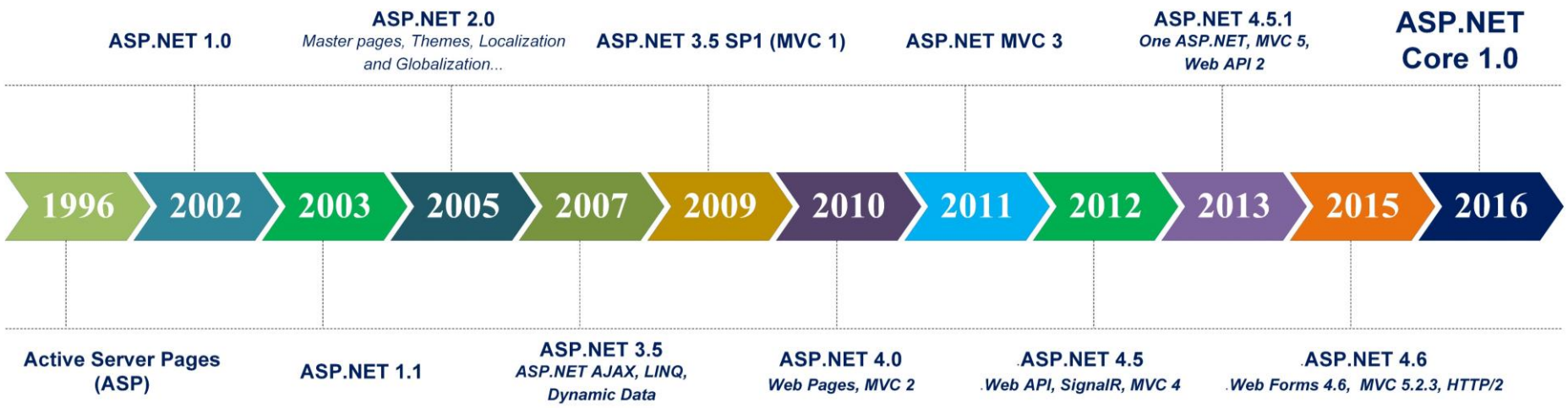
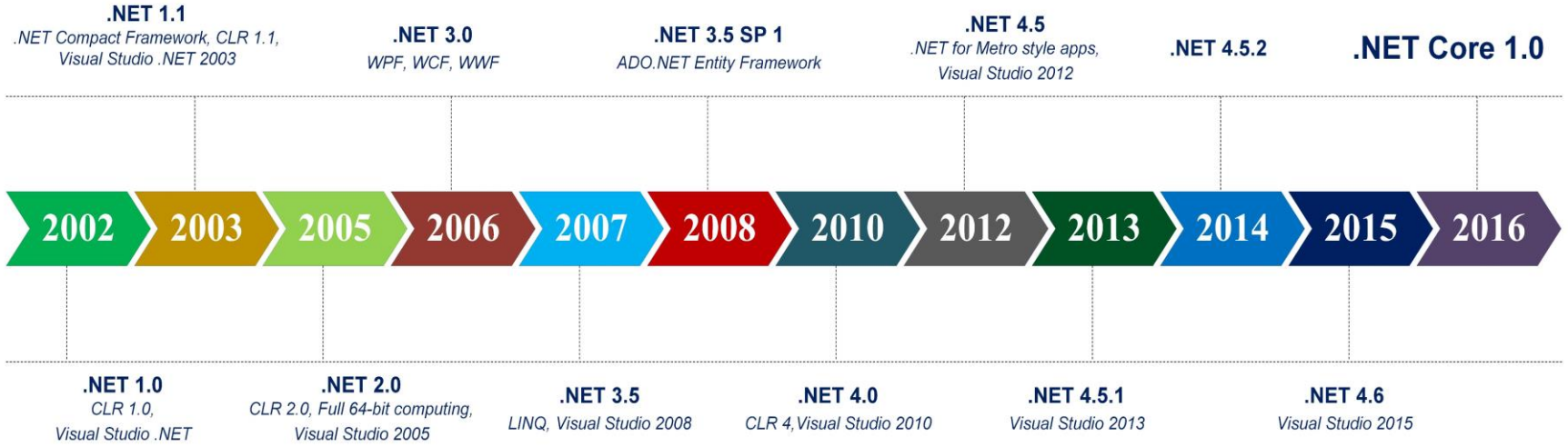
# Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

## Asp.Net Core MVC.
## Model.
## View.
## Controller.

# История развития: от классического ASP до ASP.Net Core

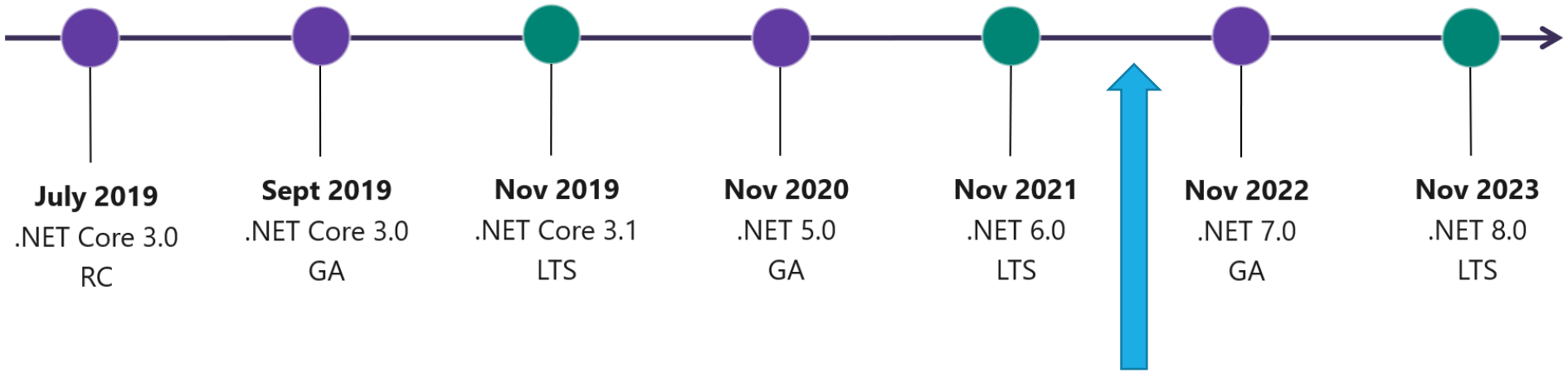



## Синхронизация с .Net Framework:




https://codingsight.com/asp-net-core-1-0-part-1-introduction-general-description-future-net-framework/

# ASP.Net Core: переход на open source, текущее состояние и планы

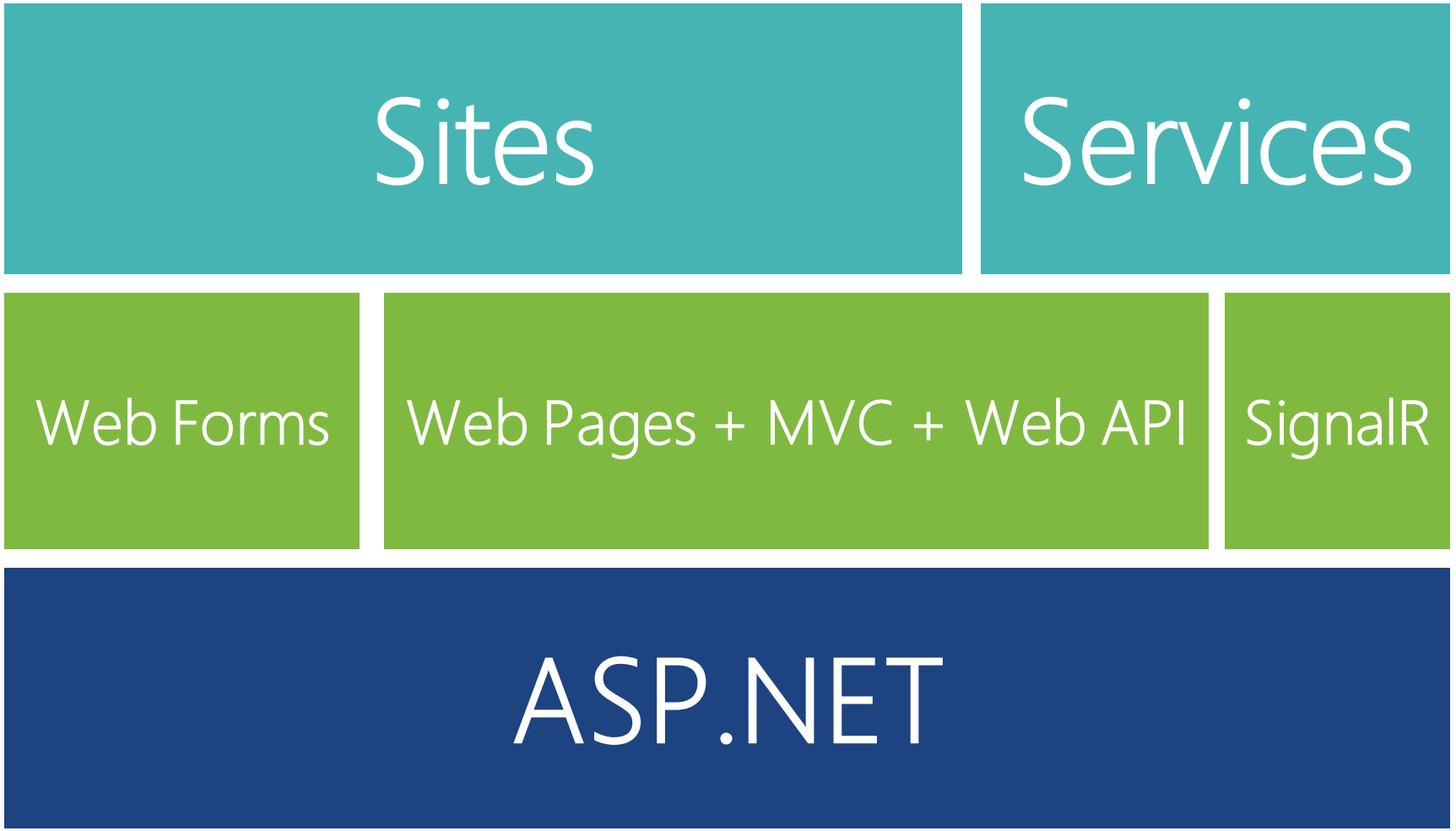



GA = General Availability (общедоступность)

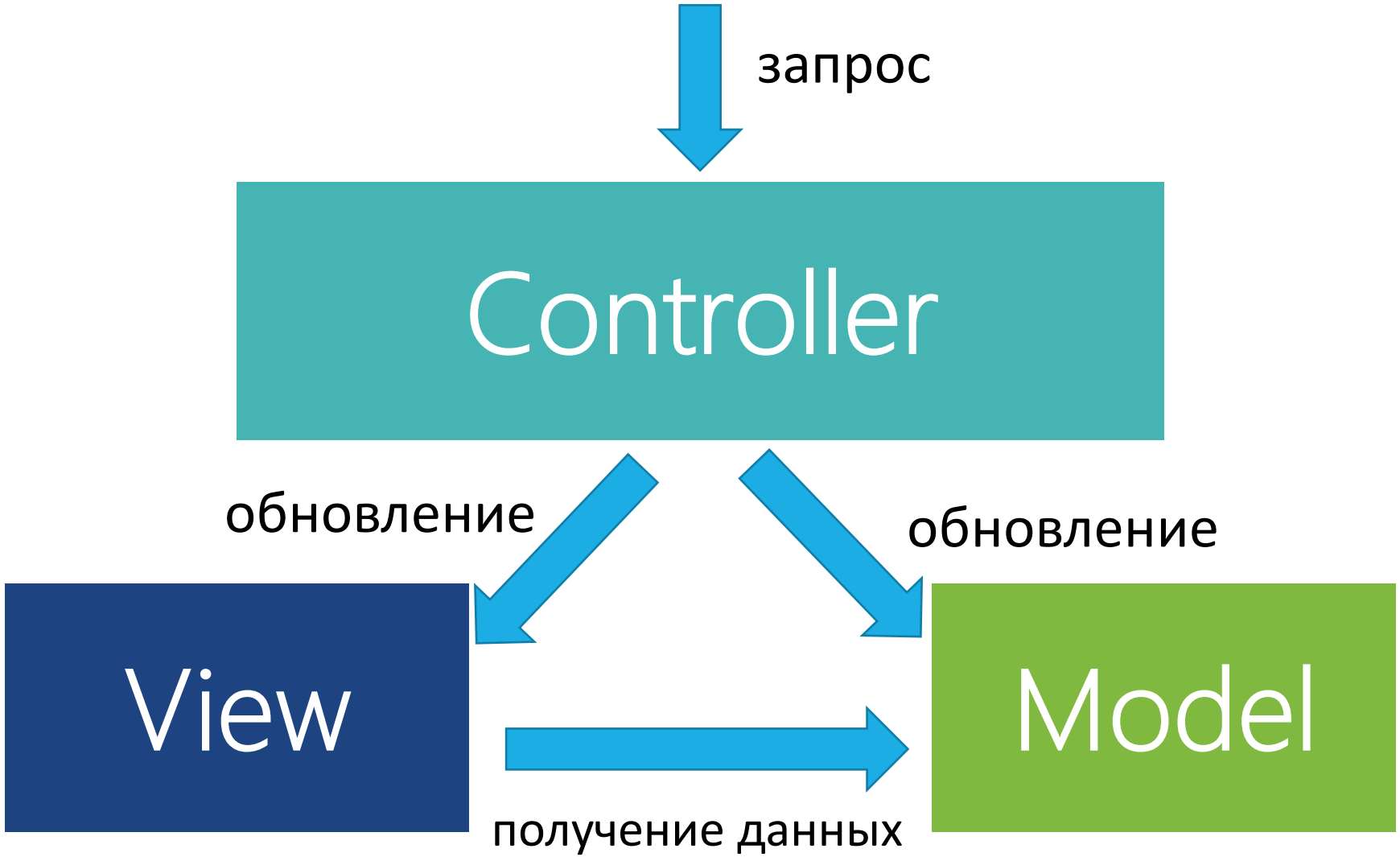
LTS = Long Term Support (долгосрочная поддержка)

https://dotnet.microsoft.com/platform/support/policy/dotnet-core/

# ASP.Net и технологии построения приложений

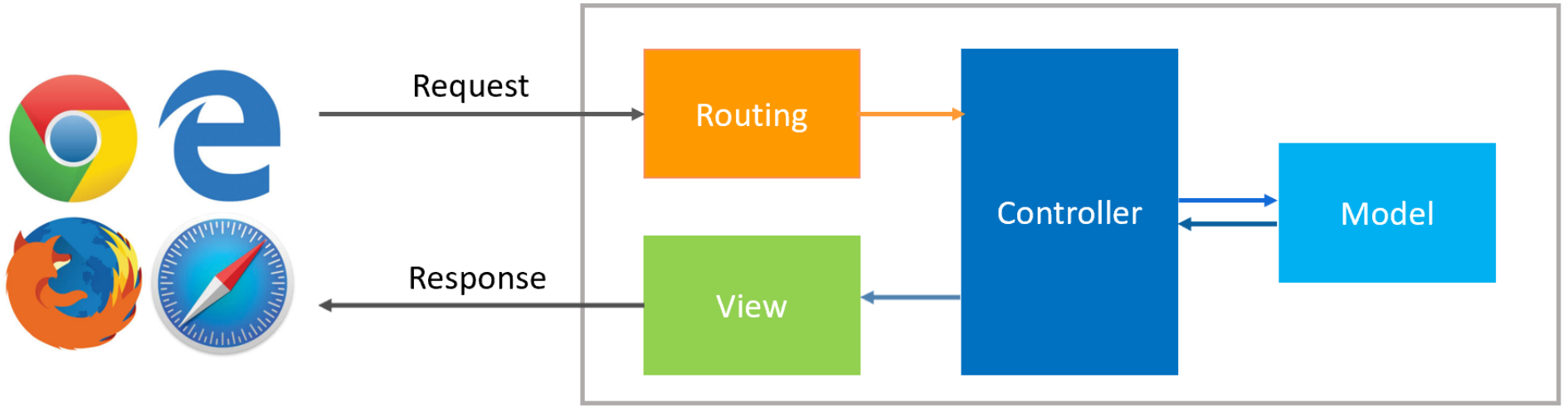


# Шаблон проектирования Model-View-Controller (MVC)

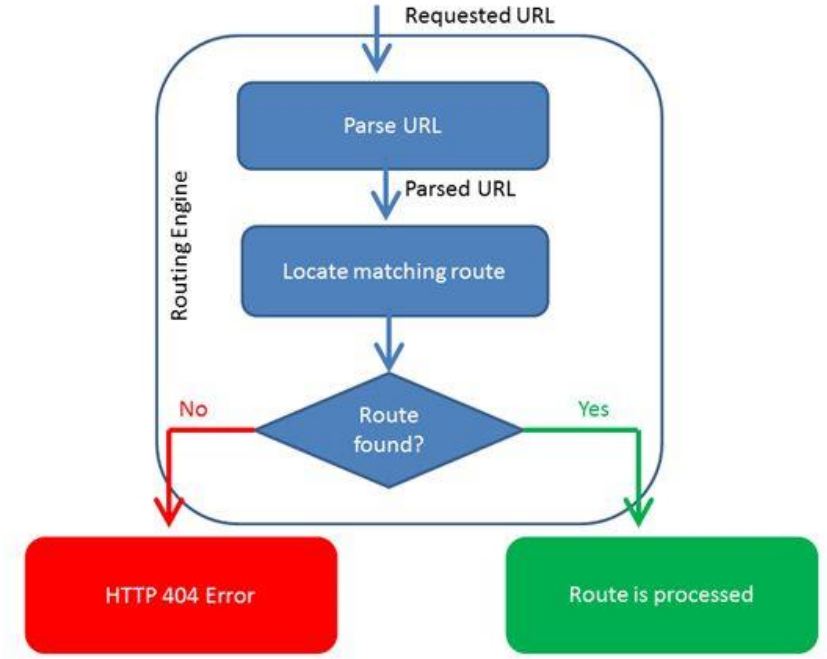



Controller – контроллер.
View – представление.
Model – модель (структура данных).

# Маршрутизация и обработка запроса




# Маршрутизация запроса



# Маршрутизация в ASP.Net Core MVC

**Способы маршрутизации:**

- с помощью конечных точек (endpoints);

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

- с помощью промежуточных обработчиков маршрутизации (router middleware);

```
app.UseMvc(routes => {
    routes.MapRoute(name: "default", template:
    "{controller=Home}/{action=Index}/{id?}");
});
```

- с помощью атрибутов (attribute-based, т.е. декларативное).

```
[Route("main/index/{name}")]
public IActionResult Index(string name) =>
    Content(name);
```

https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing?view=aspnetcore-5.0

# Контроллер в ASP.Net Core MVC

Контроллер - класс, который наследуется от абстрактного базового класса *Microsoft.AspNetCore.Mvc.Controller* и отвечает за обработку данных и отправку клиенту результата обработки.

**Правила и соглашения:**

- помещаются в папку **Controllers**
- названия классов контроллеров должны оканчиваться на "**Controller**" (префикс считается именем контроллера).
- **публичные** методы – это действия (action), которые могут сопоставляться с запросами при маршрутизации.

# Контроллер и полезные атрибуты

**Атрибуты и их назначение:**

- **NonController –** указание, что класс – не является контролером

```
[NonController]
public class HomeController : Controller { ... }
```

- **NonAction –** public-метод не рассматривается как действие:

```
[NonAction]
public string Hello() => "Hello ASP.NET";
```

- **ActionName** – задаем имя действия для метода (отработает /Home/Welcome/ вместо /Home/Hello/):

```
[ActionName("Welcome")]
public string Hello() => "Hello ASP.NET";
```

# Методы контроллера и типы запросов

Два метода в соответствии с именем образуют одно действие Buy, но для разных типов запросов и параметров.

```
[HttpGet]
public IActionResult Buy(int id) =>
    View(new Order { PhoneId = id });


[HttpPost]
public string Buy(Order order) {
    db.Orders.Add(order);
    db.SaveChanges();
    return "Спасибо, " + order.User + ", за покупку!";
}
```

Методы в рамках одного действия могут обслуживать разные запросы. Для указания типа HTTP-запроса применяем атрибуты (если не указано - HttpGet):

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]

# Передача данных в контроллер

Система привязки MVC сопоставляет параметры запроса и параметры метода **по имени**. Т.е. для параметра ***a*** в строке запроса, его значение будет передаваться именно параметру метода с именем ***a*** (при этом должен соответствовать тип).

```
public string Square(int a = 3, int h = 10)
{
    return $"Площадь треугольника (a={a}; h={h}) = {a * h / 2.0}";
}
```

Можно передавать и массивы, например, при вызове */Home/Sum?nums=1&nums=2&nums=3*
вызовется метод, а в массиве окажется 3 элемента:

```
public string Sum(int[] nums)
{
    return $"Сумма чисел равна {nums.Sum()}";
}
```

***Внимание!***
Помните, что получить все данные можно из контекста запроса через **Request.**

# Результаты действий контроллера

По типу ответа из метода действия контроллера инфраструктура MVC определяет формат для отправки клиенту:

- при возврате строки / числа / DateTime из метода – строка-результат отправляется клиенту “как есть”, а Content-Type устанавливается в text/plain;
- при возврате объектов классов, данные форматируются в JSON, а Content-Type устанавливается в application/json;
- с помощью метода Content() можно вернуть объект типа string;
- с помощью метода Json() можно вернуть объект, сериализованый в формат JSON.

Однако, в большинстве случаев, тип возврата – объект, поддерживающий интерфейс IActionResult (из ***Microsoft.AspNet.Mvc***), который предназначен для генерации результата действия.

# Результаты действий контроллера, примеры

Примеры простых ответов:

- **ContentResult**: пишет указанный контент напрямую в ответ в виде строки;
- **EmptyResult**: отправляет пустой ответ со кодом HTTP 200;

```
public IActionResult GetVoid()
{
    return new EmptyResult();
}
```

- **NoContentResult**: похож на EmptyResult и отправляет пустой ответ, но с кодом HTTP 204;

```
public IActionResult GetVoid()
{
    return new NoContentResult();
}
```

**Примеры для отправки файлов:**

- **FileResult**: является базовым классом для всех объектов, которые пишут набор байтов в выходной поток. Предназначен для отправки файлов
- **FileContentResult**: класс, производный от FileResult, пишет в ответ массив байтов.

# Результаты действий контроллера, примеры

- **JsonResult**: возвращает в качестве ответа объект или набор объектов в формате JSON.
- **PartialViewResult**: производит рендеринг частичного представления в выходной поток.
- **RedirectResult**: перенаправляет пользователя по другому адресу URL, возвращая статусный код 302 для временной переадресации или код 301 для постоянной переадресации зависимости от того, установлен ли флаг Permanent.
- **ViewResult**: производит рендеринг представления и отправляет результаты рендеринга в виде html-страницы клиенту.

**Пример с JSON:**

```
public class User {
    public string Name { get; set; }
    public int Age { get; set; }
}

// Controller method
public JsonResult GetUser() {
    User user = new User { Name = "Tom", Age = 28 };
    return Json(user);
}
```

# Переадресация из действия контроллера

В ASP.NET Core MVC для создания переадресации используются классы **RedirectResult, LocalRedirectResult, RedirectToActionResult** и **RedirectToRouteResult**.

Протокол HTTP поддерживает два типа переадресации:

- **постоянная переадресация (301):**

```
public IActionResult Index()
{
    return RedirectPermanent("~/Home/About");
}
```

- **временная переадресация (302):**

```
public IActionResult Index()
{
    return Redirect("~/Home/About");
}
```

Обратите внимание на знак тильды в начале пути "~". В ASP.NET данный знак обозначает корень приложения.

# О контексте в контроллерах

В контроллере можем получить информацию, связанную с контекстом выполнения запроса. Для получения контекста в контроллере используется свойство **ControllerContext**, представляющее одноименный класс **ControllerContext**. В объекте определен набор 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.

## Немного о HttpContext

Объект HttpContext инкапсулирует всю информацию о запросе, например:

- **Request**: содержит информацию о текущем запросе клиента;
- **Response**: управляет ответом сервера;
- **User**: представляет текущего пользователя, который обращается к приложению;
- **Session**: объект для работы с текущей сессией.

# О контексте - Request

Свойство HttpContext.Request возвращает объект HttpRequest с информацией о запросе. Этот же объект доступен через свойство Request класса Conroller.

Среди свойств объекта Request следует выделить следующие:

- **Body -** объект Stream для сырого чтения данных запроса;
- **Cookies -** куки, полученные в запросе;
- **Form -** коллекция значений отправленной формы (метод POST);
- **Headers -** коллекция заголовков запроса;
- **Path -** возвращает запрошенный путь (строка запроса без домена и порта);
- **Query -** возвращает коллекцию переданных через строку запроса параметров;
- **QueryString -** возвращает ту часть запроса, которая содержит параметры. Например, в запросе http://localhost:52682/Home/Index?alt=4 это будет ?alt=4

# О контексте - Response

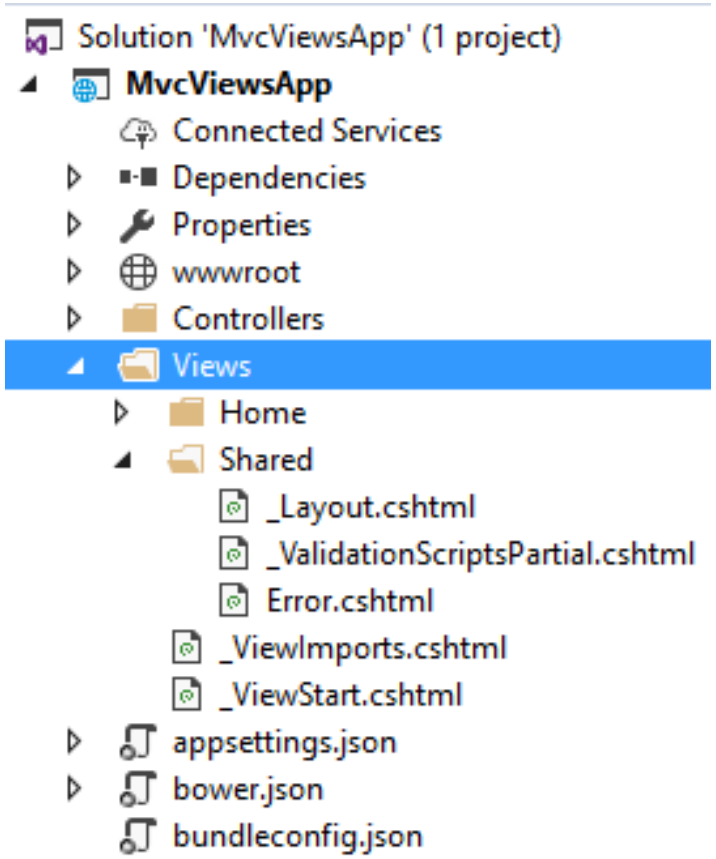
Свойство HttpContext.Response возвращает объект HttpResponse и позволяет управлять ответом на запрос: устанавливать заголовки ответа, куки, писать в выходной поток некоторые данные. Этот же объект доступен через свойство Response класса Conroller. Среди свойств объекта Response следует выделить следующие:

- **Body** - объект Stream, который применяется для отправки данных в ответ пользователю;
- **Cookies** - куки, отправляемые в ответе;
- **ContentType** - MIME-тип ответа;
- **Headers** - коллекция заголовков ответа;
- **StatusCode** - код ответа HTTP.

# Представление (View)

В большинстве случаев веб-приложение возвращает HTML-страницу с информацией. В MVC для этого, как правило, используются представления, которые и формируют результирующий HTML-код.

Представления – это файлы с расширением cshtml, которые содержат код пользовательского интерфейса в основном на языке HTML, а также конструкции **Razor** - специального движка представлений, который позволяет переходить от кода HTML к коду на языке C#.




**Правила и соглашения:**

- помещаются в папку **Views** (часто для каждого контроллера в проекте создается подпапка в папке Views, содержащая представления, используемые в анном контроллере). Например, для HomeController в папке Views есть подпапка Home с представлениями для методов контроллера HomeController
- есть папка Shared, которая хранит общие представления для всех контроллеров. По умолчанию это файлы:
  - *_Layout.cshtml* (используется в качестве мастер-страницы),
  - *Error.cshtml* (используются для отображения ошибок)
  - *_ValidationScripsPartial.cshtml* (частичное представление, которое подключает скрипты валидации формы).

# ViewResult – результат действия контроллера

Объект ViewResult производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту.

Чтобы возвратить объект ViewResult используется метод View:

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Контроллер производит поиск представления в проекте по следующим путям:

- */Views/Имя_контроллера/Имя_представления.cshtml*
- */Views/Shared/Имя_представления.cshtml*

Если название представления не указано явным образом, то в качестве представления будет использоваться то, имя которого совпадает с именем действия контроллера. Например, вышеопределенное действие Index по умолчанию будет производить поиск представления Index.cshtml в папке /Views/Home/.

# ViewResult – перегрузки метода View

Метод View() имеет четыре перегруженных версии:

- View(): для генерации ответа используется представление, которое по имени совпадает с вызывающим методом
- View(string viewName): в метод передается имя представления, что позволяет переопределить используемое по умолчанию представление

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View("About");
    }
}

ИЛИ

public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View("~/Views/Some/Index.cshtml");
    }
}
```

- View(object model): передает в представление данные в виде объекта model;
- View(string viewName, object model): переопределяет имя представления и передает в него данные в виде объекта model.

# Передача данных в представление

Существуют различные способы передачи данных из контроллера в представление:

- **ViewData** – представляет словарь из пар ключ-значение:

```
public IActionResult About() {
    ViewData["Message"] = "Hello ASP.NET Core";
    return View();
}
// И далее в представлении:
@ViewData["Message"]
```

- **ViewBag** – подобен ViewData, но содержит динамические свойства, которым можно присвоить значение любого типа (в т.ч. сложные):

```
ViewBag.Message = "Hello ASP.NET Core"; // и далее
```

- **TempData** – словарь из пар ключ-значение для текущего и последующего запроса.

- **Модель представления**.

# Модель (модель представления)

Модель представления является более предпочтительным способом для передачи данных в представление. Для передачи данных в представление используется одна из перегрузок метода View:

```
public IActionResult About() {
    List<string> countries = new List<string> { "Бразилия",
"Аргентина", "Уругвай", "Чили" };
    return View(countries);
}
```

```
@model List<string>
@{
    ViewBag.Title = "About";
}

<h3>В списке @Model.Count элемента</h3>
@foreach(string country in Model)
{
    <p>@country</p>
}
```

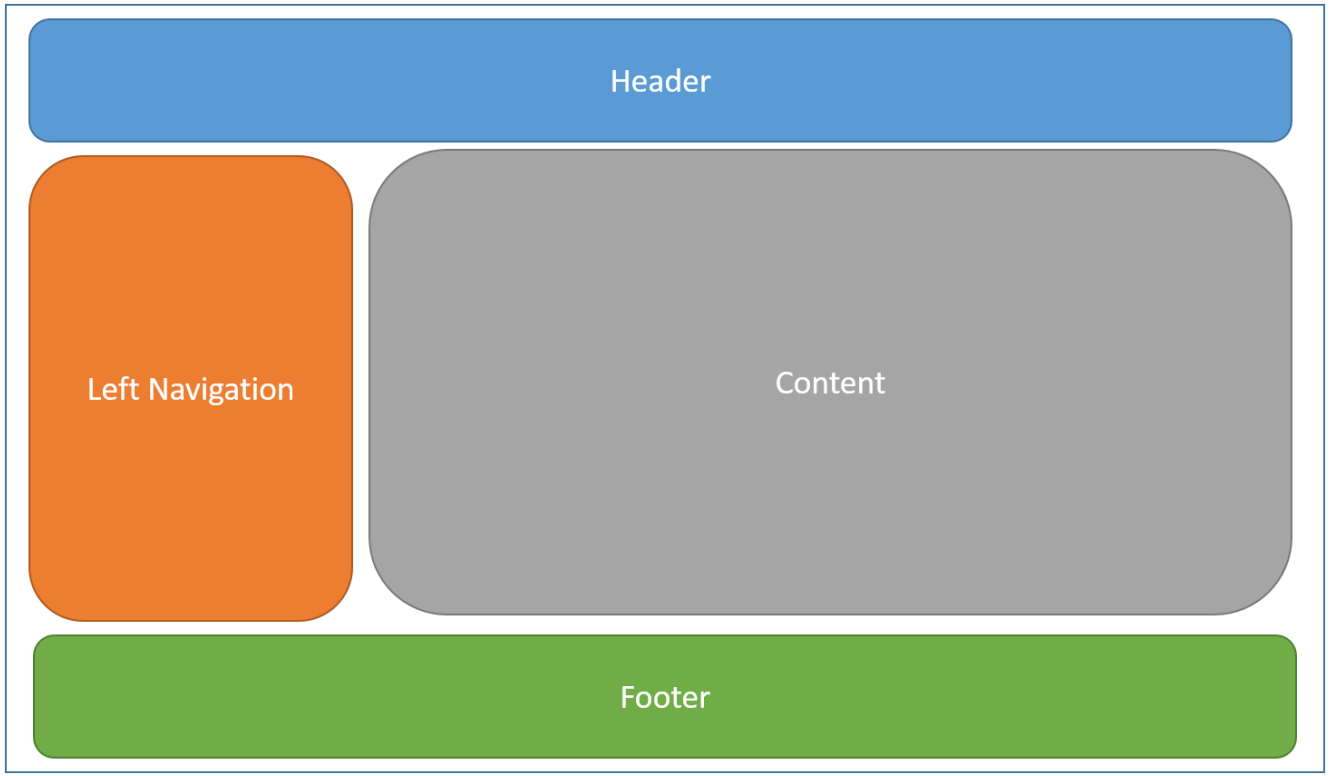
Далее в представлении

Представления, для которых определена модель, называют **строго типизированными.**

Т.к. в метод View передается список, то моделью представления является тип List<string> (либо IEnumerable<string>).

# Мастер-страницы

Мастер-страницы – это представления, которые включают в себя другие представления (специальные теги позволяют вставлять в определенное место на мастер-страницах другие представления). Мастер-страницы применяются для создания унифицированного вида приложения, т.е. позволяют задать общий шаблон.




По умолчанию в проекте есть мастер-страница _Layout.chtml в папке Views/Shared.
Главное отличие от обычных представлений - использование метода **@RenderBody()**, на место которого подставляется код сформированный другими представлениями, использующими данную мастер-страницу.

# Представление ViewStart

По умолчанию представления уже подключают мастер-страницу за счет файла _ViewStart.cshtml через свойство *Layout* (расширение можно не использовать):

```
@{
    Layout = "_Layout";
}
```

Поиск мастер-страницы _Layout осуществляется по следующим путям:

- /Views/[Название_контроллера]/_Layout.cshtml
- /Views/Shared/_Layout.cshtml

Для отключения мастер-страницы достаточно присвоить значение null:

```
@{
    Layout = null;
}
```

# Секции

Мастер-страница может использовать специальный метод **RenderSection()** для вставки секций. Мастер-страница может иметь несколько секций, куда представления помещают свое содержимое. Например, можно добавить к мастер-странице секцию footer (или @RenderSection("**Footer**", required: false)):

```
<footer> @RenderSection("Footer") </footer>
```

Представление должно передавать содержимое для каждой обязательной секции мастер-страницы. Например, в представление Index необходимо добавить секцию footer с помощью выражения @section:

```
@section Footer {
    All rights reserved. 2021.
}
```

# Частичные представления (partial views)

Частичные представления – это представления, которые можно встраивать в другие представления (например, фрагмент HTML для AJAX). Частичные представления похожи на секции, но их код выносится в отдельные файлы.

**Использование частичных представлений**

- из контроллера:

```
public class HomeController : Controller{
    public ActionResult GetMessage(){
        return PartialView("_GetMessage");
    }
}
```

- из представления:

```
@Html.Partial("_GetMessage")
```

**или**

```
@Html.Partial("_GetMessage", new List<string> { "case1", "case2" })
```