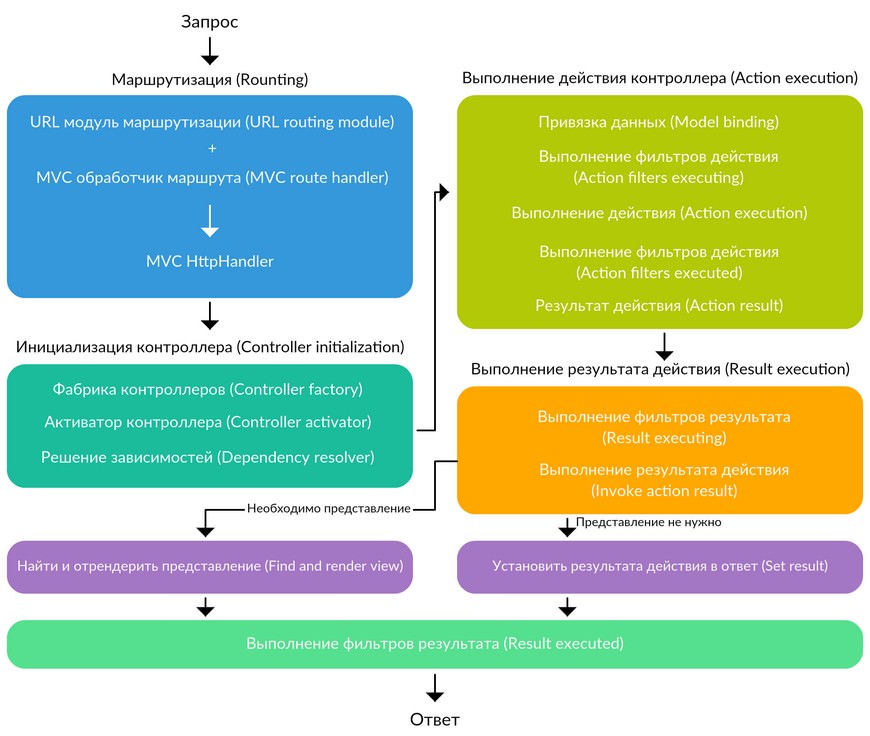
**ASP.NET Core** представляет платформу (**набор средств разработки, библиотек, ускоряющих разработку приложений**) для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языков программирования для разработки приложений на ASP.NET Core используются C# и F#.ASP.NET Code является кроссплатформенным фреймворком с открытым кодом.

**ASP.NET Core:** **wwwroot** – папка для статического контента (html, css, js,…), **статический контент не отображается по умолчанию**.

По умолчанию система маршрутизации проверяет, соответствует ли URL дисковому файлу, перед оценкой маршрутов приложения.



**Жизненный цикл запроса в ASP.NET Core**



Фреймворк **ASP.NET Core MVC** является частью платформы ASP.NET Core, его отличительная особенность - применение паттерна MVC.

1. **Маршрутизация**:

Первой остановкой запроса является подсистема маршрутизации, ведь необходимо определить, какой цели должен достигнуть наш запрос, какое действие и какого контроллера является для него желанным. Необходимо получить некую пару "ключ – значение" для маршрута и единицы, которая будет его обрабатывать дальше. Эта единица называется "**обработчик запроса**" (**route handler**) и ее задача вернуть объект **HttpHandler**, который **будет выполнять всю дальнейшую цепочку обработки запроса**.

1. **Инициализация контроллера**

Метод **ProcessRequest** отвечает за генерацию ответа на входящий запрос. Для этого он должен создать и запустить контроллер. Сначала он вызывает дочерний метод с именем ProcessRequestInit, который запрашивает у компонента, называемого фабрикой контроллеров, контроллер для текущего запроса. **Фабрика контроллеров выбирает соответствующий класс контроллера из текущего приложения, используя переданные данные маршрута**. Затем фабрика использует компонент, называемый **активатором контроллера**, для фактического создания **экземпляра этого класса**. Он использует внедрение зависимостей (Dependency Injection, DI) для создания контроллера с учетом его зависимостей. Например, если в конструкторе контроллера требуется какой-то сервис, активатор получит его через контейнер зависимостей.

это позволяет реализовать такие шаблоны, как внедрение зависимостей, после того, как MvcHandler получил контроллер на этих этапах, он вызывает метод Execute на этом контроллере. Именно из этого метода открываются все мощные инструменты, такие как методы действий, и начинается их обработка.

1. **Выполнение действий контроллера**

После создания контроллера компонент под названием **action invoker** находит и выбирает подходящий метод действия для вызова на нашем контроллере, для этого берутся за ключевые фильтры: имя действия, параметры действия, атрибуты (HttpGet, HttpPost). При выполнении действий контроллера, так же выполняются фильтры.

1. **Выполнение результата действия**

В результате выполнения всей цепочки мы получаем объект **ActionResult**. Если результатом является тип представления, будет вызван механизм просмотра, который найдет представление и отобразит его. Если это не тип представления, то результат действия будет выполнен сам по себе. Это выполнение результата является ничем иным, как генерированием фактического ответа на исходный HTTP-запрос. Объект [ActionResult](https://goo.gl/JN5tvf) записывает свои данные в ответ запросу.

* **ExecuteResult** — это метод, который вызывается для любого результата, который возвращает контроллер. В случае ViewResult, именно ExecuteResult запускает процесс рендеринга представления.
* Когда метод действия возвращает ViewResult, MVC фреймворк вызывает его метод ExecuteResult. Этот метод:
  + Ищет представление с помощью **ViewEngines**.
  + Рендерит найденное представление, передавая в него данные модели.
  + Записывает готовый HTML-результат в **HTTP-ответ** для клиента.

**Описание MVC**

Концепция паттерна MVC предполагает разделение приложения на три компонента:

**Модель (model):** описывает используемые в приложении данные, а также логику, которая связана непосредственно с данными, например, логику валидации данных. Как правило, объекты моделей хранятся в базе данных.

В MVC модели представлены двумя основными типами: модели представлений, которые используются представлениями для отображения и передачи данных, и модели домена, которые описывают логику управления данными.

Модель может содержать данные, хранить логику управления этими данными. В то же время модель не должна содержать логику взаимодействия с пользователем и не должна определять механизм обработки запроса. Кроме того, модель не должна содержать логику отображения данных в представлении.

**Представление (view):** отвечают за визуальную часть или пользовательский интерфейс, нередко html-страница, через который пользователь взаимодействует с приложением. Также представление может содержать логику, связанную с отображением данных. В то же время представление не должно содержать логику обработки запроса пользователя или управления данными.

**Контроллер (controller):** представляет центральный компонент MVC, который обеспечивает связь между пользователем и приложением, представлением и хранилищем данных. Он содержит логику обработки запроса пользователя. Контроллер получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления, наполненного данными моделей.

**Структура проекта**

**Dependencies**: все добавленные в проект пакеты и библиотеки

**wwwroot**: этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.

**Controllers**: папка для хранения контроллеров, используемых приложением. По умолчанию здесь уже есть один контроллер - Homecontroller

**Models**: каталог для хранения моделей. По умолчанию здесь создается модель ErrorviewModel

**Views**: каталог для хранения представлений. Здесь также по умолчанию добавляются ряд файлов - представлений

**appsettings.json**: хранит конфигурацию приложения

**Program.cs**: файл, который определяет входную точку в приложение ASP.NET Core

builder.Services.AddControllersWithViews(); - добавляется поддержка контроллеров и представлений (views). Это указывает приложению, что оно будет использовать архитектуру MVC.

**Маршрутизация**

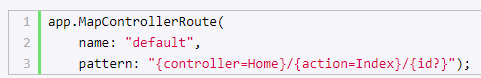
**Маршрутизация** в ASP.NET Core — это механизм, который отвечает за обработку запросов к веб-приложению. Она определяет, какой контроллер и какое действие должны быть вызваны в зависимости от URL, который ввёл пользователь.

**MapControllerRoute()** – добавить маршрут.

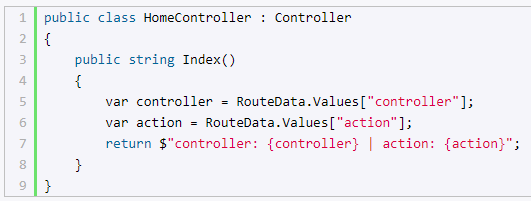
Шаблон маршрута использует три параметра. Параметр "controller" будет сопоставляться по имени с одним из контроллеров приложения, а параметр "action" - с действием этого контроллера.

* **Defaults** — это параметр, который задаёт значения по умолчанию для сегментов маршрута, если они отсутствуют в URL. Эти значения используются, когда определённый сегмент маршрута не указан в запросе.

**MapDefaultControllerRoute()** эквивалентен вызову



Мы можем получить в контроллере все параметры маршрута, используя объект **RouteData.**



**Контроллер**

Ключевым элементом контроллера являются его **действия**. Действия контроллера - это **публичные** методы, которые могут сопоставляться с запросами. Сопоставление запроса с контроллером и его действием происходит благодаря системе маршрутизации.

Чтобы указать, что этот класс не является контроллером, нам надо использовать над ним атрибут **[NonController].** Аналогично, если мы хотим, чтобы какой-либо публичный метод контроллера не рассматривался как действие, то мы можем использовать над ним атрибут**NonAction. Атрибут [ActionName]** позволяет для метода задать другое имя действия.



Кроме того, методы могут обслуживать разные типы запросов. Для указания типа запроса HTTP нам надо применить к методу один из атрибутов: **[HttpGet]**, **[HttpPost]**, **[HttpPut]**, **[HttpPatch]**, **[HttpDelete]** и **[HttpHead]**.

Про контекст контроллера:

Для получения контекста в классе контроллера нам доступно свойство **ControllerContext**, которое представляет одноименный класс ControllerContext. Этот класс определяет ряд важный свойств:

* **HttpContext**: содержит информацию о контексте запроса
* **ActionDescriptor**: возвращает дескриптор действия - объект ActionDescriptor, который описывает вызываемое действие контроллера
* **ModelState**: возвращает словарь ModelStateDictionary, который используется для валидации данных, отправленных пользователем
* **RouteData**: возвращает данные маршрута

Свойство **HttpContext.Response** представляет объект **HttpResponse** и позволяет управлять ответом на запрос, в частности, устанавливать заголовки ответа, куки, отправлять в выходной поток некоторый ответ. Этот же объект доступен через свойство **Response** класса Conroller.

Свойство **HttpContext.Response** представляет объект **HttpResponse** и позволяет управлять ответом на запрос, в частности, устанавливать заголовки ответа, куки, отправлять в выходной поток некоторый ответ. Этот же объект доступен через свойство **Response** класса Conroller.

Свойство **HttpContext.Request** представляет объект **HttpRequest** и предоставляет разнообразную информацию о запросе. Этот же объект доступен через свойство **Request** класса контроллера.

Класс контроллера должен удовлетворять как минимум одному из следующих условий:

* Класс контроллера имеет суффикс "Controller"
* Класс контроллера наследуется от класса, который имеет суффикс "Controller"
* К классу контроллера применяется атрибут [Controller]

IActionResult — это **интерфейс**, который используется в ASP.NET Core MVC для представления результата действия (метода контроллера). Он определяет результат выполнения метода контроллера и указывает, что будет возвращено в ответ на HTTP-запрос.

**Зачем нужен IActionResult:**

Этот интерфейс позволяет методам контроллера возвращать различные типы результатов, обеспечивая гибкость при формировании ответа. В зависимости от ситуации, контроллер может вернуть:

* Представление (View),
* Перенаправление (Redirect),
* Статус код (например, 404 Not Found или 500 Internal Server Error),
* **View()**:
  + Это один из возможных типов результата, который возвращает представление (HTML-страница) для запроса. Этот метод ищет соответствующий файл представления (например, Index.cshtml) и возвращает его.

**Представление**

Подобный проект для хранения представлений в папке **Views** определяет некоторую структуру:

* Во-первых, как правило, для каждого контроллера в проекте создается подкаталог в папке Views, который называется по имени контроллера и который хранит представления, используемые методами данного контроллера. Так, по умолчанию имеется контроллер HomeController и для него в папке Views есть подкаталог **Home** с представлениями для методов контроллера HomeController - в данном случае это файлы **Index.cshtml** и **Privacy.cshtml**.
* Также здесь есть папка **Shared**, которая хранит общие представления для всех контроллеров. По умолчанию это файлы **\_Layout.cshtml** (используется в качестве мастер-страницы), **Error.cshtml** (использутся для отображения ошибок) и **\_ValidationScripsPartial.cshtml** (частичное представление, которое подключает скрипты валидации формы).
* И в корне каталога **Views** также можно найти два файла **\_ViewImports.cshtml** и **\_ViewStart.cshtml**. Эти файлы содержат код, который автоматически добавляется ко всем представлениям. *\_ViewImports.cshtml* устанавливает некоторые общие для всех представлений пространства имен, а *\_ViewStart.cshtml* устанавливает общую мастер-страницу.

За работу с представлениями отвечает объект **ViewResult**. Он производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту. Чтобы возвратить объект ViewResult, в методе контроллера вызывается метод **View**:

Вызов метода View возвращает объект ViewResult. Затем уже ViewResult производит рендеринг определенного представления в ответ. По умолчанию контроллер производит поиск представления в проекте по следующим путям:

* /Views/Имя\_контроллера/Имя\_представления.cshtml
* /Views/Shared/Имя\_представления.cshtml

**ViewBag** — это динамический объект, который позволяет передавать данные из контроллера в представление (view) без необходимости создавать конкретные модели.

Метод Redirect() выполняет следующие задачи:

* Возвращает **HTTP-ответ с кодом 302**. Этот код указывает браузеру или клиенту, что страница временно перемещена, и нужно выполнить запрос на новый URL.
* Перенаправляет пользователя на указанный URL (в данном случае на /index.html).

**Виды маршрутизации**

**Маршрутизация на основе условностей / соглашений** — это подход, при котором маршруты определяются в конфигурации приложения на основе заранее установленных правил. MapControllerRoute.

**Маршрутизация на основе атрибутов** позволяет разработчикам определять маршруты непосредственно в контроллерах и методах действий с использованием атрибутов.

**Таблица маршрутов (routing table)** — это набор правил, которые определяют, как запросы URL-адресов сопоставляются с действиями контроллеров. **Маршрутизация** — это процесс сопоставления URL-запросов с определёнными контроллерами и действиями (методами) в приложении.

Порядок, в котором роуты добавляются в таблицу маршрутизации, определяет порядок, в котором они будут найдены при поиске соответствий. Это значит, что роуты в исходном коде должны быть перечислены по приоритету: от высшего, с наиболее конкретными условиями, до самого низкого, или *catchall* роута.

Сейчас мы добавим в пример приложения *catchall* роут, предназначенный для URL, которые не соответствует никакому другому роуту. Цель этого роута - показать сообщение об ошибке HTTP 404. Глобальные *catchall* роуты будут соответствовать любому роуту, и они должны быть определены в *последнюю очередь*:

Значение catchall заменяет значение, которое было передано в *catchall* роут. В отличие от обычных параметров роутов, *catchall* параметры (с префиксом звездочкой) охватывают целую часть URL, включая слеши, которые обычно используются для разделения параметров роута.

1. Сравните свойство **Order** атрибута маршрута.
2. Просмотрите каждый сегмент URI в шаблоне маршрута. Для каждого сегмента упорядочено следующим образом:

#### **1. Литеральные сегменты**

**Литеральные сегменты** — это самые специфичные и, следовательно, наиболее приоритетные маршруты. Литеральным сегментом считается любой текст, который напрямую сопоставляется с частью URL.

**2. Параметры маршрута с ограничениями**

После литеральных сегментов следующим по приоритету идёт обработка **параметров маршрута с ограничениями**. Ограничения параметров маршрута позволяют указать, каким должен быть параметр, например, тип данных или формат.

**3. Параметры маршрута без ограничений**

Если нет точного литерального совпадения или параметров с ограничениями, то следующий по приоритету маршрут — это **параметры без ограничений**. Это более общее определение, где параметры маршрута могут принимать любые значения без ограничений на тип данных.

1. Литеральные сегменты – это части маршрута, которые являются статическими и явно указаны в URL.
2. Параметры маршрута с ограничениями.
3. Параметры маршрута без ограничений.
4. Сегменты параметров с catch all с ограничениями.
5. Сегменты параметров с catch all без ограничений.

В ASP.NET Core MVC параметры, переданные через URL, можно получить несколькими способами, в зависимости от того, как они переданы в запросе.

**1. Параметры из строки запроса (Query string)**

Если параметры передаются через строку запроса (например, http://example.com/home/index?id=5&name=John), их можно получить с помощью свойства Request.Query.

string id = Request.Query["id"];

string name = Request.Query["name"];

**2. Параметры из маршрута (Route parameters)**

Если параметры передаются через URL как часть маршрута (например, http://example.com/home/index/5), их можно получить как параметры метода действия.

**3. Параметры формы или тела запроса (Form or Body parameters)**

Если параметры передаются через тело POST-запроса (например, при отправке формы), их можно получить с помощью Request.Form.

name = Request.Form["name"];

**4. Использование привязки модели (Model binding)**

Также можно использовать привязку модели, чтобы ASP.NET Core автоматически привязывал параметры из строки запроса, маршрута, тела запроса и т.д. к аргументам метода действия или объектам.

public IActionResult Index(int id, string name)

{

// ASP.NET Core автоматически свяжет параметры id и name

return View();

}

Если параметры передаются через форму, можно привязать их к модели:

public class MyModel

{

public int Id { get; set; }

public string Name { get; set; }

}

[HttpPost]

public IActionResult Index(MyModel model)

{

// Используйте модель, автоматически привязанную ASP.NET Core

return View();

}

* В конвейере обработки запросов (**middleware pipeline**) маршрутизация и поиск конечных точек (endpoints) выполняются **на разных этапах**.
* На этапе вызова UseRouting() система "запоминает" все маршруты, но ещё **не обрабатывает** запросы.
* **Обработка** происходит именно на этапе UseEndpoints(). Здесь система фактически **проверяет** все ранее зарегистрированные маршруты и выполняет соответствующие действия.

В последних версиях ASP.NET Core (начиная с .NET 6), был введён **минимальный хостинг-модель** (Minimal Hosting Model), которая упрощает настройку приложения и изменяет способ конфигурации конвейера обработки запросов. Хотя ты явно не видишь UseEndpoints, **внутри вызовов MapControllerRoute он уже встроен**. Это стало возможным благодаря тому, что вызов MapControllerRoute фактически **регистрирует конечные точки в маршрутизаторе** и интегрирует их в конвейер обработки запросов. При этом система автоматически включает необходимый функционал для маршрутизации на основе твоих маршрутов.

 **В старой модели**:

* **UseEndpoints** был шагом, где ты явно регистрировал все конечные точки маршрутизации, будь то для контроллеров, API, SignalR, или других частей приложения.
* **MapControllerRoute** внутри UseEndpoints определял отдельные маршруты для контроллеров.

 **В новой модели (с .NET 6)**:

* **MapControllerRoute** теперь сам по себе делает всё, что раньше требовалось от UseEndpoints. Он не только добавляет маршрут, но и выполняет настройку конечных точек в контексте всего приложения.
* **UseEndpoints больше не нужен явно**, потому что новая система автоматически регистрирует конечные точки маршрутизации при вызове MapControllerRoute.
* **Маршрутизация**: Это механизм **поиска правильной конечной точки** на основе URL.
* **Конечная точка**: Это **конечный обработчик**, к которому направляется запрос после маршрутизации.

Альтернативы MapControllerRoute:

* 1. MapDefaultControllerRoute
  2. MapAreaControllerRoute. Если у вас есть области (areas) в приложении, вы можете использовать этот метод для маршрутизации контроллеров внутри этих областей.



* 1. MapControllers. Этот метод автоматически обрабатывает маршруты, основываясь на атрибутах, определённых в контроллерах.
  2. MapGet, MapPost, MapPut и MapDelete. Эти методы позволяют создать маршруты для обработки HTTP-запросов определённых методов.

