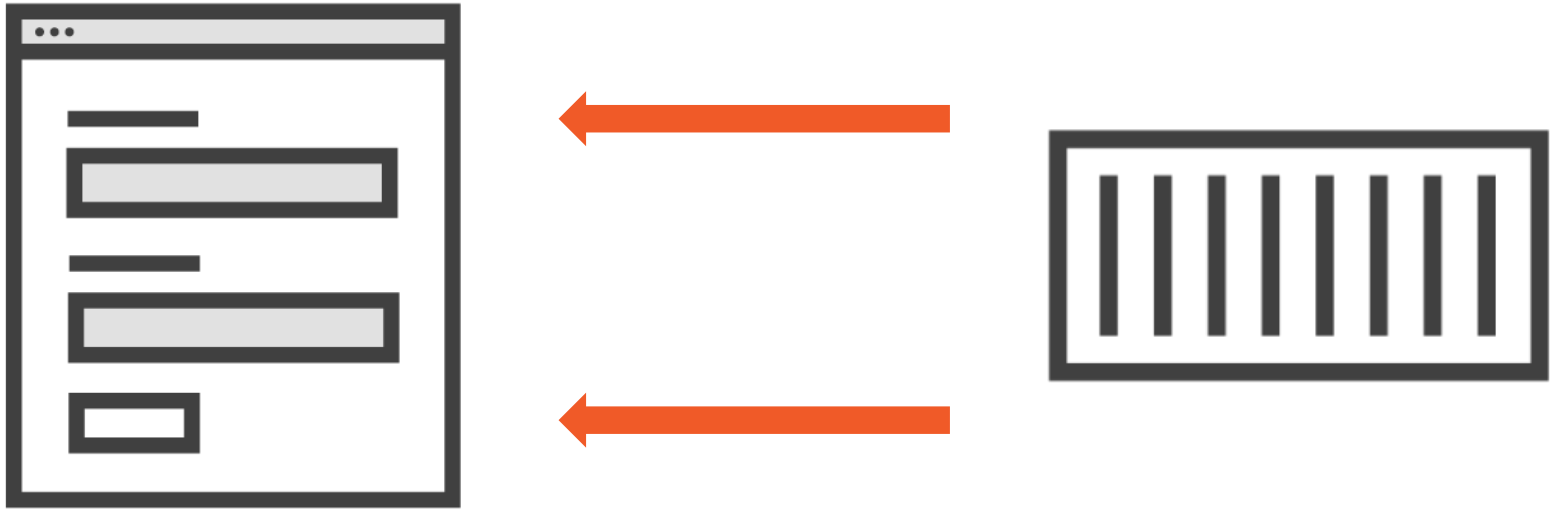
```
public class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        // Custom property validation logic
    }
}
```

Custom Model Validation using IValidatableObject

CustomValidator.cs

```
public class Employee : IValidatableObject

{

    public IEnumerable<ValidationResult> Validate(ValidationContext
        validationContext)

    {

        // Custom model validation logic

    }

}
```

Extended Blazor Validation



Complex Blazor form

Blazor validation
NuGet package

Demo

Implementing complex model validation

Demo

Creating a custom validation attribute

Demo

Adding custom model validation

Demo

Exploring a use case for custom inputs

Demo

Building a custom form input

A note about custom inputs.

Summary

Blazor supports form data models with complex properties and collections.

The Blazor Validation NuGet package can help with complex validation issues.

Custom Validation Attributes can apply specific business rules to properties.

The IValidatableObject allows for custom model level validation.

The InputBase class provides a starting point for building custom form inputs.

Thank you, and good luck!