# 1. SPA vs MPA

## 1.1. Multi-Page Application (MPA) History:

- **Early Web Development (1990s):**
    - **Static Pages:** The early web was primarily composed of static HTML pages. Websites were collections of individual pages linked together.
    - **Server-Side Technologies:** As the web evolved, server-side technologies like CGI, PHP, ASP, and JSP were introduced. These technologies allowed for dynamic content generation on the server, but each interaction typically resulted in a full page reload.

**Rise of Server-Side Frameworks (2000s):**

- **Frameworks:** The 2000s saw the rise of server-side frameworks such as Ruby on Rails, Django, and ASP.NET. These frameworks facilitated the creation of MPAs by providing tools to handle routing, templating, and database interactions.
- **Enhanced User Interfaces:** Techniques like AJAX began to be used to enhance user interfaces by allowing parts of a page to be updated without a full reload, though these were still within the context of an MPA.

## 1.2. Single-Page Application (SPA) History:

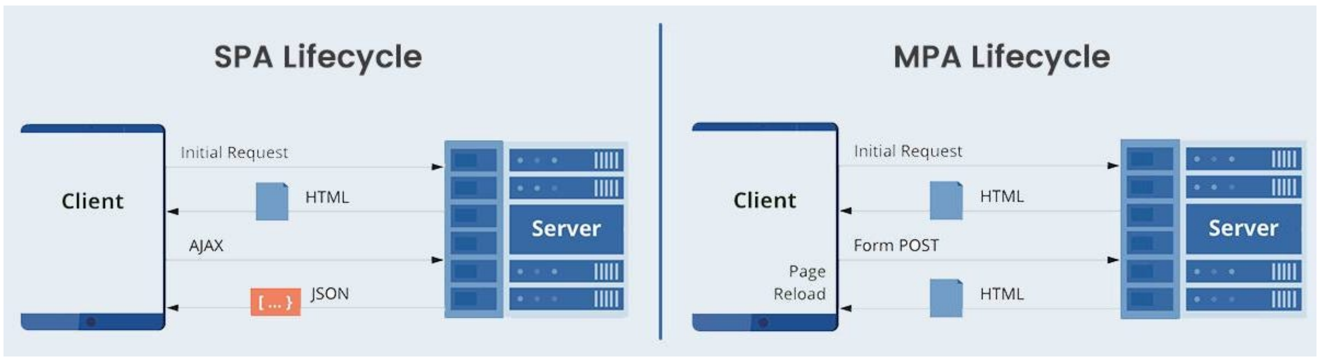
**Introduction of AJAX (Mid-2000s):**

- **Asynchronous JavaScript and XML (AJAX):** Introduced in the mid-2000s, AJAX was a pivotal technology that allowed web pages to fetch data asynchronously without reloading the entire page. This laid the groundwork for more dynamic web applications.
- **Early Examples of SPAs (Late 2000s):**
  - **Gmail and Google Maps:** Early examples of SPAs include Gmail and Google Maps. These applications demonstrated the potential for a seamless user experience without full page reloads.
  - **JavaScript Libraries:** JavaScript libraries like jQuery became popular, simplifying AJAX calls and DOM manipulation, further enabling the development of SPAs.

  **Modern JavaScript Frameworks (2010s):**
- **Frameworks and Libraries:** The early 2010s saw the emergence of frameworks and libraries specifically designed for building SPAs, such as AngularJS (2010), Backbone.js (2010), Ember.js (2011), and React (2013).
- **Client-Side Rendering:** These frameworks emphasized client-side rendering, where the browser executes JavaScript to dynamically update the UI, allowing for a more responsive user experience.
- **State Management:** As SPAs grew in complexity, state management libraries like Redux (2015) were introduced to handle the application state efficiently.

**Evolution and Optimization (2020s):**

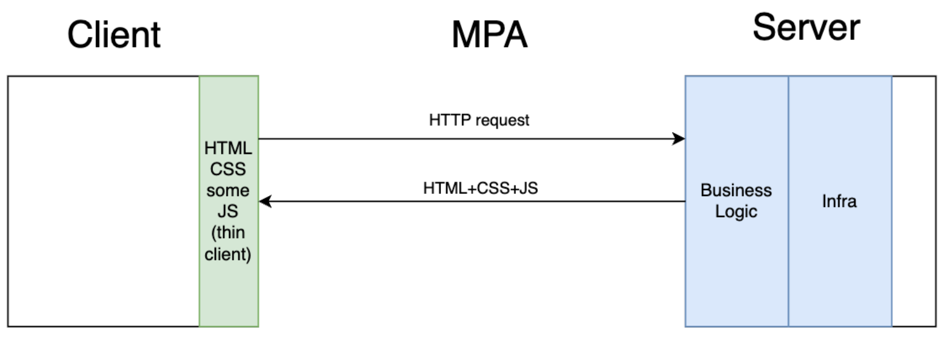
- **Server-Side Rendering (SSR) and Static Site Generation (SSG):** To address SEO challenges and performance issues, techniques like SSR and SSG became more prevalent. Frameworks like Next.js (React) and Nuxt.js (Vue.js) made it easier to implement these techniques.
- **Micro Frontends:** The concept of micro frontends emerged, allowing large applications to be split into smaller, independently deployable units, combining benefits of SPAs and MPAs.
- **MPAs** have a long history rooted in the early days of the web, evolving through server-side technologies and frameworks that emphasized full page reloads.
- **SPAs** emerged with the advent of AJAX and have been propelled by modern JavaScript frameworks, focusing on a seamless and dynamic user experience.

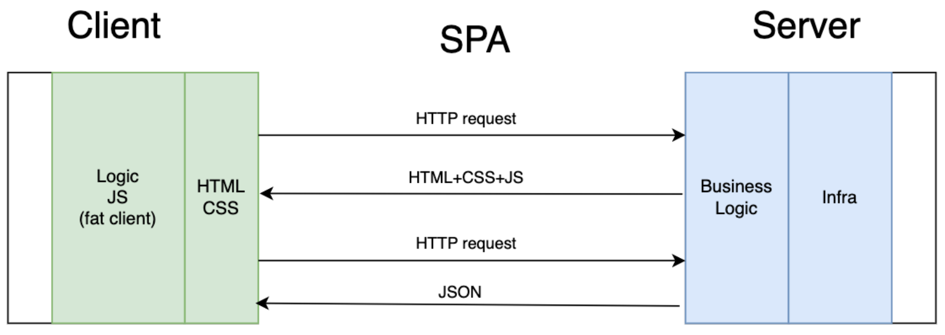


## 1.3. Thin client vs fat client

The first diagram below represents an MPA (Multi-Page Application). On the left, we have a browser, which is considered a "thin client." We use this term because the client code consists mostly of HTML, CSS, and some JavaScript. The rest of the logic resides on the server side.

In the second diagram, we see an SPA (Single-Page Application). In this case, much more logic is located on the frontend, written in JavaScript. We can use frameworks like React or Angular to help manage the DOM updates. We can also see that, in an SPA, the response to the first request returns HTML, CSS, and JavaScript. For subsequent requests, the server returns only data in JSON format. The frontend application then uses JavaScript to update the UI accordingly.





## 1.4. When to Use SPA vs. MPA

### 1.4.1. Use SPA When:

1. **Rich Interactivity:** The application requires a high level of interactivity and a seamless user experience (e.g., dashboards, social media platforms, email clients).
2. **Single Functionality:** The application focuses on a single, complex functionality (e.g., real-time collaboration tools).
3. **Mobile Experience:** The application needs to provide a mobile app-like experience with fast and dynamic interactions.
4. **Desktop-Like Application:** When the web app should behave more like a desktop application with minimal load times after the initial load.

### 1.4.2. Use MPA When:

1. **SEO Importance:** The website relies heavily on SEO and requires easy indexing by search engines (e.g., blogs, news sites, e-commerce sites).
2. **Content-Rich:** The website contains a large amount of content that is best served through multiple distinct pages.
3. **Simplicity:** The application is simpler, and traditional page reloads do not detract from the user experience (e.g., corporate sites, informational sites).
4. **Low Interactivity:** When the application does not require significant client-side interactivity and can function well with standard page reloads.

### 1.4.3. Summary:

- **SPAs** are ideal for applications requiring high interactivity and a seamless user experience, but come with complexities in development and SEO.
- **MPAs** are better suited for content-rich sites where SEO is crucial and page reloads are acceptable, offering simplicity in development.

Remember that nowadays we can often mix both approaches together.

## 2. ASP.NET Core

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.

With ASP.NET Core, you can:

- Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET.

## 3. Features of ASP.NET Core platform

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
    - Kestrel
    - IIS
    - HTTP.sys
    - Nginx
    - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

## 3. Web applications

ASP.NET Core MVC provides features to build web APIs and web apps:

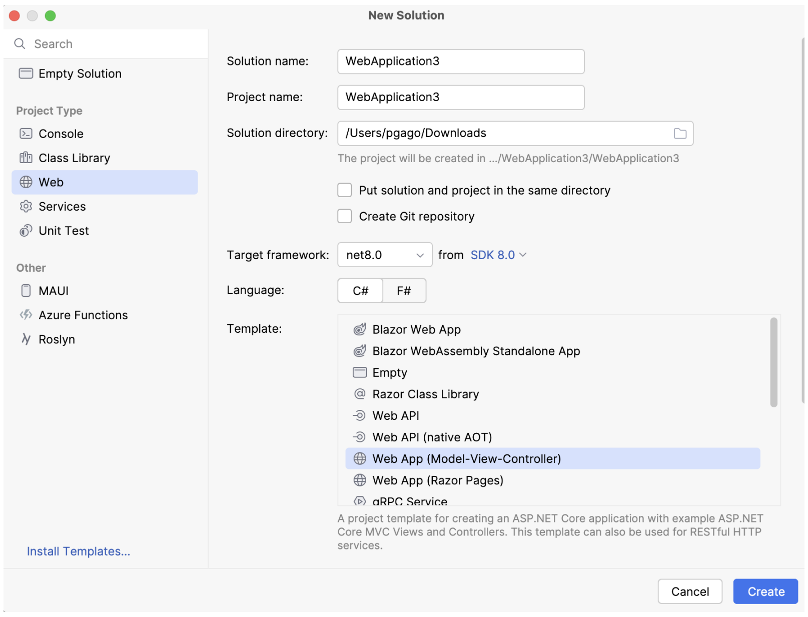
- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

## 4. Client-side development

ASP.NET Core includes Blazor for building richly interactive web UI, and also integrates with other popular frontend JavaScript frameworks like Angular, React, Vue, and Bootstrap. For more information, see ASP.NET Core Blazor and related topics under *Client-side development*.

## 5. MVC approach vs Razor pages

During our classes we will show MVC (ang. Model-View-Controller) variant for ASP.NET project.



New Solution

Solution name: WebApplication3

Project name: WebApplication3

Solution directory: /Users/pgago/Downloads

The project will be created in .../WebApplication3/WebApplication3

☐ Put solution and project in the same directory

☐ Create Git repository

Target framework: net8.0 from SDK 8.0

Language: C# F#

Template: Web App (Model-View-Controller)

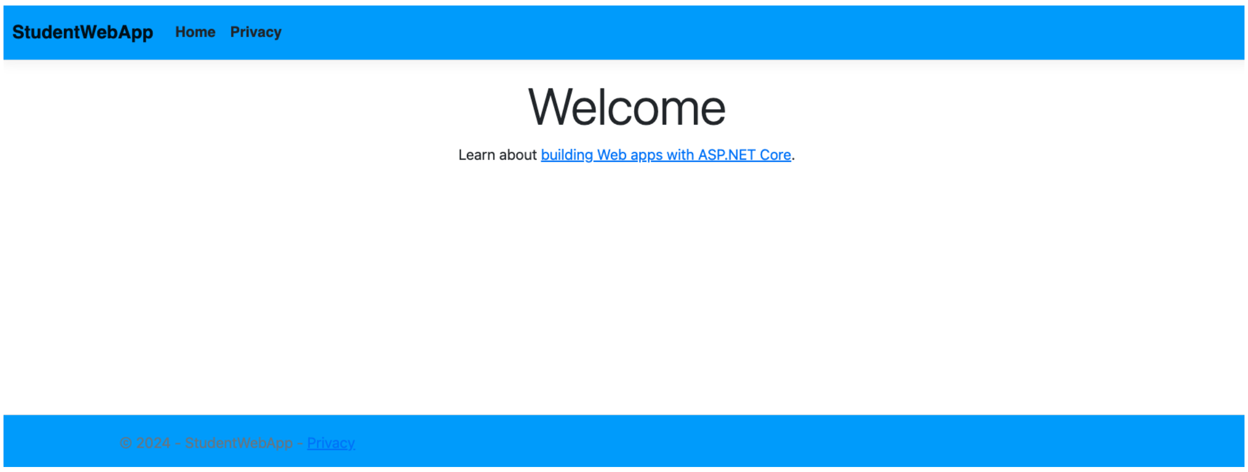
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

The second variant of WebApp is called "Razor Pages." This is a newer variant that we will not discuss during our classes. The MVC variant is much more popular, and you can encounter a similar approach in many other web application frameworks.

## 6. New project

After creating a new webpage, we already have a generated template. The created template includes:

- The Bootstrap CSS framework added
- jQuery added
- A basic layout generated for us
- A single controller, Home, generated with two pages: Index and Privacy



Welcome

Learn about building Web apps with ASP.NET Core.

© 2024 - StudentWebApp - Privacy

# 7. Differences between ASP.NET API (REST) vs ASP.NET Web App (MPA)

There are a lot of similarities between the REST api application and ASP.NET Web App. Both of them are using a lot of the same mechanics.

## 7.1. Different routing convention

By default, in REST APIs, we use attribute-based routing. This means that we use attributes like `[Route("api/students")]` or `[HttpPost("{id}")]` to define our routes. In ASP.NET Web Apps, we use convention-based routing.

However, we can still use attribute-based routing to override the convention.

Convention-based routing is a general pattern of URLs defined at the global level. We can change this pattern or even add multiple other patterns. This can be found in the `Program.cs` file. The first segment of the URL is the name of the controller, the second is the name of the method, and the third is the ID.

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

## 7.2. Different base class for the controllers

The controller classes in our ASP.NET WebApp have different base classes.

```
using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using StudentWebApp.Models;

namespace StudentWebApp.Controllers;

public class HomeController : Controller
{
```

This time, this is a Controller class. This parent class allows us to use different methods, such as the `View()` method, which can be used to identify the view we should return for a specific page.

## 7.3. Static files

In our project we will notice a project called wwwroot which is used to store static files - like JS, CSS and JS libraries. Those files later can later be used within our views.

## 7.4. Views

Within our project, we will notice the `Views` folder. In this directory, we can find files using the Razor syntax by default. These files are later returned to the client by the controllers.

## 8. Razor view engine

A view engine is a general term used for frameworks/libraries that allow us to more easily generate dynamic HTML. Web applications constantly need to generate new HTML depending on the client's interaction with the webpage. Razor syntax allows us to more easily combine dynamic C# elements with static HTML elements.

Example:

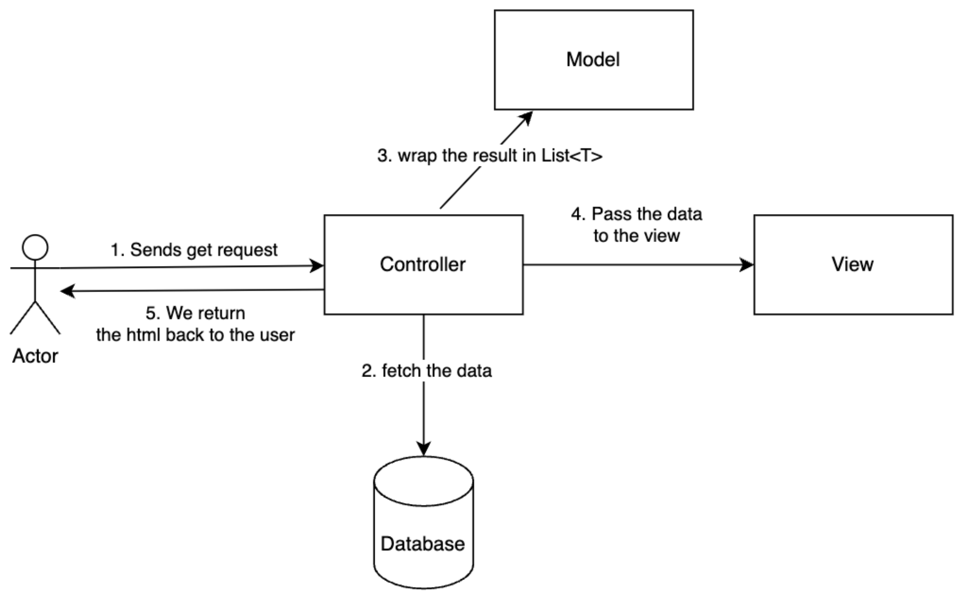
```
@{
    ViewBag.Title = "Home Page";
    var currentTime = DateTime.Now;
}

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
</head>
<body>
    <h1>Welcome to our website!</h1>
    <p>The current time is: @currentTime</p>
</body>
</html>
```

In this example, the Razor syntax is used to set the title of the page and display the current time using C#. The `@{}` block allows you to write C# code, and the `@` symbol is used to insert dynamic content into the HTML.

# 9. Passing the data to the view

In this section, we will take a look at how we can pass data from the controller to the view.

This is a fundamental aspect of MVC architecture, where the controller handles user input and interactions, processes data using the model, and then passes that data to the view for presentation to the user.


Model
3. wrap the result in List<T>
4. Pass the data to the view
1. Sends get request
Controller
View
5. We return the html back to the user
Actor
2. fetch the data
Database

There are several ways to pass data from a controller to a view in ASP.NET, including using ViewBag and strongly-typed models. Each method has its own use cases and benefits.

## 9.1. ViewBag

ViewBag is a dynamic object that provides a convenient way to pass data to the view. It is essentially a wrapper around ViewData.

Example (controller):

```
public ActionResult Index()
{
    ViewBag.Message = "Hello, World!";
    return View();
}
```

Example (view):

```
<h1>@ViewBag.Message</h1>
```

- **Benefits:** Simple, flexible, no need for model classes.

- **Disadvantages:** No compile-time checking, harder to maintain, limited Intellisense support.

## 9.2. Strongly-typed views

Strongly-typed models provide a robust and type-safe way to pass data to the view by using model classes.

Example (controller):

```
public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public ActionResult Index()
{
    var model = new Student { FirstName = "Jan", LastName="Kowalski" };
    return View(model);
}
```

Example (view):
**Remember that we have to define the model that this view expects at the top of the view.**

```
@model Student

<ul>
        <li>Model.FirstName</li>
        <li>Model.LastName</li>
</ul>
```

- **Benefits:** Compile-time checking, Intellisense support, clear and maintainable code, type safety.
- **Disadvantages:** Increased overhead, less flexibility, initial learning curve.

## 9.3. Summary

The choice between ViewBag and strongly-typed models depends on the specific needs of your application. ViewBag is useful for quick, small-scale data passing, while strongly-typed models are better for larger, more complex applications where reliability, maintainability, and type safety are crucial.

# 10. Form - interaction with the user

HTML forms are a typical way to allow the user to send data to the server. They provide a user-friendly interface for inputting and submitting data, which can include text fields, checkboxes, radio buttons, drop-down lists, and file uploads.

When a user submits a form, the browser sends the data to the server using an HTTP request, typically a POST or GET request.

The server then processes this data, which can involve storing it in a database, performing calculations, or triggering other server-side logic, and returns a response to the user.

Forms are fundamental to web applications, enabling a wide range of interactive functionalities such as user registration, login, search queries, and data entry.

## 10.1. Example of the form - search

Example (controller):

```
public IActionResult Index(string query)
{
    if (!string.IsNullOrWhiteSpace(query))
    {        return View(students.Where(s =>
s.LastName.Contains(query)).ToList());
    }    return View(students);
}
```

Example (view):

```
@model List<Student>
<h1>Lista studentów</h1>
<a class="nav-link text-dark" asp-area="" asp-controller="Students" asp-
action="Create">Dodaj</a>
<form method="get">
    <input type="text" name="query" id="query" />
    <button type="submit">Search</button>
</form>
....
```

In the example above we have created a form which allows the user to send

- **<form method="get">:**
  - This is an HTML form element that submits data using the GET method. GET requests append the form data to the URL, making it visible in the address bar.
- **<input type="text" name="query" id="query" />:**
  - This is an input element of type `text` with the `name` and `id` attributes both set to "query."
  - Users can type a search query into this text box.
- **<button type="submit">Search:**
  - This is a button element with the type `submit`. When clicked, it submits the form.
  - The button text "Search" indicates that it will initiate a search based on the user's input in the text box.

**`name` Attribute:**

1. **Purpose:**
   - The `name` attribute specifies the name of an input element and is crucial for form data submission.
   - When a form is submitted, the `name` attribute of each input element is used as the key in the key-value pair sent to the server. For example, in the input element `<input type="text" name="query" />`, the value entered by the user will be associated with the key "query".
2. **Usage:**
   - The `name` attribute is essential for server-side processing. The server-side script or controller action that handles the form submission will use the `name` attribute to retrieve the corresponding value.
   - Example: If a form with `name="query"` is submitted with the value "student", the server will receive this as part of the query string or request body, like `query=student`.

**`id` Attribute:**

1. **Purpose:**
   - The `id` attribute provides a unique identifier for an HTML element.
   - It is used for client-side scripting and styling, allowing JavaScript and CSS to interact with the element directly.
2. **Usage:**
   - The `id` attribute must be unique within a document. It can be used to target the element with CSS for styling or with JavaScript for dynamic behavior.
   - Example: Using CSS, you can style an element with `id="query"` as `#query { color: blue; }`. Using JavaScript, you can reference it as `document.getElementById("query")`.

Setting `name` and `id` to the same value is a common practice for convenience and consistency, ensuring both server-side and client-side code refer to the same element easily.

## 10.2. Example of the form for creating new student

In the example below we have created a form for adding new student. It is useful to notice a few things.

First we created a model:

```
public class Student
{
    [Key]
    public int IdStudent { get; set; }
        [Required]
    [MaxLength(100)]
    [DisplayName("First name")]
    public string FirstName { get; set; }
    [Required]
    [MaxLength(100)]
    [DisplayName("Last name")]
    public string LastName { get; set; }
    [Required]
    [MaxLength(100)]
    public string Address { get; set; }
    [Required]
    [EmailAddress]
    [MaxLength(100)]
    public string Email { get; set; }
}
```

ASP.NET uses data annotations to define validation rules on model properties. These annotations are applied in the model class and specify the criteria that the input data must meet.

Secondly we create the proper methods in controller:

```
public IActionResult Create()
{
    return View();
}

[HttpPost]
public IActionResult Create(Student newStudent)
{
    if (ModelState.IsValid)
    {         students.Add(newStudent);
        return RedirectToAction("Index");
    }    return View(newStudent);
}
```

The we create the view which will display form for the student.

```
@model Student
<h1>Dodaj studenta</h1>
<form method="post">
    <div class="form-group">
        <label asp-for="FirstName"></label>
        <input class="form-control" asp-for="FirstName"/>
        <span asp-validation-for="FirstName"></span>
    </div>    <div class="form-group">
        <label asp-for="LastName"></label>
        <input class=    "form-control" asp-for="LastName"/>
        <span asp-validation-for="LastName"></span>
    </div>    <div class="form-group">
        <label asp-for="Address"></label>
        <input class="form-control" asp-for="Address"/>
        <span asp-validation-for="Address"></span>
    </div>    <div class="form-group">
        <label asp-for="Email"></label>
        <input class="form-control" asp-for="Email"/>
        <span asp-validation-for="Email"></span>
    </div>    <input type="submit" value="Dodaj"/>
</form>
```

### 10.2.1. Directives in ASP.NET

When we generate the HTML often we have to make sure that we use proper names. In the example of the form we have to make sure to set the name and if properties to corresponding name of properties in our model.
To make it simple we can use asp directives. **Those directives gives us compile-type checking**.
Example:

```
<input class="form-control" asp-for="FirstName"/>
```

asp-for will automatically extract information from the Student model and add proper HTML attributes.

- **Automatic Binding:**
  - The `asp-for` tag helper simplifies binding form inputs to model properties, reducing the chance of errors that can occur with manual binding.
- **Consistency:**
  - Ensures consistency in `name` and `id` attributes, which is crucial for model binding and for client-side interactions like JavaScript or CSS targeting.
- **Ease of Maintenance:**
  - If the property name changes in the model, it needs to be updated only in the `asp-for` attribute, making maintenance easier and reducing the risk of discrepancies.

### 10.2.2. Server-side validation

The provided code demonstrates how to handle form submission with server-side validation in ASP.NET Core.
`ModelState.IsValid` is used to check if the model binding and validation succeeded.

Example (controller):

```
[HttpPost]
public IActionResult Create(Student newStudent)
{
    if (ModelState.IsValid)
    {
        students.Add(newStudent);
        return RedirectToAction("Index");
    }
    return View(newStudent);
}
```

- **Model Binding:**
  - When the form is submitted, the data entered by the user is bound to the `Student` model object ( `newStudent` ) by the ASP.NET Core model binding system.
- **ModelState:**
  - `ModelState` is a property of the `Controller` base class. It represents the state of model binding and validation.
  - It contains information about errors that occur during model binding and validation.
- **Validation Check:**
  - `ModelState.IsValid` checks whether the model binding and validation process was successful. It returns `true` if there are no validation errors; otherwise, it returns `false`.
- **Handling Valid Data:**
  - If `ModelState.IsValid` is `true`, indicating that there are no validation errors:
    - The new `Student` object ( `newStudent` ) is added to the `students` collection.
    - The user is redirected to the `Index` action, typically displaying a list of students, with the `RedirectToAction` method.
- **Handling Invalid Data:**
  - If `ModelState.IsValid` is `false`, meaning there are validation errors:
    - The view is returned with the `newStudent` object, which contains the user-entered data and validation errors.
    - This allows the view to display the validation messages to the user, prompting them to correct the errors.

To display the validation on the server side we use:

```
<span asp-validation-for="FirstName" class="text-danger"></span>
```

The line above will display the result of server side and client side validation (if it is turned on).

## 10.2.3. Client-side validation

ASP.NET Core uses unobtrusive JavaScript for client-side validation, which means that validation logic is automatically applied without cluttering the HTML markup with validation code.
This is enabled by including jQuery and jQuery Validation scripts in your project.

In the `_Layout.cshtml` file or the specific view, you need to ensure these scripts are referenced:

```
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"
```

Then to display the validation result in the view we should remember to add:

```
<span asp-validation-for="FirstName" class="text-danger"></span>
```

The line above will display validation for FirstName field.

**Example of Client-Side Validation Workflow:****

1. **User Input:**
   - The user fills out the form fields. As they enter data, the validation rules defined by the data annotations are applied.
2. **Immediate Feedback:**
   - If the user violates any validation rule (e.g., leaving a required field empty), an error message is shown immediately next to the input field.
3. **Form Submission:**
   - When the user clicks the submit button, the form is checked for validation errors by jQuery Validation.
   - If any validation rules are violated, the form is not submitted, and the user is prompted to correct the errors.
4. **Valid Data:**
   - If all inputs pass the validation checks, the form is submitted to the server for further processing.

## 10.2.4. Server-side vs client-side validation

Server-side validation and client-side validation serve complementary roles in web application development.

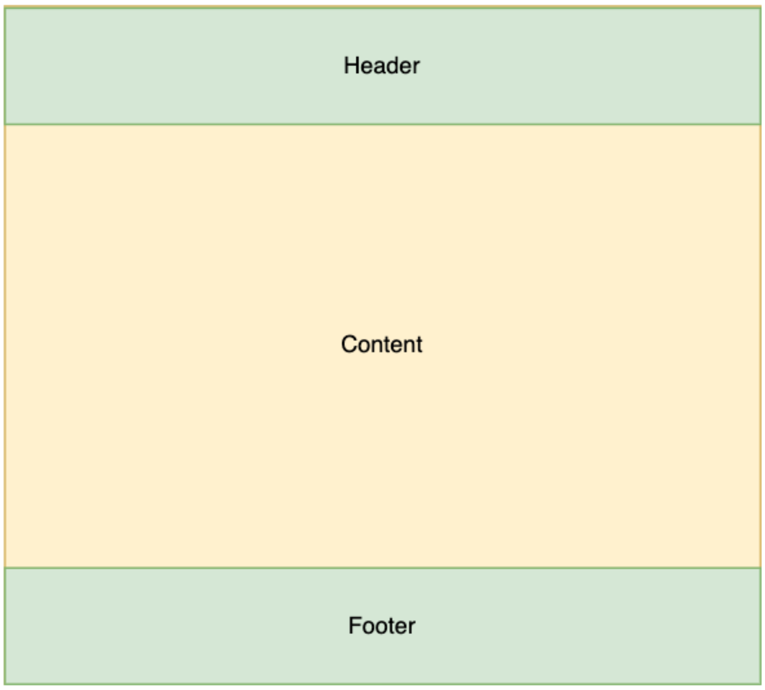
Client-side validation occurs in the user's browser, providing immediate feedback as they interact with the form. This enhances the user experience by preventing submission of invalid data and allowing for quick corrections without a round trip to the server. However, client-side validation relies on JavaScript, which can be disabled or bypassed, making it inherently less secure.

Server-side validation, on the other hand, takes place on the server after the data has been submitted. It is crucial for ensuring the integrity and security of the application, as it verifies all input regardless of client-side manipulations. Because server-side validation is executed in a controlled environment, it cannot be bypassed and protects against malicious inputs and vulnerabilities such as SQL injection and XSS attacks. Therefore, while client-side validation can improve usability, server-side validation must always be implemented to guarantee robust security and data integrity.

# 11. Layout

## 11.1. Problem

Most websites often have elements that do not change while we traverse between different pages, such as the header and footer. Repeating the same HTML in every view would be hard to maintain in the future. Usually, only a portion of the website changes as a result of user interaction.



## 11.2. Solution

Layouts in ASP.NET are used to define a common structure for multiple views, providing a consistent look and feel across the entire web application. They act as a template that contains the common elements (such as headers, footers, and navigation bars) which are shared by multiple views.

**How Layouts Work:****
A layout is typically defined in a file named `_Layout.cshtml`, which is located in the `Views/Shared` folder. This file serves as the master template for the views that use it.

Example of a basic `_Layout.cshtml` file:

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title - My ASP.NET Application</title>
    <link rel="stylesheet" href="~/css/site.css" />
    <script src="~/js/site.js"></script>
</head>
<body>
    <header>
        <h1>My ASP.NET Application</h1>
        <nav>
            <ul>
                <li><a asp-controller="Home" asp-action="Index">Home</a>
</li>
                <li><a asp-controller="Home" asp-action="About">About</a>
</li>
                <li><a asp-controller="Home" asp-
action="Contact">Contact</a></li>
            </ul>
        </nav>
    </header>
    <main>
        @RenderBody()
    </main>
    <footer>
        <p>&copy; 2024 - My ASP.NET Application</p>
    </footer>
</body>
</html>
```

To use a layout in a view, you set the `Layout` property at the top of the view. By default, new views created in ASP.NET are configured to use `_Layout.cshtml`.

Example of a view using the layout:

```
@model MyApp.Models.MyModel
@{
    Layout = "_Layout";
}

<h2>My View Content</h2>
<p>This is the content of my view.</p>
```

RenderBody and Sections:

- `@RenderBody()`: This method is called within the layout file and acts as a placeholder where the content of the child view will be inserted.
- `@RenderSection()`: This method is used to define optional sections that can be implemented by the child views. For example, a script section might be defined in the layout to allow views to include additional scripts.

Example of defining and using a section:

```
<!-- In _Layout.cshtml -->
<body>
    ...
    <main>
        @RenderBody()
    </main>
    @RenderSection("Scripts", required: false)
    ...
</body>
```

# 12. Architecture, entity framework

All the elements related to architecture that we have talked about are still valid. We still have business logic and infrastructure logic. Now we simply replaced the representation layer from JSON (REST API) to HTML, CSS, and JS. This means that we can still use the domain layer, services, repositories, etc.