В ASP.NET MVC общий макет (layout) — это шаблон, который используется для создания общего интерфейса и структуры страниц приложения. Макет служит основной «оболочкой» для представлений, позволяя им наследовать общий стиль, меню, заголовок, подвал и другие элементы, которые должны быть одинаковыми на разных страницах.

Обычно макетный файл расположен в папке Views/Shared и имеет название Layout.cshtml по умолчанию.

Главное же отличие от обычных представлений состоит в использовании метода **@RenderBody()**, который является плейсхолдером и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу. В итоге мы сможем легко установить для всех представлений веб-приложения единообразный стиль оформления.

то есть если я хочу сделать общий макет с шапкой и футером. То я создаю представление с любым названием, закидываю в него код шапки и футера. Между ними вставляю @RenderBody(). Потом в обычном представлении прописываю @{ Layout = "~/Views/...путь\_к\_макету"; }.

Если в представлении не указать свойство Layout, то:

1. **Будет использован макет по умолчанию**, если он задан в настройках приложения.

В ASP.NET MVC можно указать глобальный макет в файле ViewStart.cshtml, который обычно находится в папке Views. Этот файл применяется ко всем представлениям, которые не задают собственный макет.

Файл \_ViewStart.cshtml (обязательное название) в ASP.NET MVC представляет собой специальное представление, которое выполняется **до рендеринга каждого представления** и позволяет задать общие настройки для всех представлений приложения, такие как Layout.

Файл \_ViewImports.cshtml в ASP.NET MVC и ASP.NET Core MVC используется для **подключения пространств имен, вспомогательных функций и тег-хелперов**, которые будут применяться ко всем представлениям в определенной папке и ее подпапках.

* \_**ViewStart.cshtml** используется для настройки представлений перед их рендерингом, например, для указания общего макета (Layout) и задания глобальных параметров (например, ViewBag или ViewData). Этот файл выполняется **перед каждым представлением**, а его код интерпретируется и выполняется при каждом запросе.
* **\_ViewImports.cshtml** управляет **импортом пространств имен и тег-хелперов** и действует как статический импорт. Пространства имен, добавленные через \_ViewImports.cshtml, доступны **сразу для всех представлений в папке и подпапках** без необходимости их подключать заново.

Кроме обычных представлений и мастер-страниц в ASP.NET Core MVC также можно использовать частичные представления или **partial views**. Их отличительной особенностью является то, что их можно встраивать в другие обычные представления.

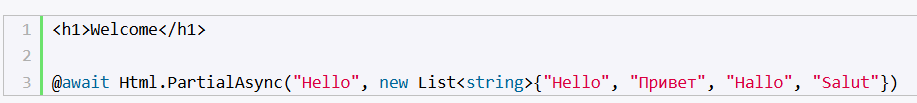
За рендеринг частичных представлений отвечает объект **PartialViewResult**, который возвращается методом **PartialView()**. Этот метод имеет четыре версии:

* PartialView(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
* PartialView(string? viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление
* PartialView(object? model): передает в представление данные в виде объекта model
* PartialView(string? viewName, object? model): переопределяет имя представления и передает в него данные в виде объекта model

Метод **Html.PartialAsync()** встраивает код частичного представления в обычное. Он является асинхронным и возвращает объект IHtmlContent, который представляет html-содержимое и который обернут в объект **Task<TResult>**. В качестве параметра в метод передается имя представления.

Кроме метода Html.PartialAsync() частичное представление можно встроить с помощью другого метода - **Html.RenderPartialAsync**. Этот метод также принимает имя представления, только он используется не в строчных выражениях кода Razor, а в блоке кода, то есть обрамляется фигурными скобками.

Одна из перегруженных версий методов Html.PartialAsync / Html.RenderPartialAsync позволяет передать модель в частичное представление. В итоге у нас получится стандартное строго типизированное представление. Например, в качестве второго параметра список строк:



* **Html.PartialAsync()**: можно вызывать как обычное Razor-выражение @Html.PartialAsync(...). Подходит для простых случаев и интеграции небольших частичных представлений в HTML-код.
* **Html.RenderPartialAsync()**: требует блока @{ }, так как не возвращает результат, а напрямую рендерит HTML. Рекомендуется для более крупных частичных представлений или в случаях, где важна оптимизация производительности.
* В частичных представлениях обычно **не указывается Layout**, так как они встроены в родительское представление и должны следовать его структуре.
* Если в частичном представлении прописать Layout, это может вызвать конфликт, так как оно начнет пытаться рендериться как самостоятельная страница, а не как часть другой страницы.
* Html.BeginForm создает форму. Она принимает параметры:
  + action — имя действия контроллера, к которому отправится форма при POST.
  + "Calc" — имя контроллера, где расположен обработчик запроса.
  + FormMethod.Post — указывает, что данные отправляются методом POST.

HTML-хелперы в ASP.NET Core MVC — это методы, упрощающие генерацию HTML-разметки на стороне сервера.

Примеры хелперов:

**Html.BeginForm** и **Html.EndForm** — создают HTML-форму и автоматически добавляют открывающий и закрывающий теги <form>.

**Html.LabelFor()** — создает метку (тег <label>) для свойства модели.

**Html.TextBoxFor()** — создает текстовое поле (тег <input type="text">), автоматически привязанное к свойству модели.

Пример как можно написать свой хелпер:

@helper BookList(IEnumerable<BookStore.Models.Book> books)

{

    <ul>

        @foreach (BookStore.Models.Book b in books)

        {

            <li>@b.Name</li>

        }

    </ul>

}

using System;

using System.Web;

using System.Web.Mvc;

using System.Linq;

namespace BookStore.Helpers

{

public static class ListHelper

{

public static MvcHtmlString CreateList(this HtmlHelper html, string[] items)

{

TagBuilder ul = new TagBuilder("ul");

foreach (string item in items)

{

TagBuilder li = new TagBuilder("li");

li.SetInnerText(item);

ul.InnerHtml += li.ToString();

}

return new MvcHtmlString(ul.ToString());

}

}

}