Project Architecture

Web Applications Designs and Architectures, Repository Pattern, Automapper, Databases and ORM

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

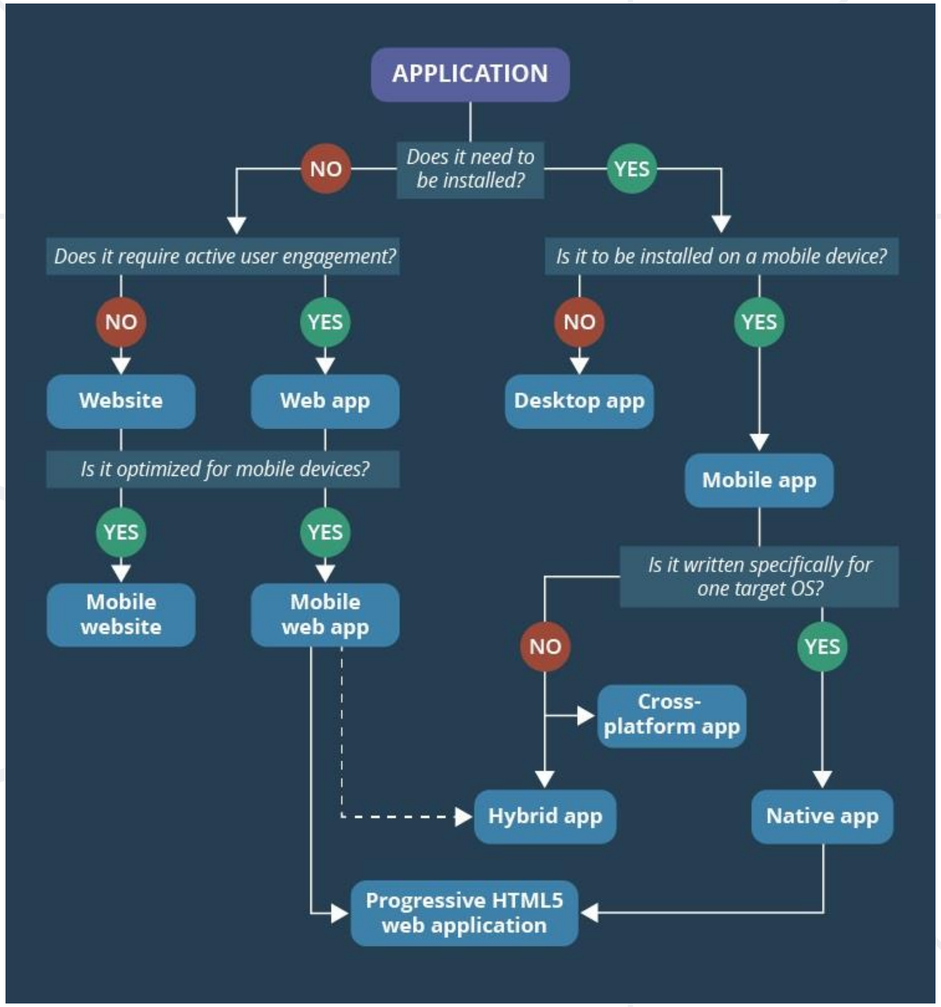
sli.do

#csharp-web

Web Application Designs

Web vs Desktop vs Mobile vs IoT

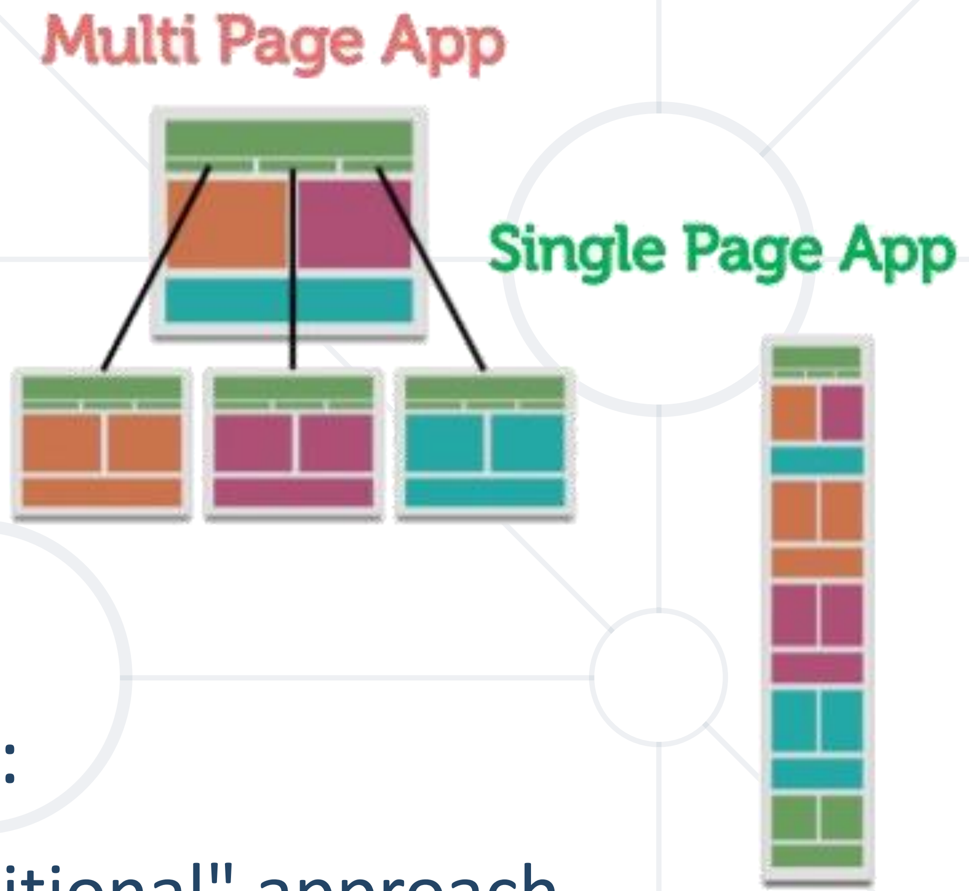
- **Desktop Application**
  - PRO: Can work offline, Has access to system resources
  - CON: Needs to be installed (updated) on each computer
- **Mobile Application**
  - PRO: App stores, Offline, Access to system resources
  - CON: Different platforms, Each update requires approval
- **Web Application**
  - PRO: No need to be downloaded, installed or updated
  - CON: Require Internet, Limited system access
- **Internet-of-Things Application**
  - Smart home, wearables, cars, farming, cities, etc.
  - They require web access to send their data



APPLICATION
Does it need to be installed?
NO
YES
Does it require active user engagement?
Is it to be installed on a mobile device?
NO
YES
NO
YES
Website
Web app
Desktop app
Is it optimized for mobile devices?
Mobile app
YES
YES
Mobile website
Mobile web app
Is it written specifically for one target OS?
NO
YES
Cross-platform app
Hybrid app
Native app
Progressive HTML5 web application

Web Application Designs

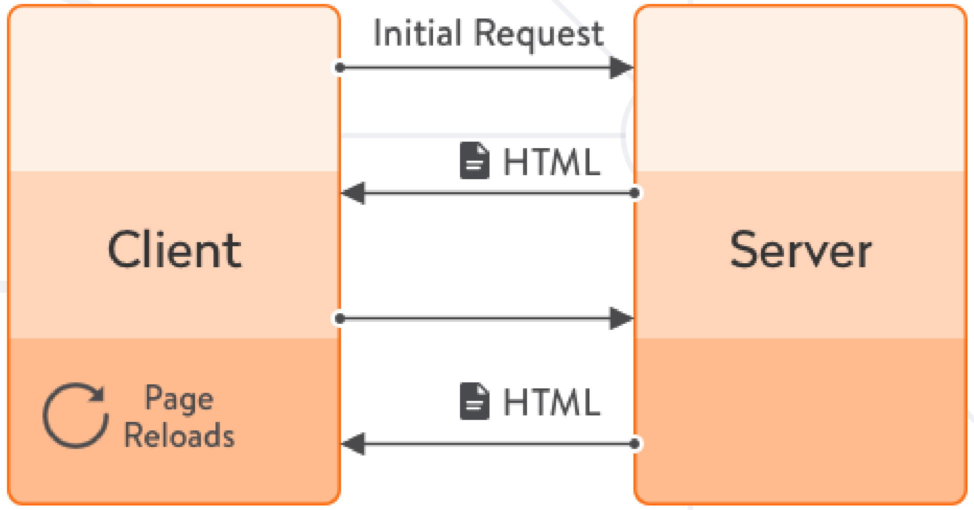
- **Web applications** are easy to install, use, update and are not bound to one device
  - In most cases, they are the preferable over desktop apps
- There are 2 participants in the web applications – **client** and **server**
- There are two main designs for web apps:
  - **Multi-Page application** (MPA) – the "traditional" approach
  - **Single-Page application** (SPA) – the "modern" approach



Multi-Page Applications (1)

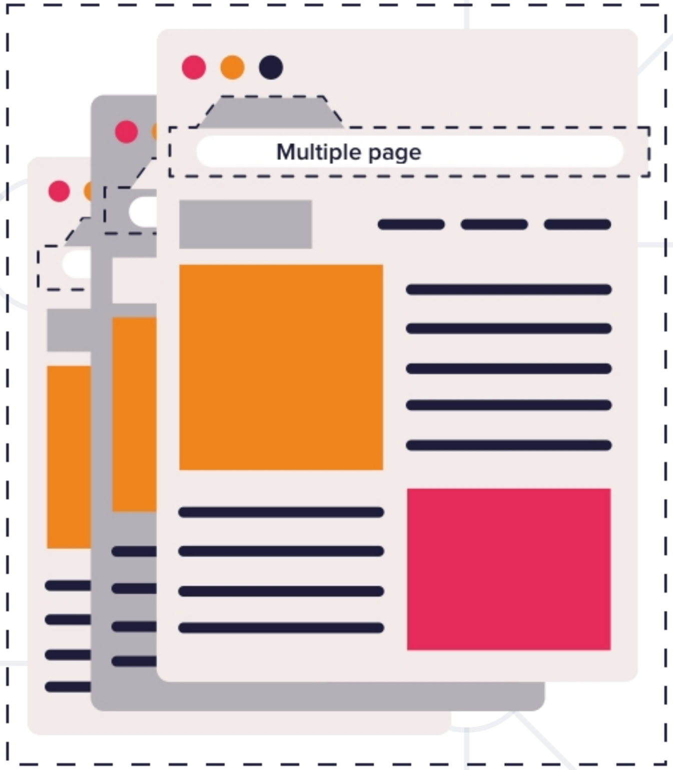
- **Multi-Page applications** work in a "**traditional**" way
  - Every change requests **rendering of a new page** in the browser
- Perform most of the application logic on the server
  - HTML is rendered on the server and returned as HTTP Response
    - AJAX and JavaScript may be used to add UI logic on the client
  - **ASP.NET Core MVC** and **Razor Pages** implement this approach

Multi-page app lifecycle



Multi-Page Applications (2)

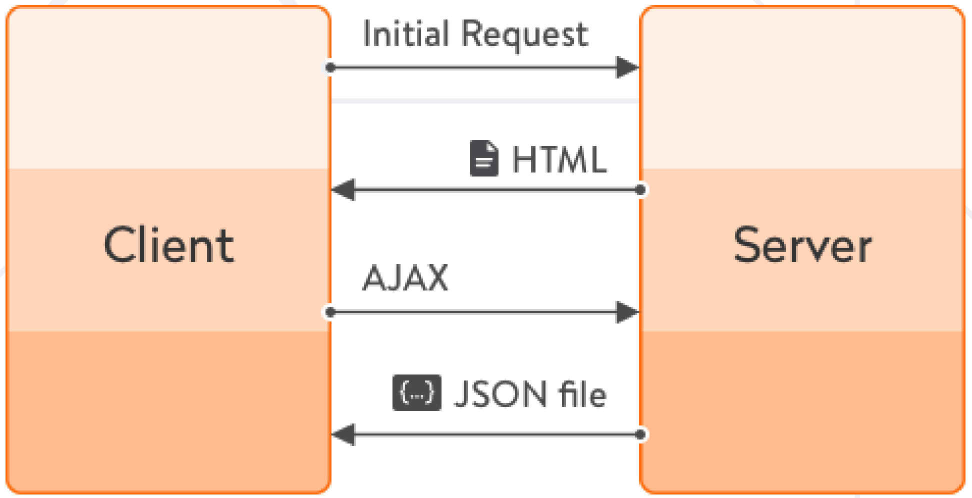
- PROs of Multi-Page applications
  - Useful for every type of projects
  - Very good and easy for proper **SEO management**
  - Using consistent languages, tools and technologies
- CONs of Multi-Page applications
  - Front-end and back-end are tightly coupled
  - The development and maintenance is quite complex
  - Requires page (state) **reload** on user action (link, form submit)



Single-Page Applications (1)

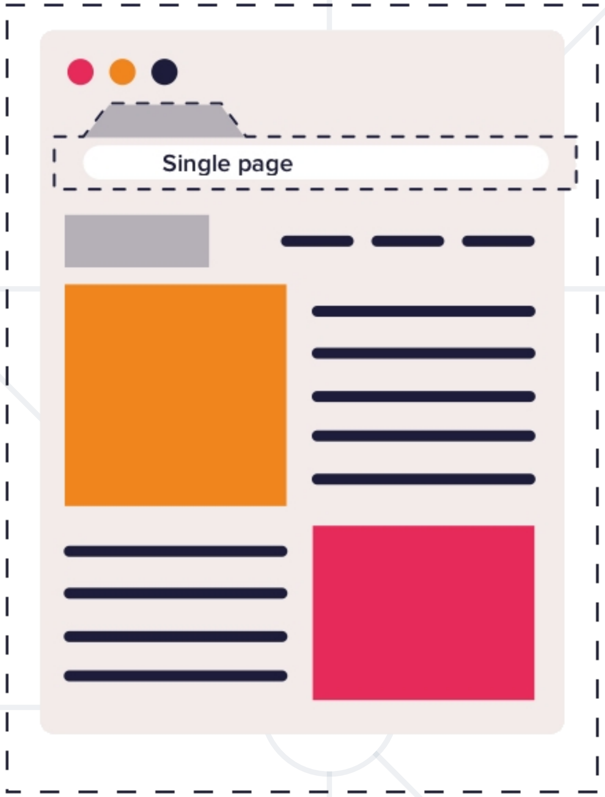
- **Single-Page applications** perform most of the **UI** in the browser
  - Does not require page reload during use
  - The **whole app is in one page** – content is changed dynamically
  - Examples: Gmail, Facebook, Instagram etc.
- **SPA** requests logic (JS, templates) and data independently
  - Back-end: ASP.NET Core Web API returning JSON data
  - Frond-end: Angular, React, Vue.js, Blazor, etc.

Single-page app lifecycle



Single-Page Applications (2)

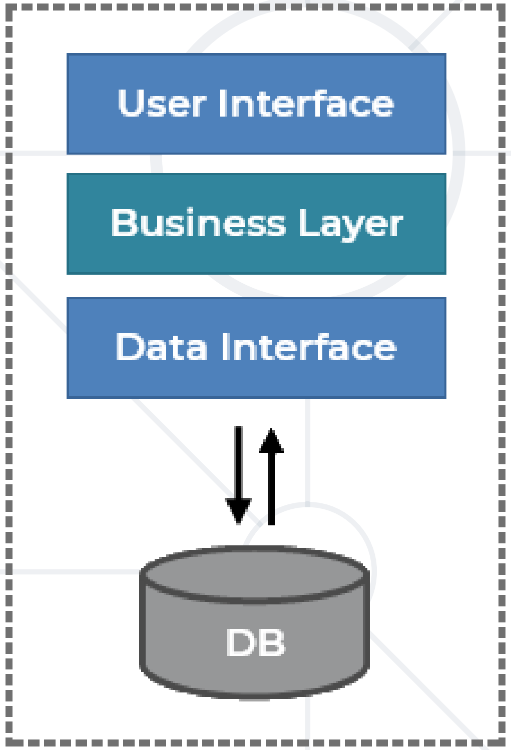
- **PRO**s of Single-Page applications
  - Animated, east-to-navigate and more user-friendly
  - SPAs are **fast**, most resources are loaded only once
  - Easy to make a corresponding **mobile application**
    - Reusing the same Back-End
- **CON**s of Single-Page applications
  - Quite tricky, and not easy to make SEO of the app
  - Slow to download, because of **heavy front-end frameworks**
  - Compared to "traditional" apps, SPAs are **less secure**
  - In most cases, require the use of **2 completely different technologies**



Web Application Architectures

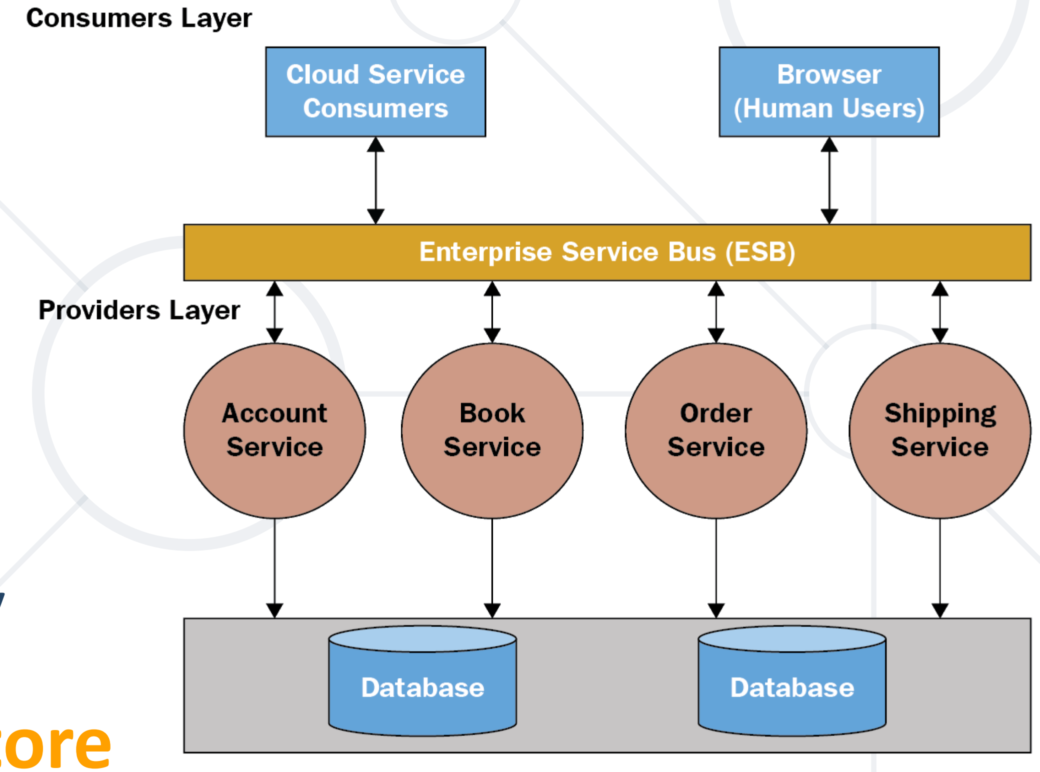
Monolithic Applications

- **Monolithic applications** are single-tiered applications
  - User interface and data access code are combined
  - The simplest form of architecture
- Deployment and maintenance is quite easy
  - Achieved due to lack of modularity and complexity
- **Monolithic apps** are recommended for small and mid-sized projects
  - Where the scope of functionality does not require abstractions
  - In most cases, monolith apps are not desired



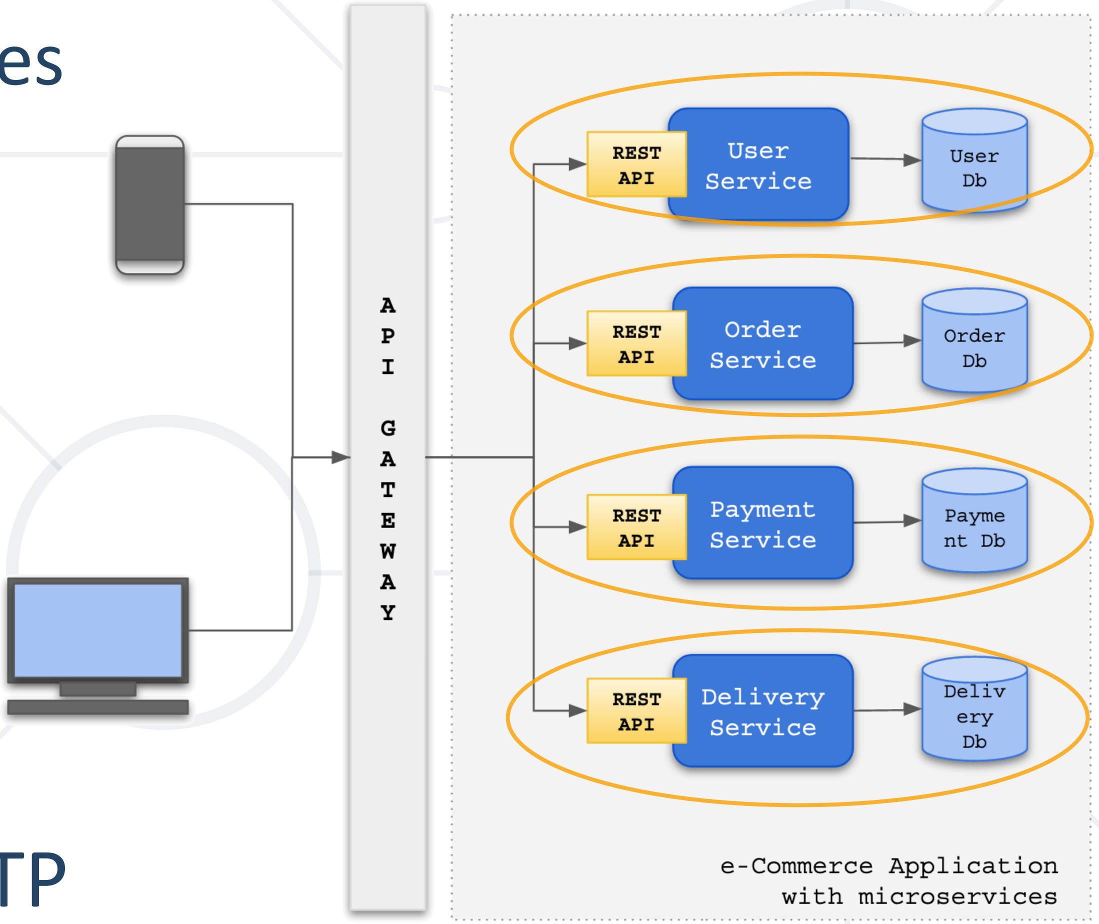
Service-Oriented Architectures (SOA)

- **Service-Oriented Architectures (SOA)**
  - Usually incorporate functions into smaller apps (**services**)
  - Communication is established over SOAP/XML, WS
    - Services communicate using **Enterprise Service Bus**
  - Services do **multiple activities** over a single scope of functionality
  - All services share **the same data store**



Microservices

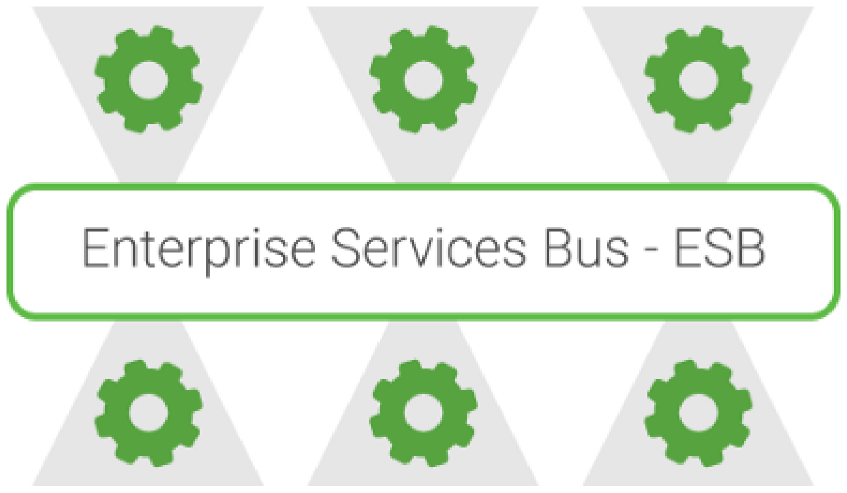
- **Microservices** is an architecture based on **lots of small applications**
  - Collection of loosely coupled services
  - The size should be minimal
- Enables **continuous deployment**
  - Can be deployed independently
- All services **communicate directly**
- Every **service has its own store**
- Communication: REST, Web API, HTTP



SOA vs Microservices

2000's

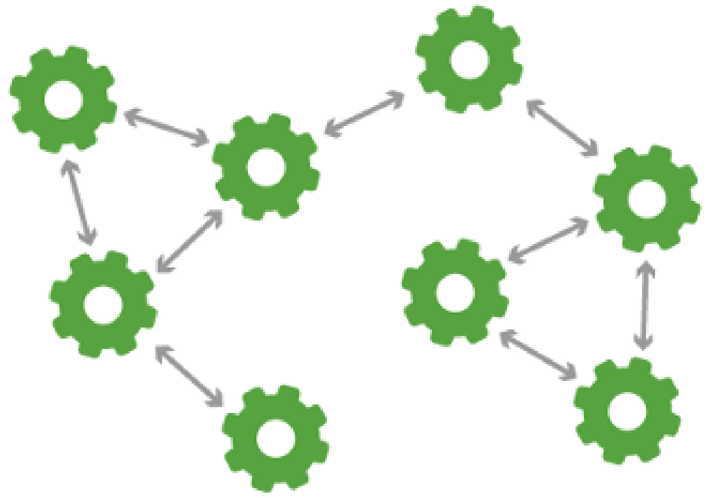
SERVICE ORIENTED ARCHITECTURE



SOA based applications are compromised of more loosely coupled components that use an Enterprise Services Bus messaging protocol to communicate between themselves.

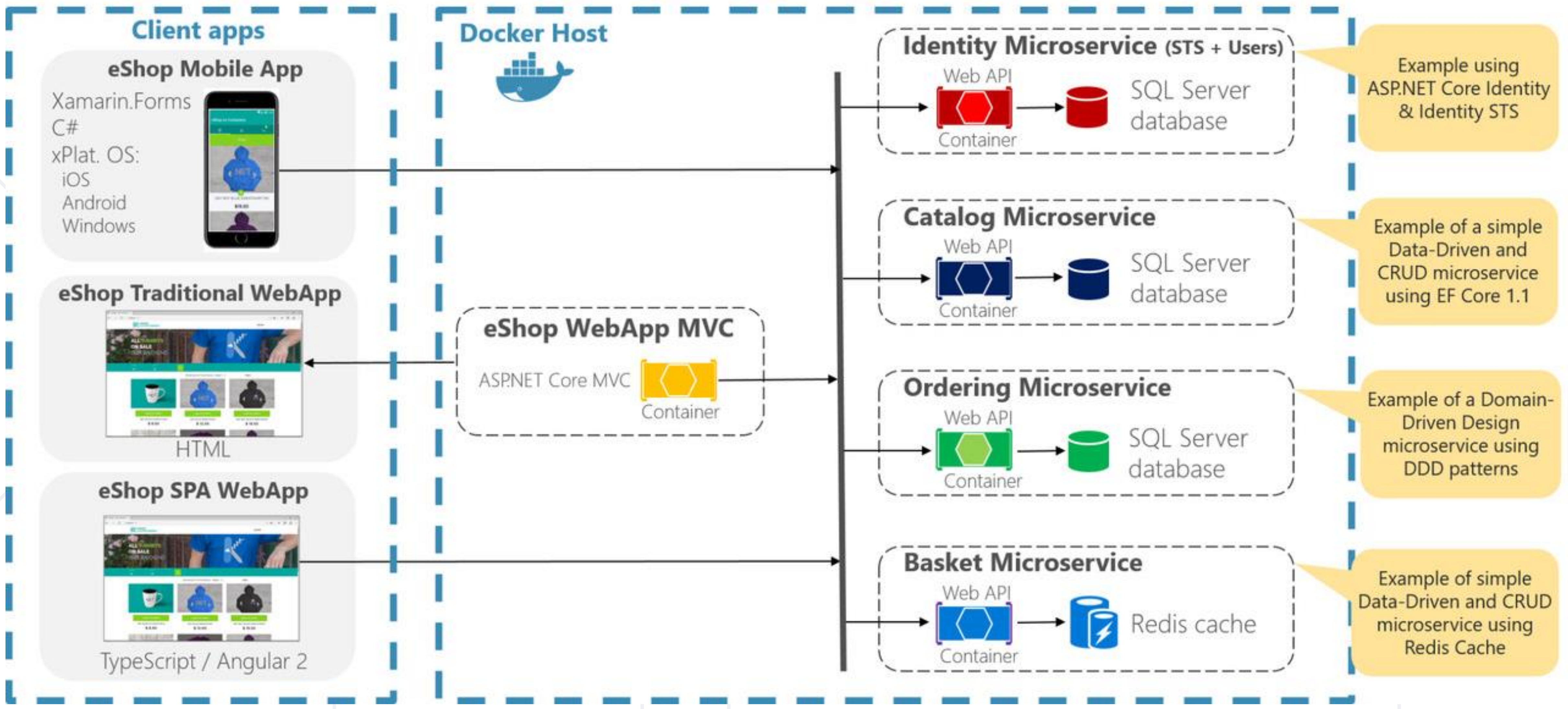
2010's

MICROSERVICES ARCHITECTURE



Microservices are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.

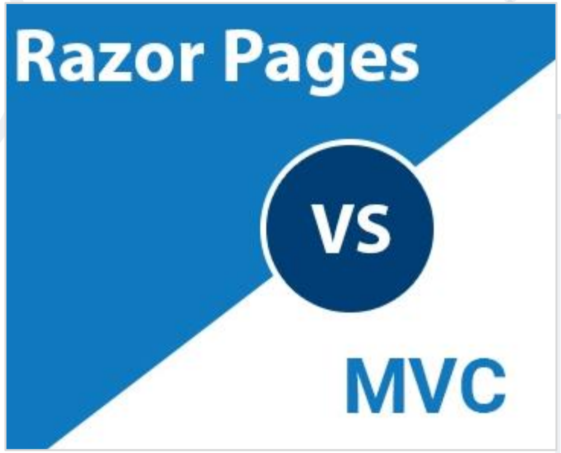
Example Microservices App



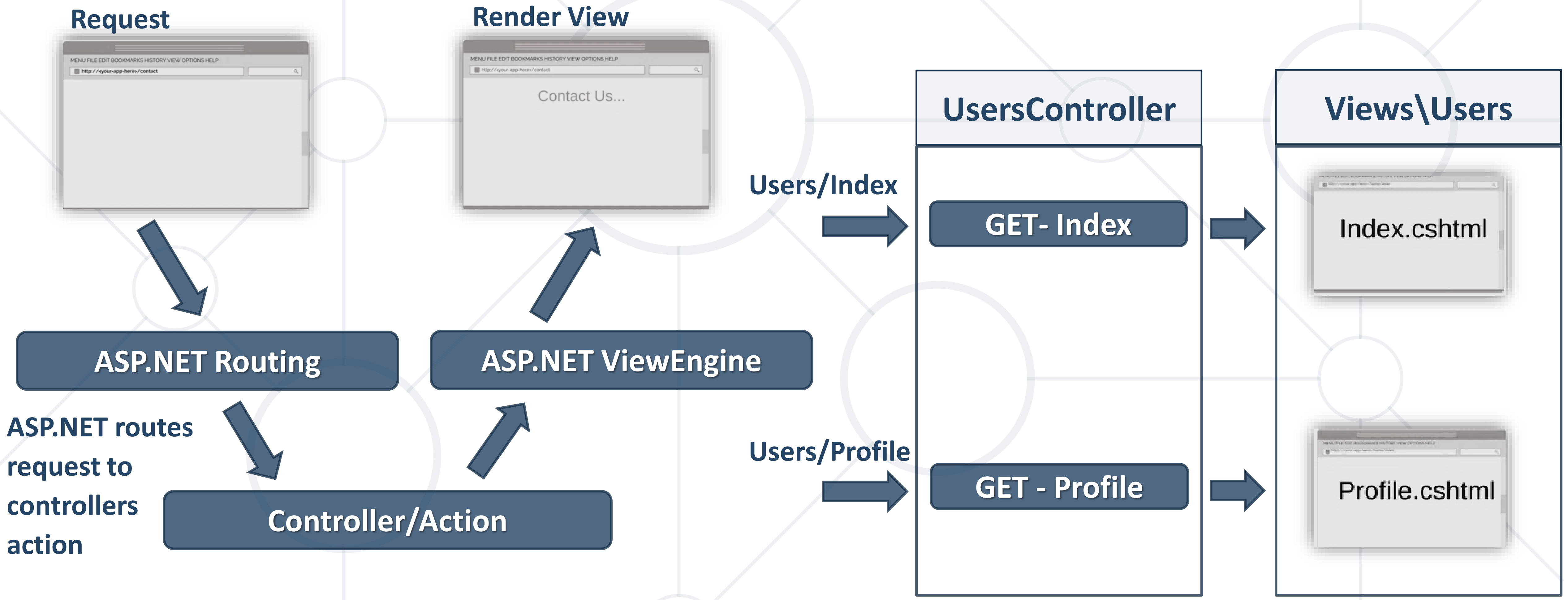
ASP.NET Core MVC vs Razor Pages

ASP.NET Core MVC vs Razor Pages

- Apart from **MVC**, **ASP.NET Core** provides another approach
  - Enter **Razor Pages**! A **Model-View-ViewModel**-like framework
- **Razor Pages** are similar to View Components
  - **Model** & **Controller** code is included in the **Page** itself
  - Enables two-way data binding and simpler development
  - Perfect for simple applications
    - With read-only functionality or simple data input
  - The single responsibility is strong

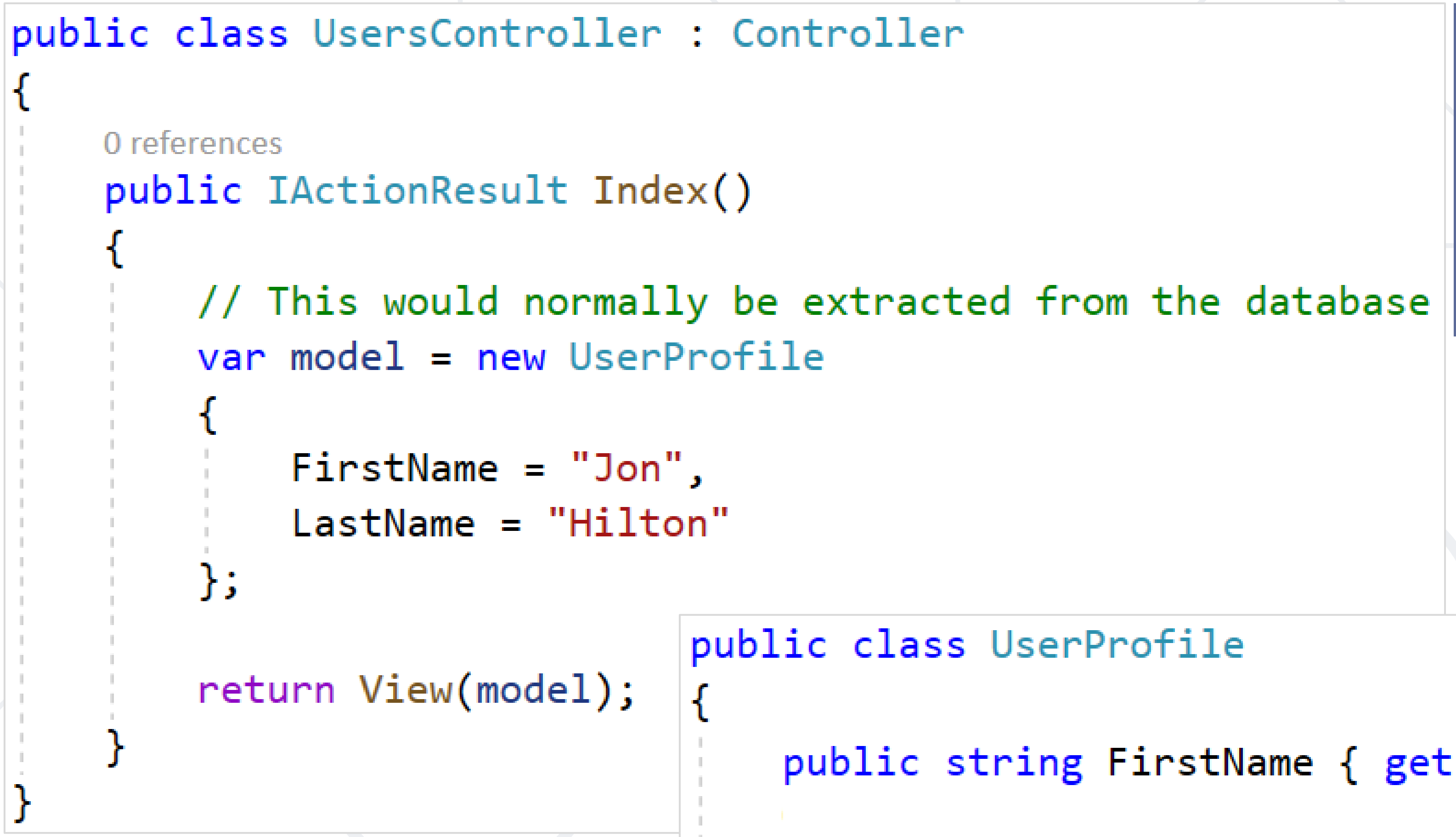


The MVC Approach (1)

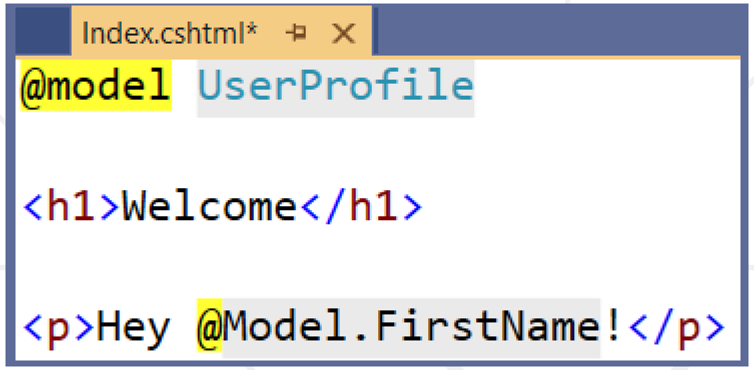


ASP.NET routes request to controllers action

The MVC Approach (2)

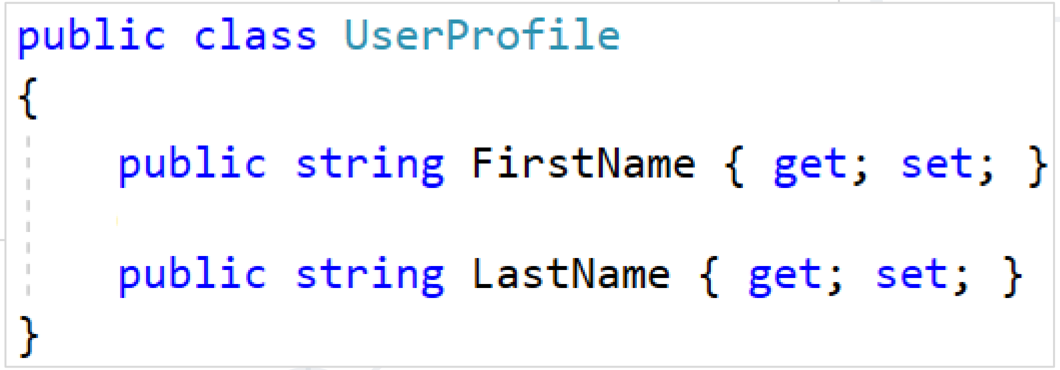


```
public class UsersController : Controller
{
    0 references
    public IActionResult Index()
    {
        // This would normally be extracted from the database
        var model = new UserProfile
        {
            FirstName = "Jon",
            LastName = "Hilton"
        };

        return View(model);
    }
}
```

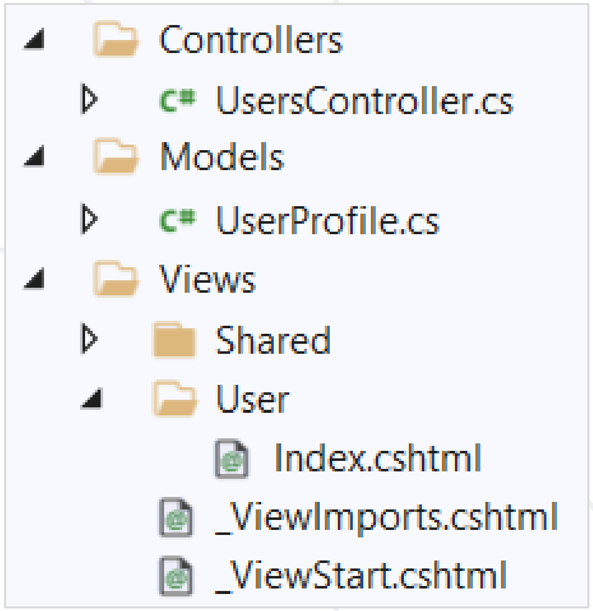


Index.cshtml*

```
@model UserProfile

<h1>Welcome</h1>

<p>Hey @Model.FirstName!</p>
```

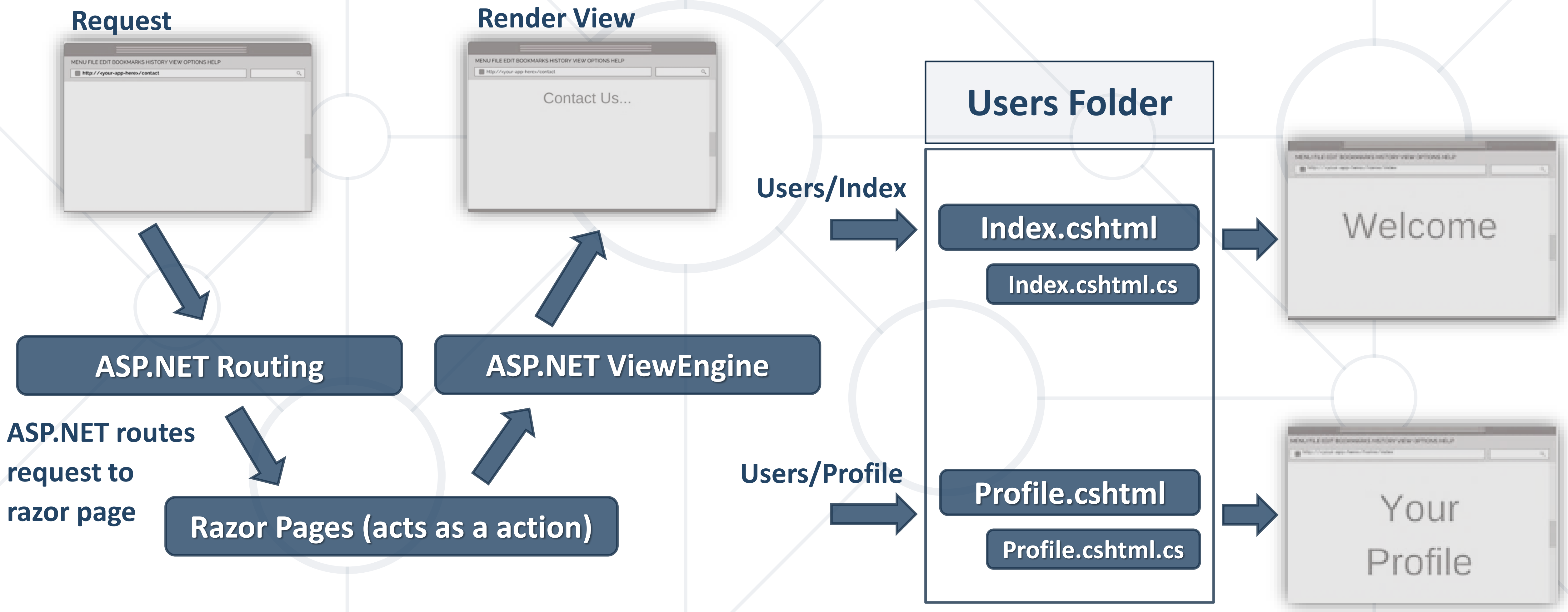


```
public class UserProfile
{
    public string FirstName { get; set; }

    public string LastName { get; set; }
}
```



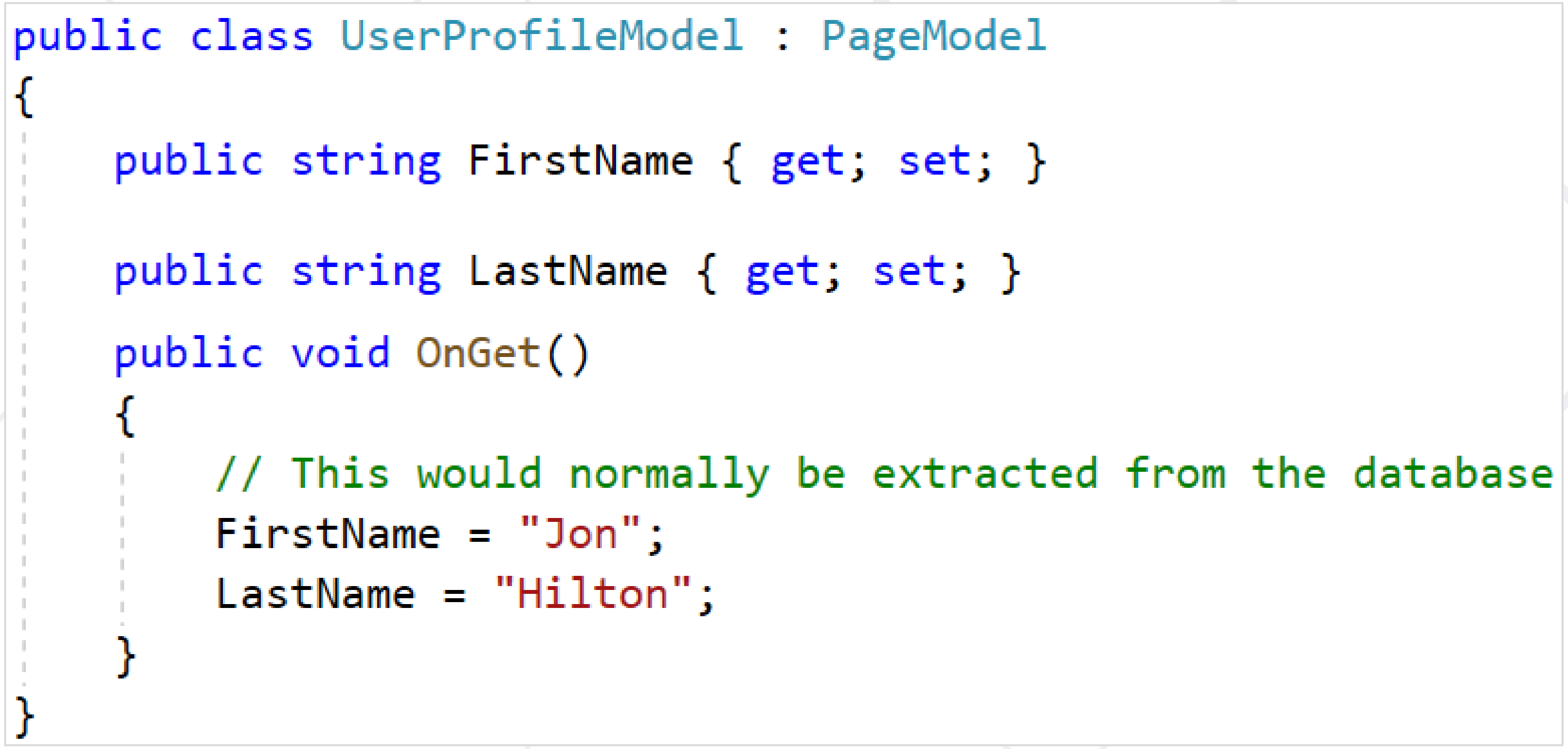
```
◢ Controllers
  ▷ UsersController.cs
◢ Models
  ▷ UserProfile.cs
◢ Views
  ▷ Shared
  ◢ User
      Index.cshtml
    _ViewImports.cshtml
    _ViewStart.cshtml
```

The Razor Pages Approach (1)

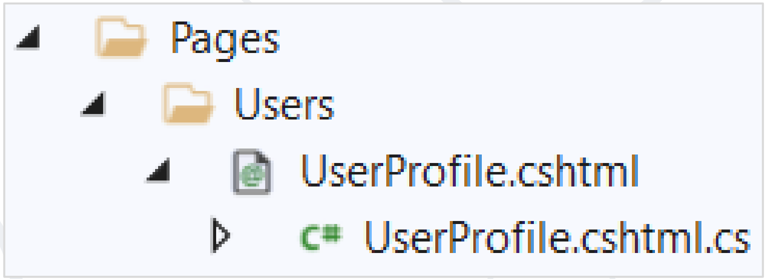


The Razor Pages Approach (2)

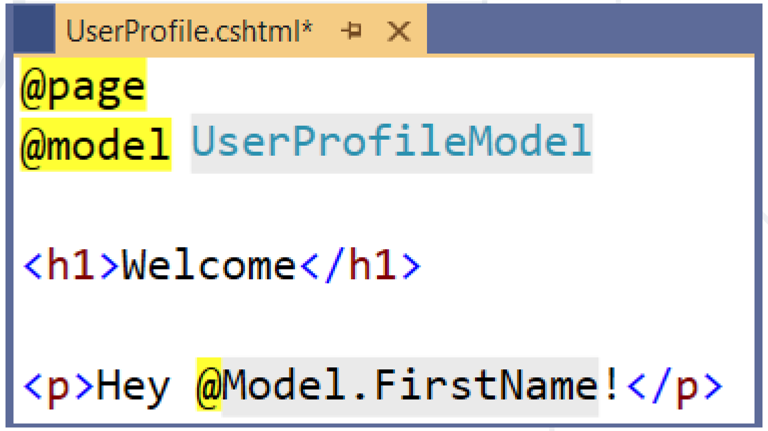
- Every **Razor Page** consists of
  - A view template (**.cshtml**), which acts as a view
  - A functional (**.cs**) file, which acts as its model + controller action



```csharp
public class UserProfileModel : PageModel
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public void OnGet()
    {
        // This would normally be extracted from the database
        FirstName = "Jon";
        LastName = "Hilton";
    }
}
```



```
Pages
  Users
    UserProfile.cshtml
      UserProfile.cshtml.cs
```

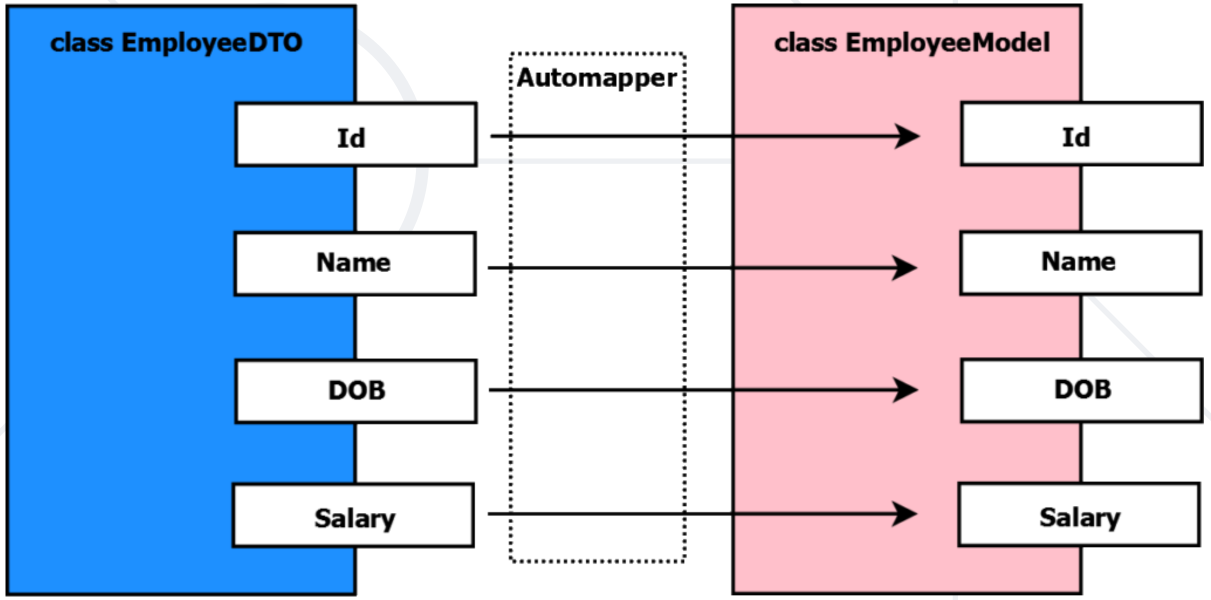


```cshtml
@page
@model UserProfileModel

<h1>Welcome</h1>

<p>Hey @Model.FirstName!</p>
```

AutoMapper

AutoMapper

- AutoMapper is a library built to simplify object mapping
  - **Easily imported** in ASP.NET Core
  - Added as a **dependency to the DI**
  - Gets rid of ugly property setters
  - Easy to use in code
  - Highly flexible
  - Easily configurable
  - Used in millions of projects



AutoMapper Setting

- Setting up the **AutoMapper** in your **ASP.NET Core** project

```
Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
```

  - This will also install the main **AutoMapper** NuGet package

- Registering **AutoMapper** as a dependency in the DI

```
builder.Services.AddAutoMapper(typeof(Program));
```

```
public class HomeController : Controller
{
    private readonly IMapper mapper;

    public HomeController(IMapper mapper)
    {
        this.mapper = mapper;
    }
    ...
}
```

AutoMapper Mapping

- Using the **AutoMapper** in your **ASP.NET Core** project

```
public class User
{
    public int Id { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Email { get; set; }
}
```

```
public class UserViewModel
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Email { get; set; }
}
```

The mapping class should inherit `Profile`

```
public class MappingProfile : Profile
{
    0 references
    public MappingProfile()
    {
        CreateMap<User, UserViewModel>();
    }
}
```

Create the **mapping** between User and UserViewModel

AutoMapper (Business Logic)

- Without **AutoMapper**

```
public class UsersController : Controller
{
    0 references
    public IActionResult Index()
    {
        // Populate the user details from DB
        var user = GetUserDetails();
        var userViewModel = new UserViewModel()
        {
            Email = user.Email,
            FirstName = user.FirstName,
            LastName = user.LastName
        };
        return View(userViewModel);
    }
}
```

Ugly, mistake-prone, unreadable

- With **AutoMapper**

```
public class UsersController : Controller
{
    private readonly IMapper mapper;
    public UserController(IMapper mapper)
        => this.mapper = mapper;
    public IActionResult Index()
    {
        // Populate the user details from DB
        var user = GetUserDetails();
        UserViewModel userViewModel =
            this.mapper.Map<UserViewModel>(user);
        return View(userViewModel);
    }
}
```

Clean, beautiful, simple

Easily modifiable

Commonly-syntaxed

Abstracting the Data Access Logic

Repository Pattern

Repository Pattern (1)

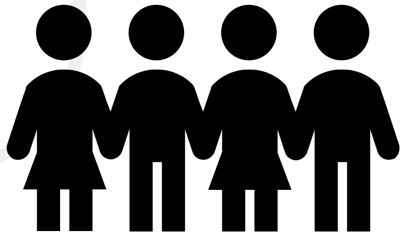
- **Repositories** are components that encapsulate data access logic
  - They **centralize** common data access functionality
  - They provide better **maintainability** and **testability**
  - They decouple the data access infrastructure from the **Domain layer**
- For each **aggregate**, you should define one **Repository**
  - Repositories, basically, allow you to populate data **in-memory**
  - Data is mapped from database to **Domain Entities**
  - Once in-memory, entities can be changed and **persisted back**

Repository Pattern (2)

- Normally you implement specific **Interface-Class** pairs
  - There are other ways, though. Like **Generic Repositories**, for example

```
public interface IRepository<TEntity>
{
    IQueryable<TEntity> All();
    void Add(TEntity entity);
    void Update(TEntity entity);
    void Delete(TEntity entity);
    Task<int> SaveChangesAsync();
}
```

```
public class EfRepository<TEntity> : IRepository<TEntity>
{
   private ApplicationContext context;
   private DbSet<TEntity> dbSet;

   public StudentRepository(ApplicationContext context)
   {
      this.context = context;
      this.dbSet = this.Context.Set<TEntity>();
   }

   public IQueryable<TEntity> All() => this.DbSet;
   public void Add(TEntity entity) => this.DbSet.Add(entity);
   public void Update(TEntity entity) { ... }
   public void Delete(TEntity entity) { ... }
   public Task<int> SaveChangesAsync() { ... }
}
```



Databases & ORMs

Object Relational Mapper (ORM)

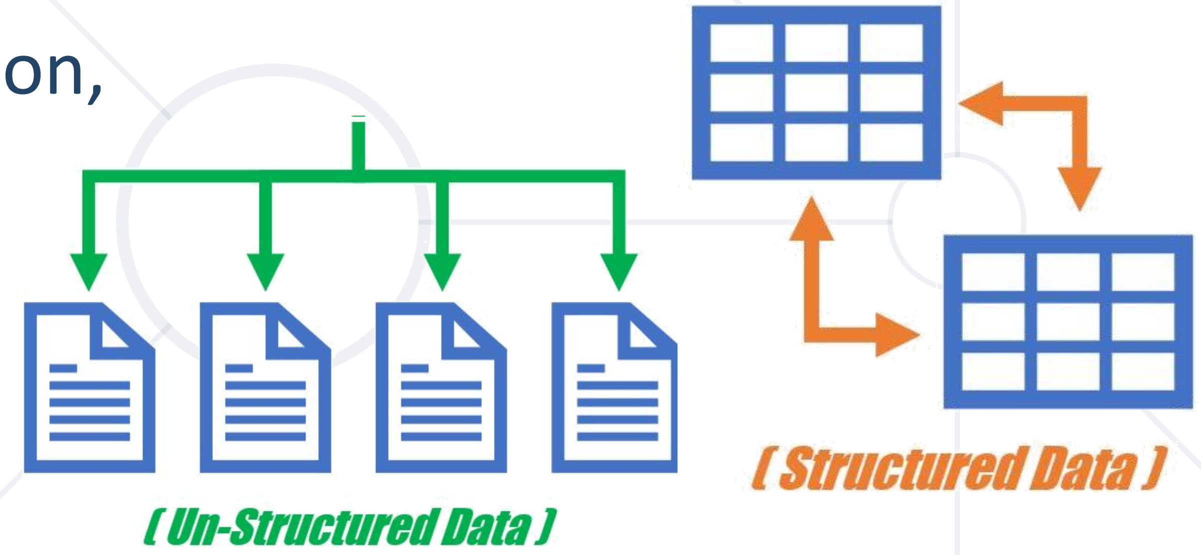
- **Entity Framework Core** is an **Object Relational Mapper** (ORM)
  - Creates a layer between your applications and data source
  - Maps the data to relational objects
- EF Core has a lot of essential and convenient features
  - Generates complex, optimized queries for your convenience
    - Translated from LINQ expression and cached
  - Manages the unit of work for you
  - Tracks changes in the Entities

Dapper

- But EF Core pays a cost for all of its features...
  - And that cost is performance
  - But there must be a faster alternative
- Enter **Dapper**! The Open-source Micro ORM
  - A lightweight micro ORM, and a very fast performing one
  - Dapper is "Closer to the metal"
  - Complex querying might be exceptionally hard
    - Not suited for lazy developers

Databases

- Developing an application requires the **choice of a database**
  - One of the most important decisions in the development
  - Two choices: **relational** (SQL) or **non-relational** (NoSQL) data structure
- **SQL** databases use **Structured Query Language** (SQL)
  - Data definition, Data manipulation, Querying, Programmability etc.
- **NoSQL** databases use dynamic schema for unstructured data
  - Data can be stored as Columns, Documents, Graphs, Key-Value pairs



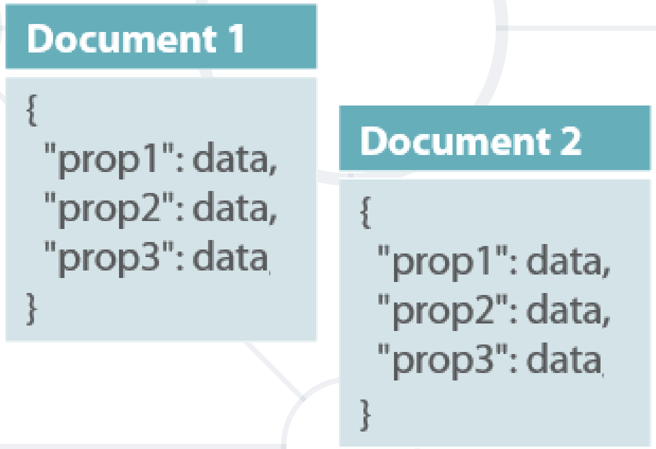
SQL

- SQL is extremely powerful, versatile, widely used
  - A safe choice, especially for complex querying
  - Very fast performing, even with large sets of data
- On the other hand, SQL can be **restrictive**
  - **Predefined schemas** are required to determine the data structure
  - All of the data must follow that predefined data structure
  - This requires significant up-front preparation and planning

Col1	Col2	Col3
Data	Data	Data
Data	Data	Data
Data	Data	Data

NoSQL

- **NoSQL databases** have their advantages and disadvantages too
  - You can create documents without pre-defining their structure
  - Each document can have its own unique structure
  - You can add fields on the go
- The drawbacks are also important to be noted
  - Lack of standardization
  - Lack of data consistency



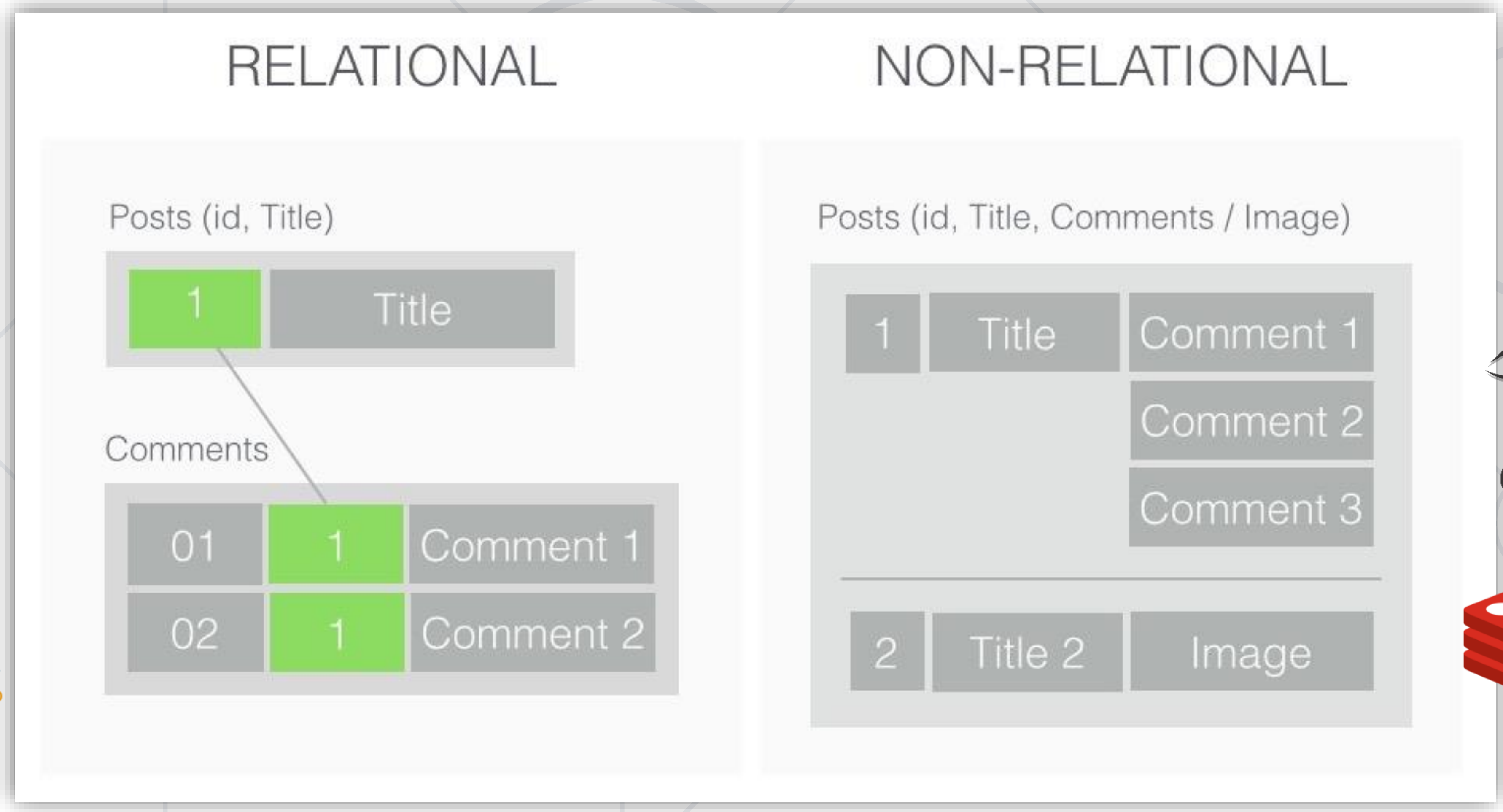
Document 1

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data
}
```

Document 2

```
{
  "prop1": data,
  "prop2": data,
  "prop3": data
}
```

SQL and NoSQL



Summary

- Web Application Designs
  - **MPA**s vs **SPA**s
- Web Application **Architectures**
  - Monolith vs SOA vs Microservices
- ASP.NET Core **MVC** vs **Razor** Pages
- **Repository** Pattern
- **AutoMapper**
- **Databases** & **ORM**s
  - ORM vs Micro-ORM
  - SQL vs NoSQL

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License