

ASP MVC

v4

Знакомство с MVC

В чем основная идея?

ASP.NET MVC является фреймворком для разработки от Microsoft, который сочетает в себе эффективность и аккуратность архитектуры MVC, самые современные идеи и методы гибкой разработки и лучшие свойства существующей платформы ASP.NET

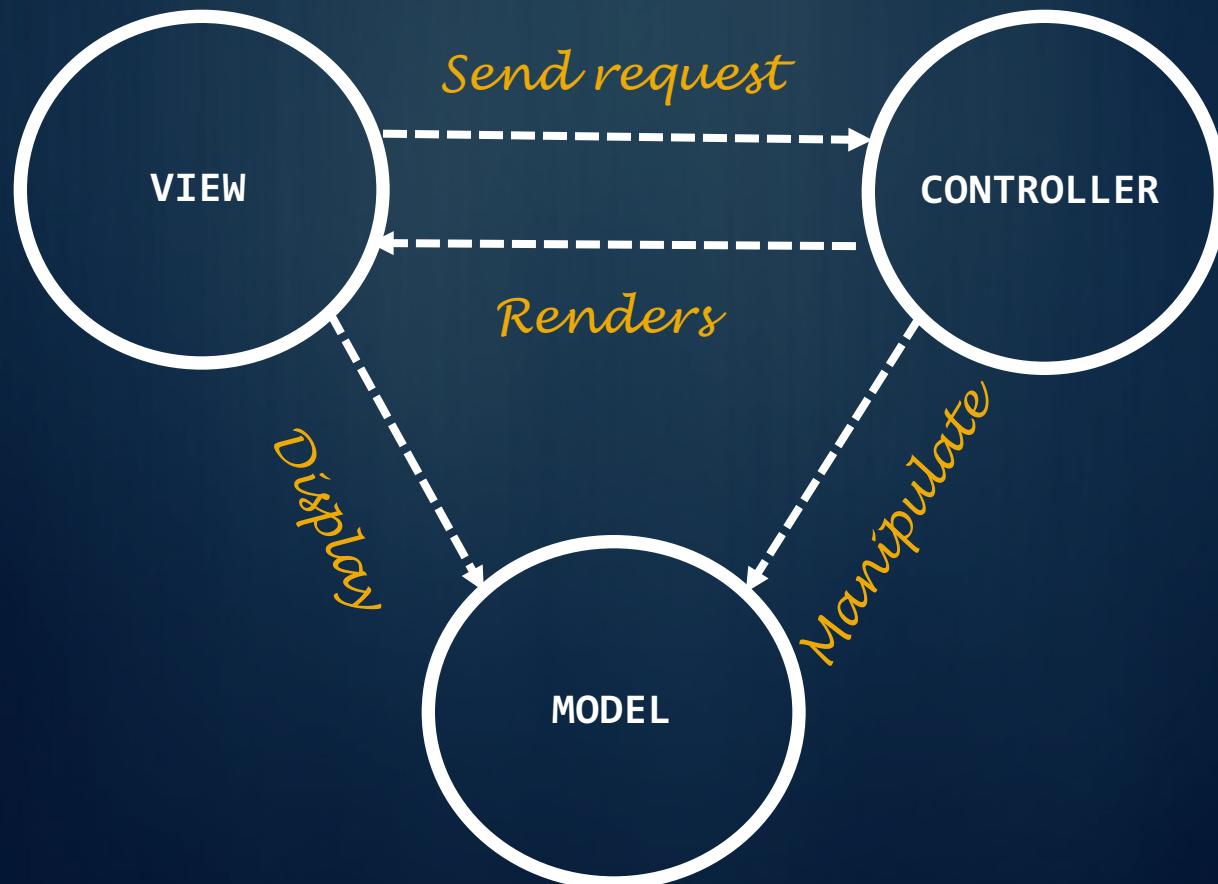
Model – View – Controller

Контроллер (Controller) – контроллеры обрабатывают входящие запросы, выполняют операции для модели и выбирают представления для показа пользователю

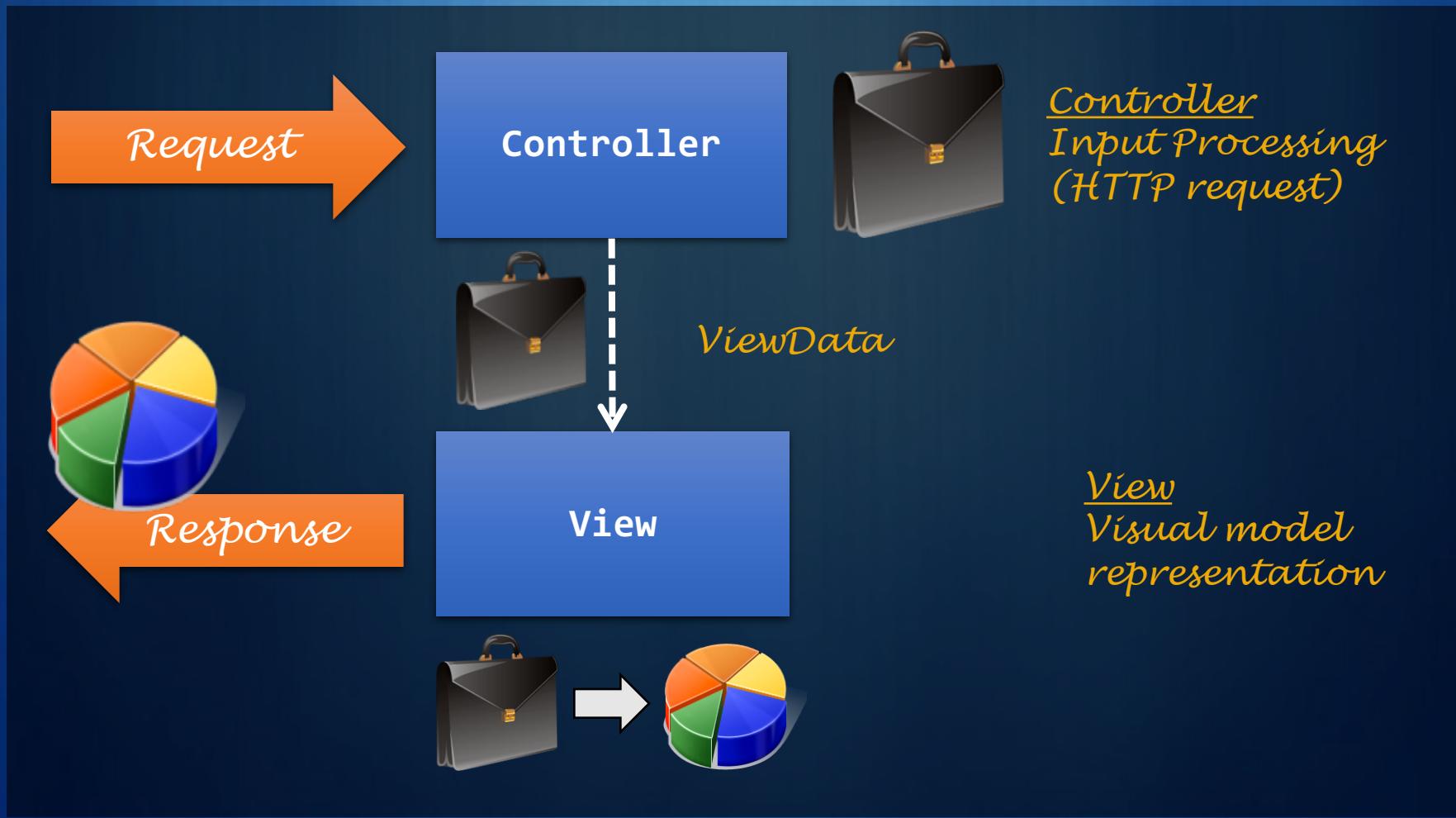
Модель (Model) – модели содержат или представляют данные, с которыми работают пользователи. Это могут быть простые *модели представления*, которые только представляют данные, передаваемые от контроллера представлению, или они могут быть *доменными моделями*, которые содержат данные домена, а также операции, преобразования и правила работы с этими данными.

Представление (View) – представления используются для того, чтобы обработать некоторые части модели в качестве пользовательского интерфейса

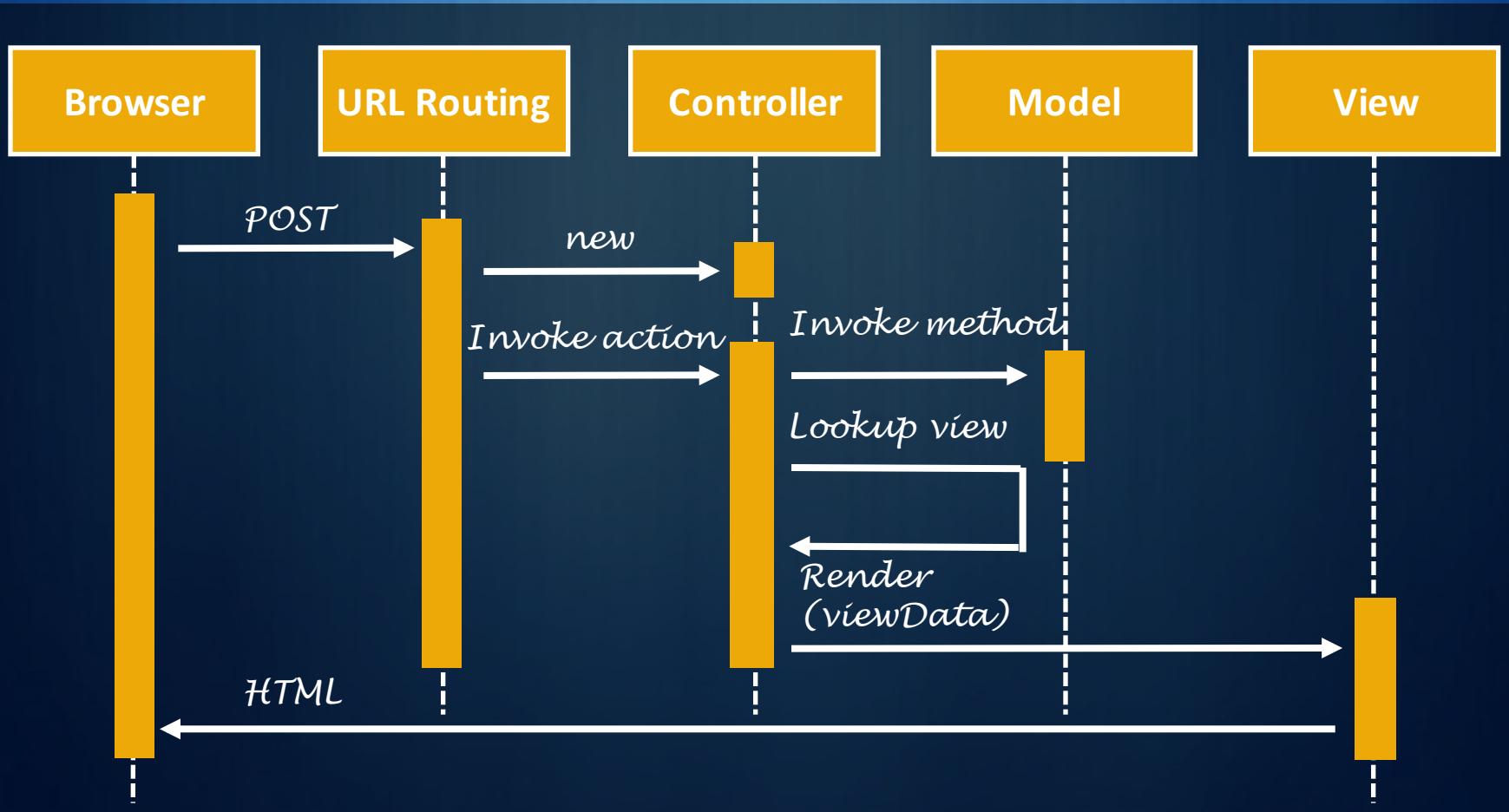
Model – View – Controller



ASP.NET реализация паттерна MVC



Цикл запроса в MVC



Структура MVC 4 приложения

Папки и файлы

/App_Data

Папка для хранения приватных данных, таких как XML, файлы или баз-данных, если используется SQL Server Express, SQLite

IIS не будет сохранять содержимое этой папки

/bin

Здесь находятся скомпилированные сборки MVC приложения, которые не входят в GAC

IIS не будет сохранять содержимое этой папки

/Content

Папка для хранения статического контента, такого как CSS файлы и картинки

Это соглашение, но не обязательное

/Controllers

Папка для контроллеров

Это соглашение

Папки и файлы

/Models

Папка для view-моделей и классов
доменной модели

Это соглашение, однако
лучше помещать
доменную модель в
отдельный проект

/Scripts

Папка для JavaScript библиотек и
собственных скриптов MVC
приложения

Это соглашение

/Views

Папка для представлений и
частичных представлений, обычно
сгруппированы вместе в папки,
названия которых совпадают с
именами соответствующих им
контроллеров

Папки и файлы

/Views/Shared

/Views/Shared – папка для layout-файлов и представлений, которые не специфичны для одного контроллера

/Views/Web.config

/Views/Web.config содержит настройки, необходимые для того, чтобы представления работали с ASP.NET

/Global.asax

Глобальный класс ASP.NET приложения

/Web.config

Конфигурационный файл для вашего приложения

Соглашения именования

По умолчанию приложения ASP.NET MVC основываются на некоторых соглашениях, позволяя разработчикам избежать излишней конфигурации:

- Название класса контроллера должно заканчиваться на **Controller**. Например, `HomeController`, `AccountController`.
- Представления располагаются в папке **Views/[Название контроллера]**.
- Название представления по умолчанию совпадает с названием **action**-метода.
- Когда MVC Framework выполняет поиск необходимого представления, он сначала ищет его в папке, соответствующей контроллеру, а потом в папке **Shared**.
- Layout-файлы должны начинаться с нижнего подчеркивания и находиться в папке **Shared**.

Создание первого приложения

!

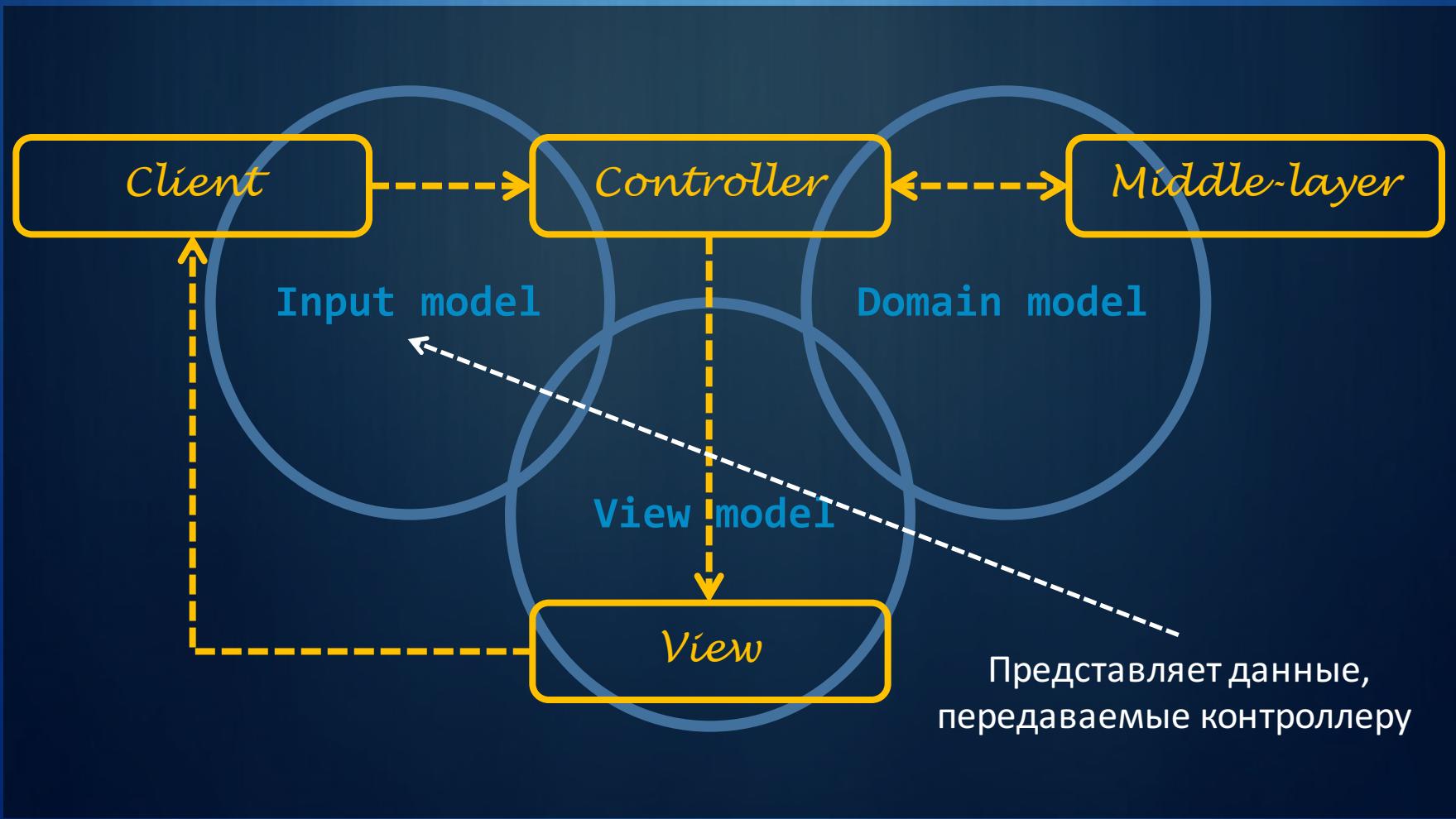
Example

2015 ©EPAM Systems

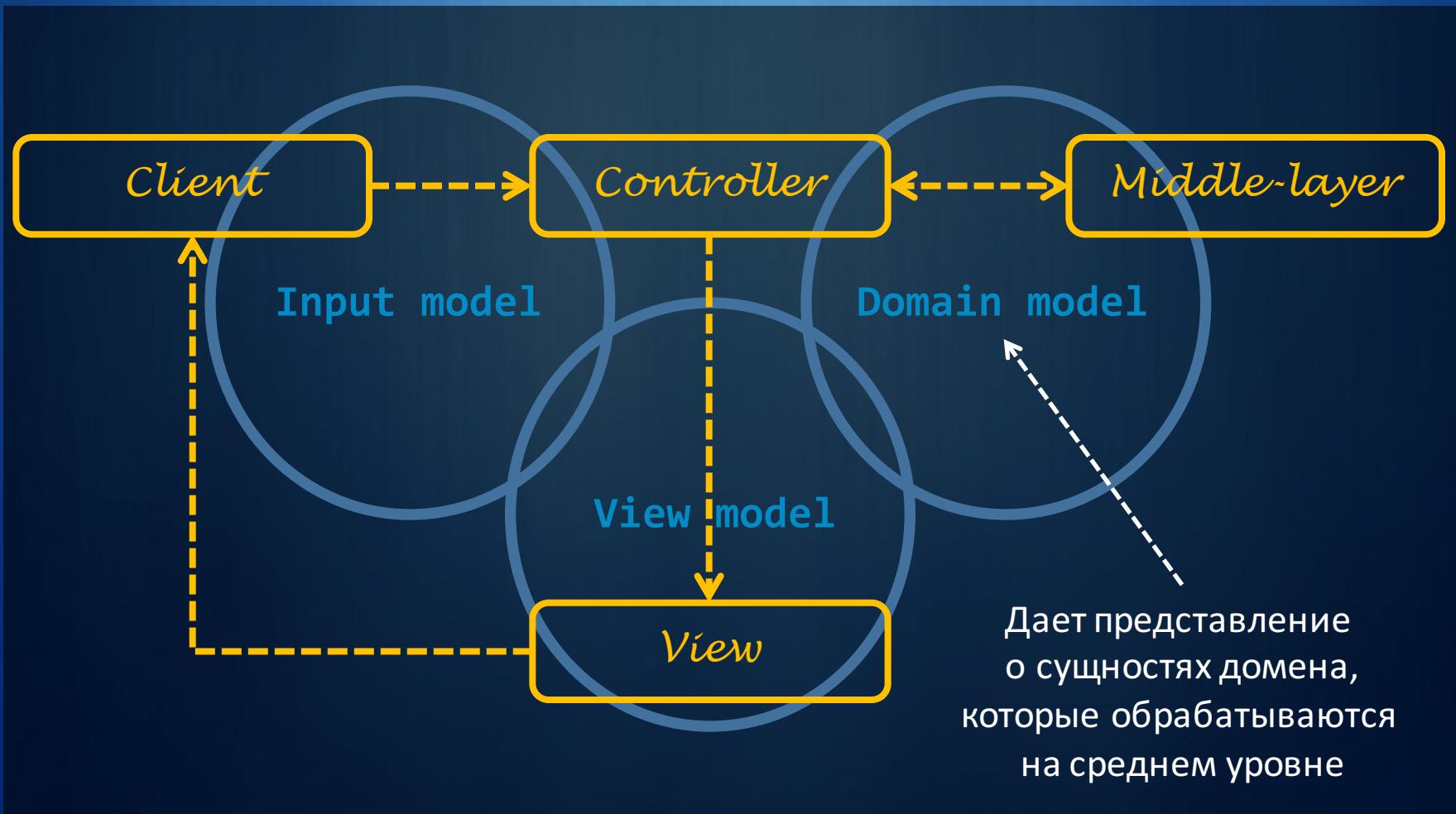
13

Понятие модели

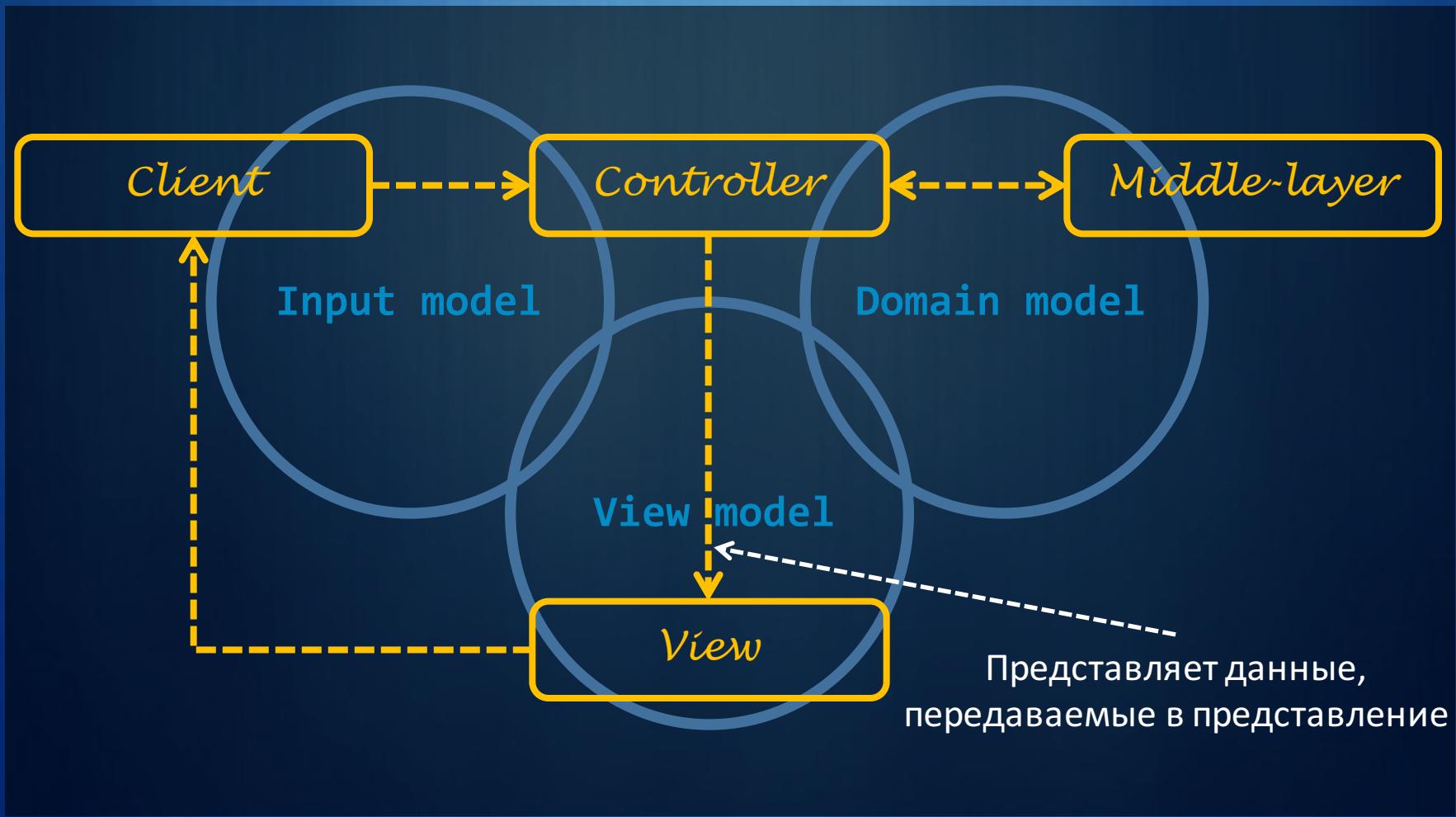
Типы моделей, доступные в приложениях ASP.NET MVC



Типы моделей, доступные в приложениях ASP.NET MVC

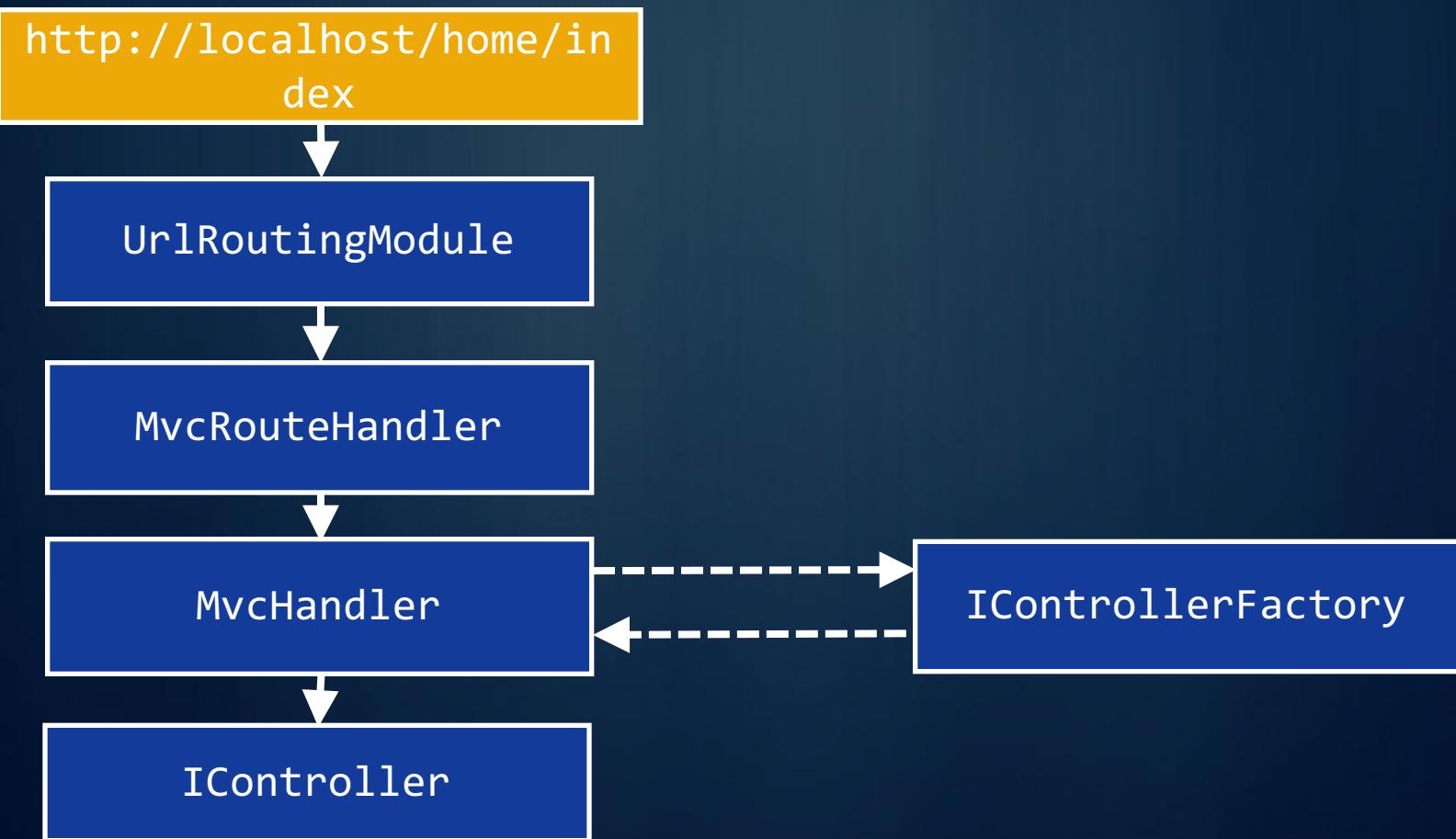


Типы моделей, доступные в приложениях ASP.NET MVC

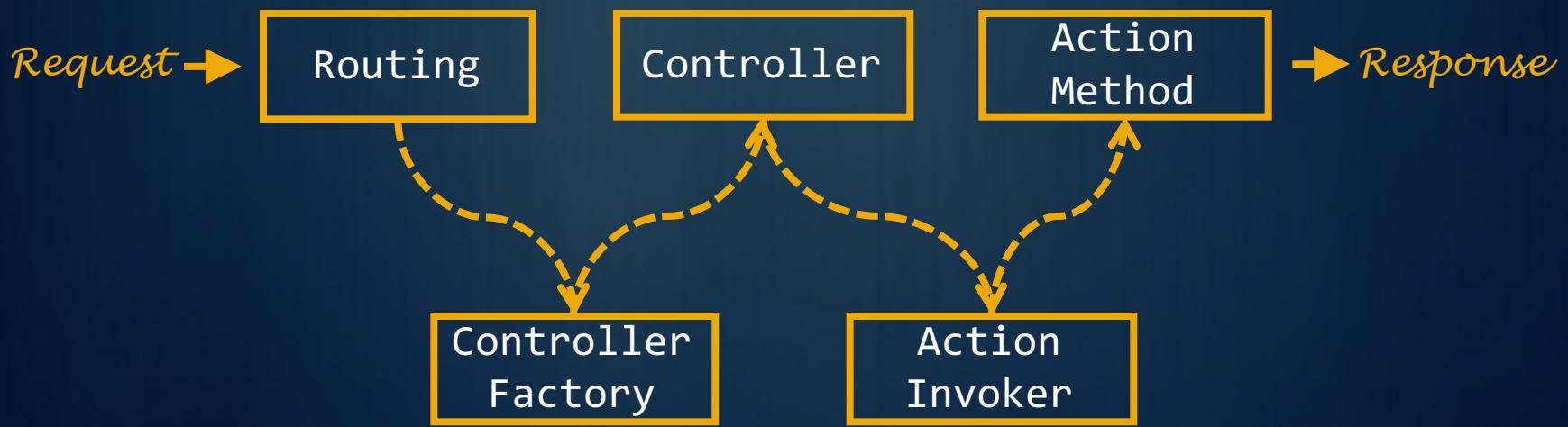


Контроллеры

Компоненты конвейера обработки запроса



Компоненты конвейера обработки запроса



Основы контроллеров

Для большинства приложений, встроенной фабрики контроллеров, называемой `DefaultControllerFactory`, вполне достаточно. Когда фабрика контроллеров получает запрос от системы маршрутизации, она смотрит на данные маршрутизации, чтобы определить соответствующий контроллер, и пытается найти класс в веб-приложении, удовлетворяющий следующим критериям:

- Класс должен быть **открытым**
- Класс не должен быть **абстрактным**
- Класс не должен принимать обобщенные параметры
- Имя класса должно завершаться словом `Controller`
- Класс должен реализовывать интерфейс `IController`

Создание контроллера на основе интерфейса IController

В MVC класс контроллера должен реализовывать интерфейс **IController** пространства имен **System.Web.Mvc**

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Когда класс реализует данный интерфейс, то MVC распознает этот класс как контроллер и сможет с его помощью обработать запрос.

Довольно непросто написать сложный контроллер, самостоятельно реализуя этот интерфейс. Лучше, создавая свои контроллеры, наследуя их от класса **System.Web.Mvc.Controller**



Example

Создание контроллера на основе интерфейса IController

```
public class BasicController : IController
{
    public void Execute(RequestContext requestContext)
    {
        string controller =
            (string)requestContext.RouteData.Values["controller"];
        string action =
            (string)requestContext.RouteData.Values["action"];
        requestContext.HttpContext.Response.
            Write(string.Format("Controller: {0}, Action: {1}",
                controller, action));
    }
}
```



Example

Создание контроллера на основе класса Controller



Создание контроллера на основе класса Controller

С классом **Controller** связаны следующие ключевые возможности:

- **Action methods** (методы действия): поведение контроллера разделено на множество методов (вместо того, чтобы иметь только один метод Execute). Каждый метод действия срабатывает для определенного, «своего» URL и вызывается с параметрами, извлеченными из входящего запроса.
- **Action results** (результаты действия): можно вернуть объект, описывая результат действия (например, отображение представления или перенаправление на другой URL или метод действия), который затем можно использовать по своему усмотрению. Разделение между указанием результатов и их выполнением упрощает модульное тестирование.
- **Filters** (фильтры): можно инкапсулировать повторяющиеся виды поведения (например, аутентификацию) в качестве фильтров, а затем добавлять каждый вид поведения в один или несколько контроллеров или методов действия, разместив **[Attribute]** в исходном коде.

Основы контроллеров

Guestbook Controller class

```
public class GuestbookController : Controller  
{
```

```
    public ActionResult Index()  
    {
```

```
        return View();  
    }
```

```
}
```

Return type

Base Controller class

Action method

*View() defined in
base Controller class*

Требования для action-метода

Чтобы рассматриваться в качестве действия, **action-метод** должен удовлетворять следующим требованиям:

- Не может быть статическим
- Не может быть методом расширения
- Не может быть конструктором или свойством
- Не может иметь открытый параметризованный тип
- Не может быть методом базового класса **Controller**
- Не может быть методом базового класса **ControllerBase**
- Не может содержать параметры **ref** или **out**
- Не может быть помечен атрибутом **NonAction**

Получение данных из набора контекстных объектов

Методам контроллеров, как правило, необходим доступ к входящим данным, таким как значения строки запроса, значения форм или параметры, полученные системой маршрутизации из входящего URL. Есть три основных способа доступа к этим данным:

- Получить данные из набора контекстных объектов
- Получить данные, переданные в качестве параметров методу действия
- Явно вызвать связывание данных модели

Получение данных из набора контекстных объектов

При создании контроллера путем наследования от базового класса **Controller**, автоматически предоставляется доступ к набору полезных свойств для получения информации о запросе

- Request
- Response
- RouteData
- HttpContext
- Server

Каждое из свойств предоставляет информацию о различных аспектах запроса. Эти свойства получают различные типы данных из экземпляра запроса **ControllerContext** (который может быть доступным через свойство **Controller.ControllerContext**)



Example

Часто используемые контекстные объекты

Свойство	Тип	Описание
Request.QueryString	NameValueCollection	Переменные GET, отправленные с этим запросом
Request.Form	NameValueCollection	Переменные POST, отправленные с этим запросом
Request.Cookies	HttpCookieCollection	Куки, отправленные браузером с этим запросом
Request.HttpMethod	string	HTTP метод (например, GET или POST), используемый для этого запроса
Request.Headers	NameValueCollection	Полный набор HTTP заголовков, отправленный с этим запросом
Request.Url	Uri	Запрашиваемый URL
Request.UserHostAddress	string	IP адрес пользователя, сделавшего запрос

Часто используемые контекстные объекты

Свойство	Тип	Описание
RouteData.Route	RouteBase	Выбранная запись из RouteTable.Routes для этого запроса
RouteData.Values	RouteValueDictionary	Активные роутовые параметры (как полученные из URL, так и значения по умолчанию)
HttpContext.Application	HttpApplicationStateBase	Состояние приложения
HttpContext.Cache	Cache	Кэш приложения
HttpContext.Items	IDictionary	Состояние текущего запроса
HttpContext.Session	HttpSessionStateBase	Состояние сессии пользователя
User	IPrincipal	Информация об аутентификации залогиненного пользователя

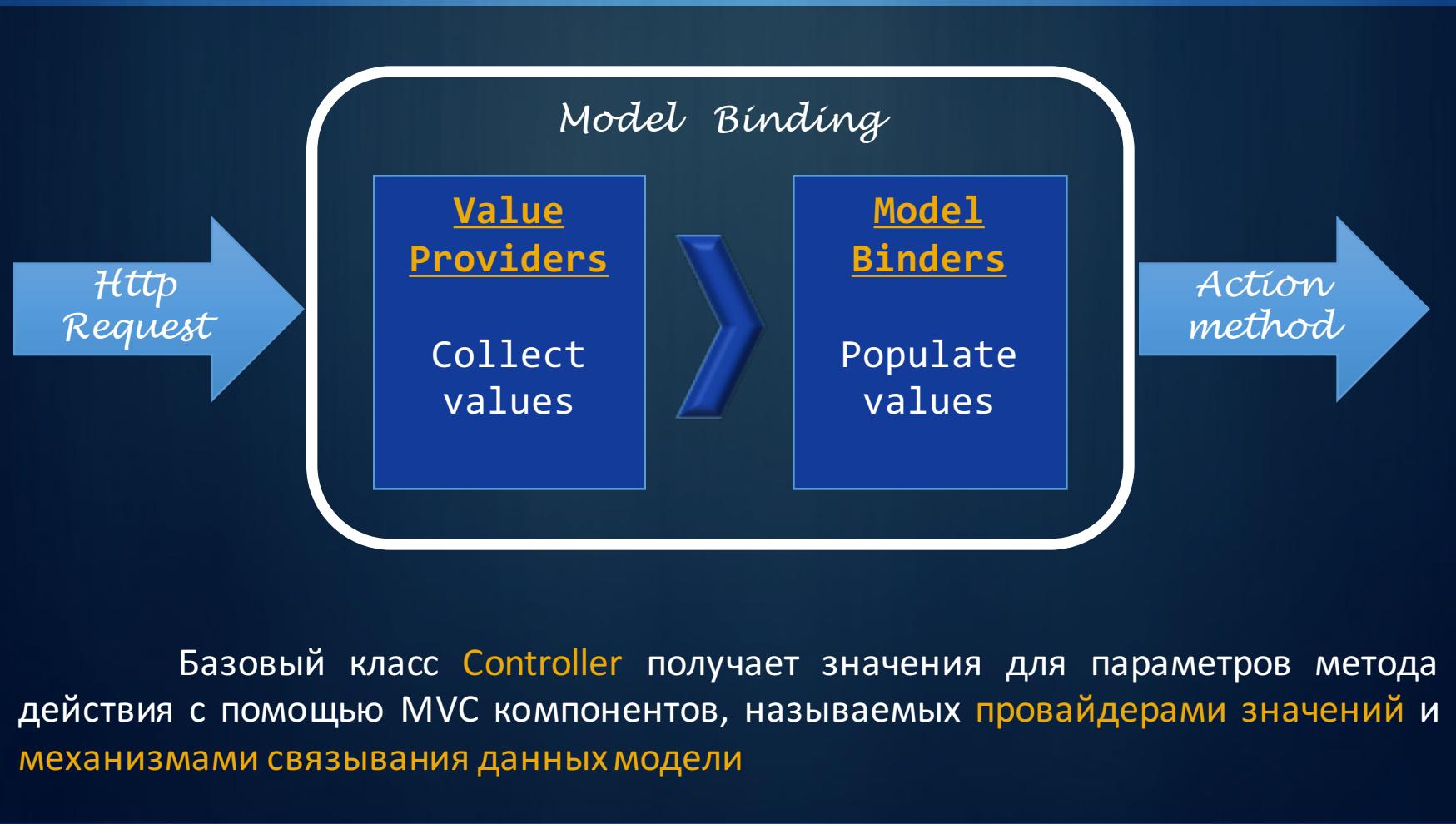
Часто используемые контекстные объекты

```
public ActionResult RenameProduct()
{
    string userName = User.Identity.Name;
    string serverName = Server.MachineName;
    string clientIP = Request.UserHostAddress;
    DateTime dateStamp = HttpContext.Timestamp;
    AuditRequest(userName, serverName, clientIP, dateStamp, "Renaming
product");
    // Получение данных из Request.Form
    string oldProductName = Request.Form["OldName"];
    string newProductName = Request.Form["NewName"];
    bool result = AttemptProductRename(oldProductName, newProductName);
    ViewData["RenameResult"] = result;
    return View("ProductRenamed");
}
```



Example

Связывание данных модели и провайдеры значений



Базовый класс **Controller** получает значения для параметров метода действия с помощью MVC компонентов, называемых **проводерами значений** и **механизмами связывания данных модели**

Связывание данных модели и провайдеры значений

Базовый класс `Controller` получает значения для параметров метода действия с помощью MVC компонентов, называемых **проводерами значений** и **механизмами связывания данных модели**

Встроенные провайдеры значений получают элементы из `Request.Form`, `Request.QueryString`, `Request.Files` и `RouteData.Values` и передают эти значения механизмам связывания данных, которые пытаются привязать их к типам, которые методы действий требуют в качестве параметров

Связывание данных модели и провайдеры значений

Поставщик данных – это класс, который выполняет поиск значения в определенной части входящего запроса

- `FormValueProvider` – значения, переданные в HTML элементах form
- `RouteDataValueProvider` – значения из маршрутов приложения
- `QueryStringValueProvider` – данные включенные в строку запроса
- `HttpFileCollectionValueProvider` – файлы загруженные как часть запроса

```
public interface IModelBinderProvider
{
    IModelBinder GetBinder(Type modelType);
}
```

Связывание данных модели и провайдеры значений

Action invoker (активатор действия) - компонент, который вызывает action-методы. Перед тем, как вызвать action-метод, необходимо заполнить значениями параметры метода. ControllerActionInvoker (активатор действия по умолчанию) для заполнения параметров использует привязчики модели (model binder), которые осуществляют привязку модели. Все привязчики модели реализуют интерфейс **IModelBinder**.

```
public interface IMModelBinder
{
    object BindModel (ControllerContext controllerContext,
                      ModelBindingContext bindingContext);
}
```

Связывание данных модели и провайдеры значений

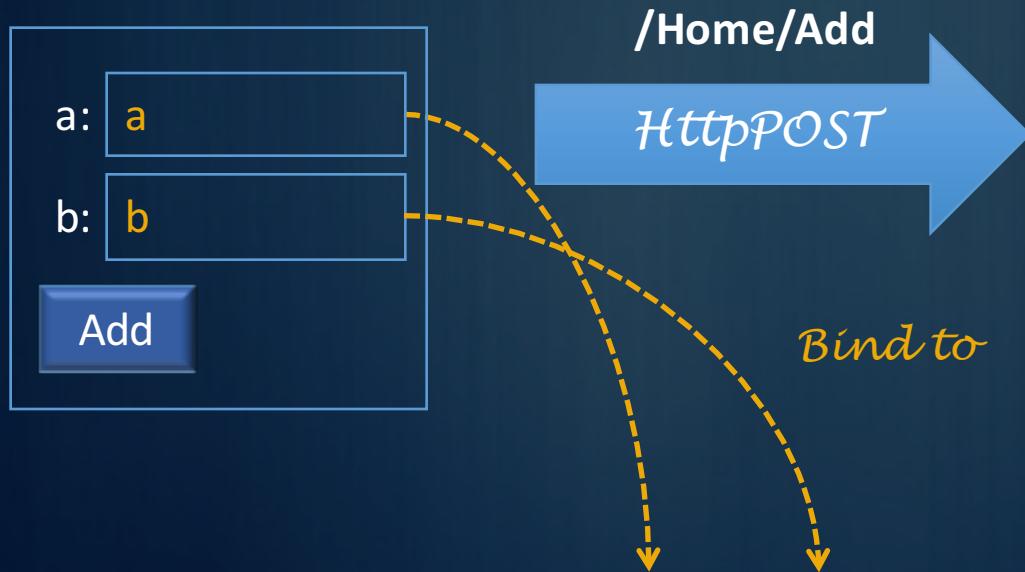
/Home/Add?a=1&b=12
/Home/Add/1/12

HttpGET

Bind to

```
public ActionResult Add(int a, int b)
{
    return Content(string.Format("{0} + {1} = {2}", a, b, a + b));
}
```

Связывание данных модели и провайдеры значений



```
public ActionResult Add(int a, int b)
{
    return Content(string.Format("{0} + {1} = {2}", a, b, a + b));
}
```

Связывание данных модели и провайдеры значений

Для каждого отдельного типа может существовать свой привязчик модели. При просмотре параметров метода действия активатор действий ищет для каждого типа параметра соответствующий привязчик и вызывает его метод `BindModel`. В случае, если соответствующего данному типу привязчика не обнаружится, то используется привязчик по умолчанию - `DefaultModelBinder`.

Привязчик использует специальные компоненты - **поставщики значений (value provider)** для поиска значений в различных частях запроса.

Связывание данных модели и провайдеры значений

Привязчик `DefaultModelBinder` используется по умолчанию, если для данного типа не определен другой привязчик. Чтобы получить значения для параметров, привязчик просматривает следующие объекты строго по порядку:

- `Request.Form`. Значения, предоставленные пользователем в элементе HTML form
- `RouteData.Values`. Значения, полученные через маршруты приложения
- `Request.QueryString`. Данные строки запроса из URL
- `Request.Files`. Файлы, загруженные как часть запроса

Если данные не найдены, параметры ссылочного типа получают значение `null`, а для параметров типов значений генерируется исключение `InvalidOperationException`.

При конвертировании данных из словаря `Request.Form` применяются настройки языковой культуры сервера

Связывание данных модели и провайдеры значений

Если привязчик `DefaultModelBinder` не может найти значение параметра ссылочного типа, все равно будет вызван метод действия, но с использованием значения `null` для этого параметра. Если значение не может быть найдено для параметра значимого типа, то будет сгенерировано исключение, и метод действия вызван не будет:

- Параметры значимого типа являются обязательными. Чтобы сделать их опциональными, нужно либо указать значение по умолчанию, либо заменить тип параметра на `nullable` (например, `int?` или `DateTime?`). Таким образом, MVC сможет передать `null`, если значение будет не доступно.
- Параметры ссылочного типа не являются обязательными. Чтобы сделать их обязательными (чтобы убедиться, что передается значение не-`null`), нужно добавить код в начало метода действия, который не принимает значения `null`. Например, если значение равно `null`, выбрасывается исключение `ArgumentNullException`

Связывание данных модели и провайдеры значений

Когда параметр action-метода является сложным типом, тогда связыватель использует отражение (reflection), чтобы получить набор public свойств и затем по очереди привязать к каждому из них значение.

Чтобы помочь связывателю, можно использовать html helper-методы, например `Html.TextboxFor(m => m.Title)`, или же у html элементов задавать атрибут `name`.

Для настройки привязки к модели можно использовать атрибут `[Bind]`, применяемый либо к параметру, либо к классу модели. При помощи `[Bind]` можно указать свойства объекта для применения или игнорирования привязки. Привязка к модели работает не только в случае скалярных объектов, но и для коллекций. Данные формы могут быть приняты действием с параметром `List<>`, `Dictionary<>`.

Связывание данных модели и провайдеры значений

/Guestbook/Create?name=Guest&message>Hello

HttpGET

```
public ActionResult Create()
{
    var name = Request.QueryString["name"];
    var message = Request.QueryString["message"];
    //update database here
    return RedirectToAction("Index");
}
```

Связывание данных модели и провайдеры значений

/Guestbook/Create

HttpPost

Enter your name

Enter your message

```
public ActionResult Create()
{
    var name = Request.Form["name"];
    var message = Request.Form["message"];
    //update database here
    return RedirectToAction("Index");
}
```

Связывание данных модели и провайдеры значений

/Guestbook/Create

HttpPost

Enter your name

name

Enter your message

message

Add

```
[HttpPost]
public ActionResult Create(
    FormCollection values)
{
    var name = values["Name"];
    var message = values["Message"];
    //update database here
    return RedirectToAction("Index");
}
```

Связывание данных модели и провайдеры значений

/Guestbook/Create

HttpPost

The diagram illustrates the binding process between a user interface form and a model class. A blue arrow points from the URL '/Guestbook/Create' and the 'HttpPost' annotation to a code block. Dashed orange arrows map the 'name' field in the form to the 'Name' property in the model, and the 'message' field to the 'Message' property. The 'Add' button is also shown.

Enter your name
name

Enter your message
message

Add

```
public class GuestbookEntry
{
    public string Name { get; set; }
    public string Message { get; set; }
    .
    .
}

[HttpPost]
public ActionResult Create(GuestbookEntry entry)
{
    //update database here
    return RedirectToAction("Index");
}
```

Селекторы метода действия

Селекторы метода действия помогают механизму маршрутизации выбрать для конкретного запроса правильный метод действия. Селекторы метода действия могут быть применены к методам действий. MVC включает следующие атрибуты:

- ActionName
- NonAction
- ActionVerbs

Селекторы метода действия

```
public class GuestbookController : Controller  
{  
    [ActionName("Find")] ← Action Selector ActionName  
    public ActionResult GetById(int id)  
    {  
        // get guestbook entry from the database  
        return View();  
    }  
}
```

Селекторы метода действия

```
public class GuestbookController : Controller
{
    [NonAction] <-- Action Selector NonAction
    public GuestbookEntry GetEntry(int id)
    {
        return entries
            .Where(s => s.EntryId == id).FirstOrDefault();
    }
}
```

Селекторы метода действия

http://localhost/Guestbook/Create

HttpGET

```
[HttpGet]  
public ActionResult Create()  
{  
    return View();  
}
```

http://localhost/Guestbook/Create

HttpPOST

```
[HttpPost]  
public ActionResult Create(  
    GuestbookEntry entry)  
{  
    //update database here...  
    return RedirectToAction("Index");  
}
```

Селекторы метода действия

Http method	Использование
GET	Для получения информации с сервера. Параметры будут добавлены в строку запроса.
POST	Для создания нового ресурса
PUT	Для обновления существующего ресурса
HEAD	Идентичен GET, но не возвращает тело запроса
OPTIONS	Представляет запрос для информации об опциях соединения, поддерживаемых веб-сервером
DELETE	Для удаления существующего ресурса
PATCH	Для полного или частичного обновления ресурса

Создание выходных данных

При создании контроллера напрямую путем реализации интерфейса `IController`, приходится реализовывать вручную все аспекты обработки запроса, в том числе за создание ответа клиенту, используя методы

- `Response.Write`
- `Response.Redirect`

Возникающие проблемы:

- классы контроллеров должны содержать информацию об HTML или URL структуре, и поэтому эти классы труднее читать и поддерживать.
- трудно провести модульное тестирование контроллера, который генерирует свой ответ непосредственно в выходные данные.
- работа с мелкими деталями каждого ответа подобным образом является утомительной и может привести ко многим ошибкам.

Создание выходных данных

Результаты действий в MVC Framework используются для разделения заявлений о намерениях и выполнения намерений

При использовании контроллеров, унаследованных от `System.Web.Mvc.Controller`, action-методы возвращают объект для описания результата своей работы. Как правило, используется класс `ActionResult` и его наследники. Метод-действие может возвращать произвольный объект или быть объявленным как `void`. В первом случае на основе результата создается объект класса `ContentResult`, во втором – возвращается объект `EmptyResult`.

Система action-результатов реализует шаблон проектирования [команда](#). Когда MVC Framework получает объект типа `ActionResult` из action-метода, он вызывает у этого объекта метод `ExecuteResult`. Реализация `ActionResult` взаимодействует с объектом `Response` и генерирует необходимые выходные данные

Создание выходных данных

```
public abstract class ActionResult
{
    protected ActionResult();
    public abstract void ExecuteResult(ControllerContext
        context);
}
```

Создание выходных данных

Тип	Описание	Вспомогательные методы
ViewResult	Отображает указанный шаблон представления или шаблон по умолчанию	View
PartialViewResult	Отображает указанный шаблон частичного представления или шаблон по умолчанию	PartialView
RedirectToRouteResult	Работает с HTTP перенаправлением 301 или 302 на метод действия или конкретный роут, генерируя URL в соответствии с вашей конфигурацией	RedirectToAction RedirectToActionPermanent RedirectToRoute RedirectToRoutePermanent

Создание выходных данных

Тип	Описание	Вспомогательные методы
RedirectResult	Работает с HTTP перенаправлением 301 или 302 на конкретный URL	Redirect RedirectPermanent
HttpUnauthorizedResult	Устанавливает ответный код HTTP статуса на 401 (что означает "не авторизирован"), который вызывает активный механизм аутентификации (form-аутентификацию или Windows-аутентификацию), чтобы попросить посетителя войти в систему	Нет

Создание выходных данных

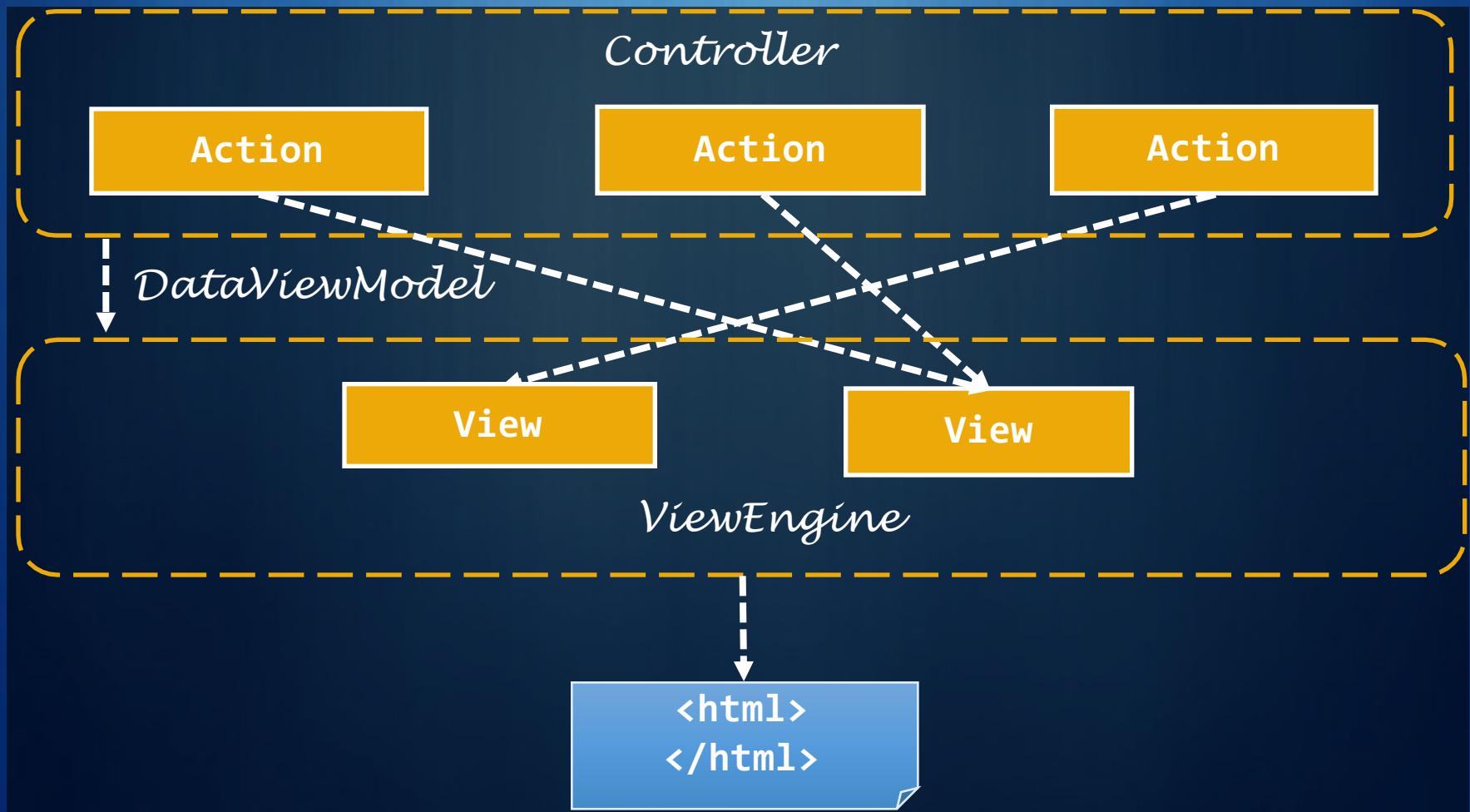
Тип	Описание	Вспомогательные методы
HttpNotFoundResult	Возвращает HTTP ошибку 404— Not found	HttpNotFound
HttpStatusCodeResult	Возвращает указанный HTTP код	Нет
ContentResult	Возвращает необработанные текстовые данные в браузер, возможно установление заголовка типа содержимого	Content
FileResult	Является базовым классом для всех объектов, пишущих бинарный ответ во выходной поток. Предназначен для отправки файлов	File

Создание выходных данных

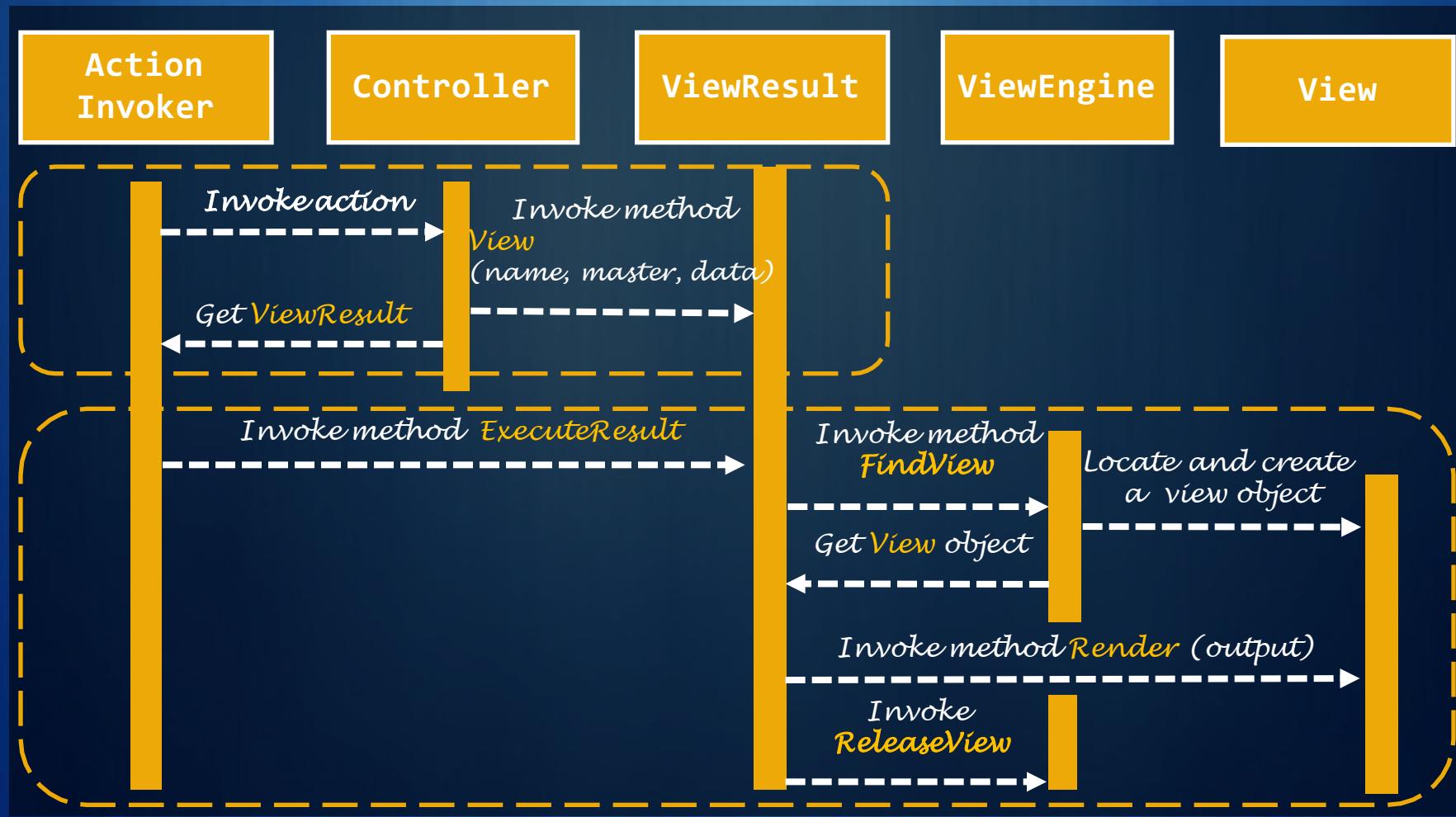
Тип	Описание	Вспомогательные методы
EmptyResult	Ничего не делает, отправляет пустой ответ	None
JsonResult	Возвращает в качестве ответа объект или набор объектов в формате JSON	Json
JavaScriptResult	Возвращает в ответ в качестве содержимого код JavaScript	JavaScript

Представление

Введение в представление



Компоненты конвейера обработки запроса



Введение в представление

Работа приложения MVC управляется главным образом контроллерами, но непосредственно пользователю приложение доступно в виде представления (**View**), которое и формирует внешний вид приложения.

Целью представления является прием переданной в него модели и использование данной модели для отображения содержимого приложения. В связи с тем, что контроллер и соответствующие службы уже исполнили всю бизнес-логику приложения и упаковали результаты в объект модели представления, представлению остается всего лишь узнать, как принять эту модель и преобразовать ее в HTML.

В ASP.NET MVC представления представляют файлы с расширением **cshtml/vbhtml/aspx/ascx**, которые содержат код с интерфейсом пользователя, как правило, на языке **html**.

Введение в представление

Компонент	Выполняет	Не выполняет
Action Method	Передает объект модели представления в представление	Не передает отформатированные данные в представление
View	Использует объект модели представления для показа контента пользователю	Не меняет ни один из аспектов объекта модели представления

Передача данных в представление

- TempData : TempDataDictionary
- ViewBag : Object
- ViewData : ViewDataDictionary
- Строго типизированные представления

Передача данных в представление



- **ViewBag** служит для передачи данных от контроллера в представление – как правило это временные данные, которые не включены в модель
- **ViewBag** это динамическое свойство, которое использует в себе новые динамические функции C# 4.0
- Для **ViewBag** можно назначить любое количество свойств и значений
- Цикл жизни **ViewBag** определяется только текущим HTTP-запросом, значения **ViewData** будут нулевыми, если происходит редирект
- **ViewBag** на самом деле является оберткой **ViewData**

Передача данных в представление



- **ViewData** служит для передачи данных от контроллера в представление, а не наоборот
- **ViewData** является словарем производным от **ViewDataDictionary**
- Цикл жизни **ViewData** определяется только текущим HTTP-запросом, значения **ViewData** будут потеряны, если происходит редирект
- Над значениями **ViewData** перед использованием должны быть выполнено приведение типа
- Поскольку **ViewBag** внутренне вставляет данные в словаре **ViewData**, ключи **ViewData** и свойства **ViewBag** **НЕ ДОЛЖНЫ** совпадать

Передача данных в представление

First request

http://localhost/Home/Index

```
public ActionResult Index()  
{  
    TempData["Test"] = "Test";  
}
```

Second request

http://localhost/Home/About

```
public ActionResult About()  
{  
    var test = TempData["Test"];  
}
```

Third request

http://localhost/Home/Contact

```
public ActionResult Index()  
{  
    var test = TempData["Test"];  
}
```



Example

Передача данных в представление

First request

http://localhost/Home/Index

```
public ActionResult Index()  
{  
    TempData["Test"] = "Test";  
}
```

Second request

http://localhost/Home/About

```
public ActionResult About()  
{  
    var test =  
        TempData.Keep("Test");  
}
```

Third request

http://localhost/Home/Contact

```
public ActionResult Index()  
{  
    var test = TempData["Test"];  
}
```



Строго типизированные представления

При использовании представлений на базе движка Razor, представления могут по умолчанию наследоваться от двух типов: `System.Web.Mvc.WebViewPage` или `System.Web.Mvc.WebViewPage<T>`

```
public class WebViewPage<TModel> : WebViewPage
{
    public new AjaxHelper<TModel> Ajax { get; set; }
    public new HtmlHelper<TModel> Html { get; set; }
    public new TModel Model { get; }
    public new ViewDataDictionary<TModel> ViewData { get; set; }
}
```

Добавление динамического контента в представление

Главная цель представлений - визуализировать компоненты доменной модели как компоненты пользовательского интерфейса. Для этого нужно уметь добавлять в представления **динамический контент**. Динамический контент генерируется во время выполнения и может быть разным для разных запросов. Это его отличие от **статического контента**, такого как HTML, который создается при написании приложения и который остается одинаковым для всех запросов.

Добавление динамического контента в представление

Способ	Использование
Код	Используется для создания небольших, независимых частей логики представления, например, операторы <code>if</code> или <code>foreach</code> . Это основной способ создания динамического контента в представлении, и на нем основаны некоторые другие подходы
Вспомогательные методы HTML	Используются для создания одного элемента HTML или небольшой коллекции элементов, обычно на основании данных модели представления или объекта <code>ViewData</code> . Можно использовать встроенные вспомогательные методы MVC Framework, можно также создавать свои собственные
Секции	Используются для разбиения контента на блоки, которые будут вставлены в макет в определенных местах

Добавление динамического контента в представление

Способ	Использование
Частичные представления	<p>Используются для включения подсекции разметки в несколько представлений. Частичные представления могут содержать код, вспомогательные методы HTML и ссылки на другие частичные представления. Частичные представления не могут вызывать методы действий, поэтому их нельзя использовать для выполнения бизнес-логики</p>
Дочерние действия	<p>Используются для создания повторно используемых элементов управления UI и виджетов, в которых необходима бизнес-логика. Дочернее действие вызывает метод действия, визуализирует представление и внедряет результат в поток ответа</p>

Добавление динамического контента в представление

- Вспомогательные методы используются, в основном, для того, чтобы уменьшить количество дублированного кода в представлениях, и для самого простого кода
- Для более сложной разметки и кода используются частичные представления
- В случае необходимости выполнения какие-либо манипуляции с моделью данных используются дочерние действия

Секции

Движок Razor поддерживает концепцию секций, которые позволяют выделять различные области в макете. Секции Razor позволяют разбивать представление на части и контролировать то, где они внедряются в макет

По соглашению секции определяются либо в начале, либо в конце представления, чтобы легче было увидеть, какие области контента будут рассматриваться как секции, а какие будут захвачены вспомогательным методом `RenderBody`. Часто используется и другой подход, согласно которому представление должно содержать только секции, тело представления также заключается в секцию



Example

Частичные представления

В приложениях часто используются одни и те же фрагменты тегов Razor и HTML-разметки в нескольких представлениях. Чтобы не дублировать контент, можно использовать **частичные представления**.

Частичные представления представляют собой отдельные файлы, которые содержат фрагменты кода с тегами и разметкой и могут быть включены в другие представления.

Можно также создать **строго типизированное частичное представление**, а затем передавать в него объекты моделей представлений, которые оно будет визуализировать.



Example

Дочерние действия

Дочерние действия – это методы действий, которые вызываются из представления. Они позволяют избежать дублирования логики контроллера, которую необходимо использовать в приложении несколько раз. Дочерние действия так же относятся к действиям, как частичные представления – к представлениям.

Дочерние действия чаще всего используются для отображения какого-либо управляемого данными виджета, который должен появляться на нескольких страницах и содержит данные, не относящиеся к основному действию (например, управляемое данными меню навигации, без необходимости поставлять данные о категориях навигации непосредственно от каждого действия метода)



Example

Синтаксис Razor

Razor – это движок представления, который обрабатывает ASP.NET контент и ищет инструкции, как правило, для вставки динамического контента в выходные данные, отправленные браузеру. Microsoft поддерживает два вида движков: движок ASPX работает с тегами <% и %>, которые являлись основой развития ASP.NET в течение многих лет. Движок Razor, который работает с отдельными областями контента, обозначается символом @.

- @ – оператор Razor (по умолчанию кодируется для предотвращения XSS-атак)
- @: – предотвращение Razor от интерпретации строки как C# выражения
- @{ ... } – блока кода Razor
- @* ... *@ – комментарий



Example

Синтаксис Razor

Основные синтаксические правила **Razor** для C#

- Кодовые блоки Razor заключа в `@ {...}`
- Встроенные выражения (переменные и функции) начинаются с `@`
- Утверждения заканчиваются точкой с запятой
- Переменные объявляются с ключевым словом `var`
- Строки заключаются в кавычки
- Код C# чувствителен к регистру
- Файлы C# имеют расширение `.cshtml`

http://www.w3schools.com/aspnet/razor_intro.asp



Example

Layout-представления

Автономные представления хороши для простых приложений-примеров, но реальный проект может иметь множество представлений, и **макеты (Layout)** являются эффективными шаблонами, которые содержат разметку, используемую для создания логичности и постоянства в веб-приложении. Это может заключаться в том, что в приложение будут включены необходимые JavaScript библиотеки, или в создании общего гармоничного вида всего приложения

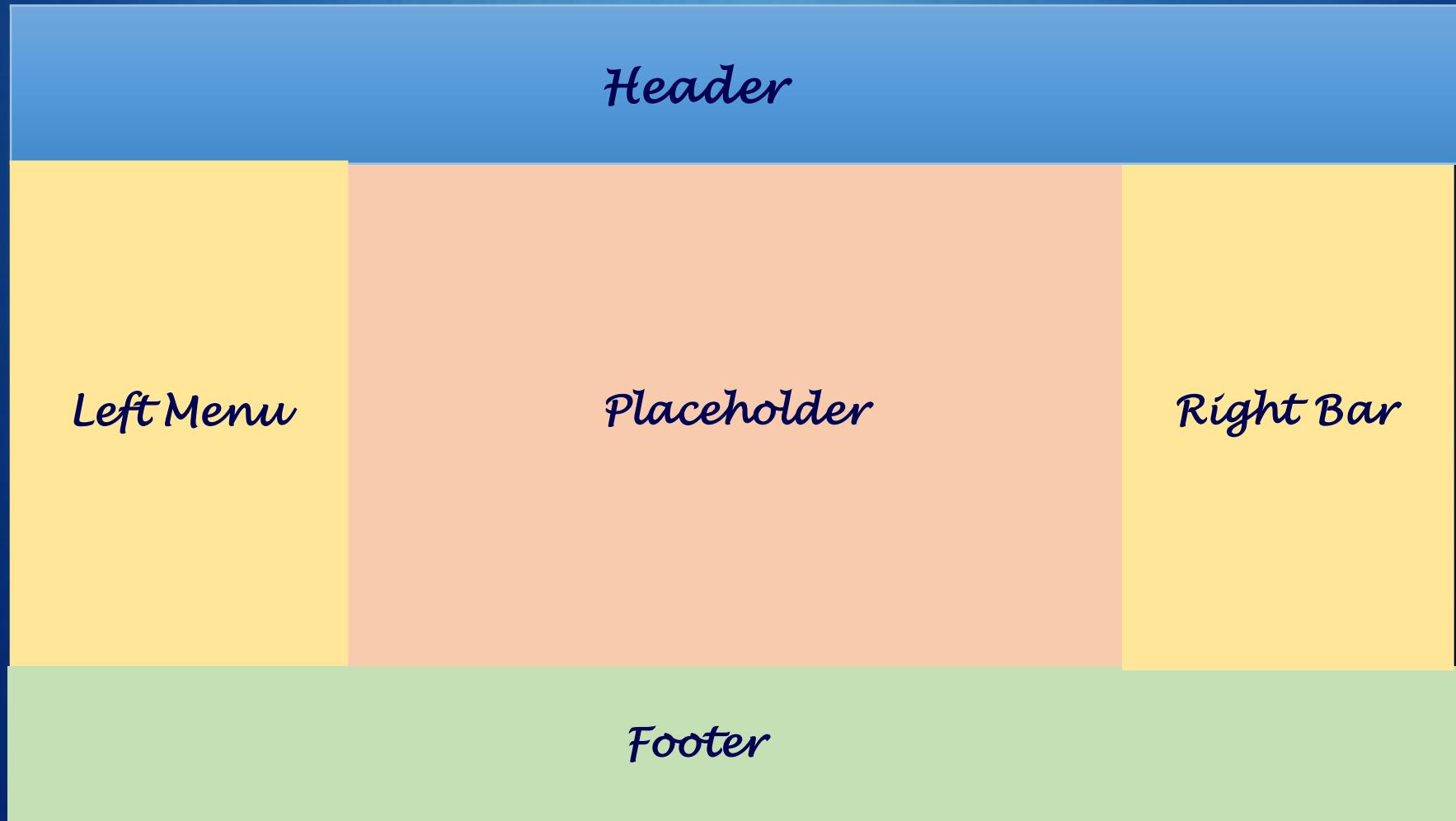


Example

Layout-представления



Layout-представления

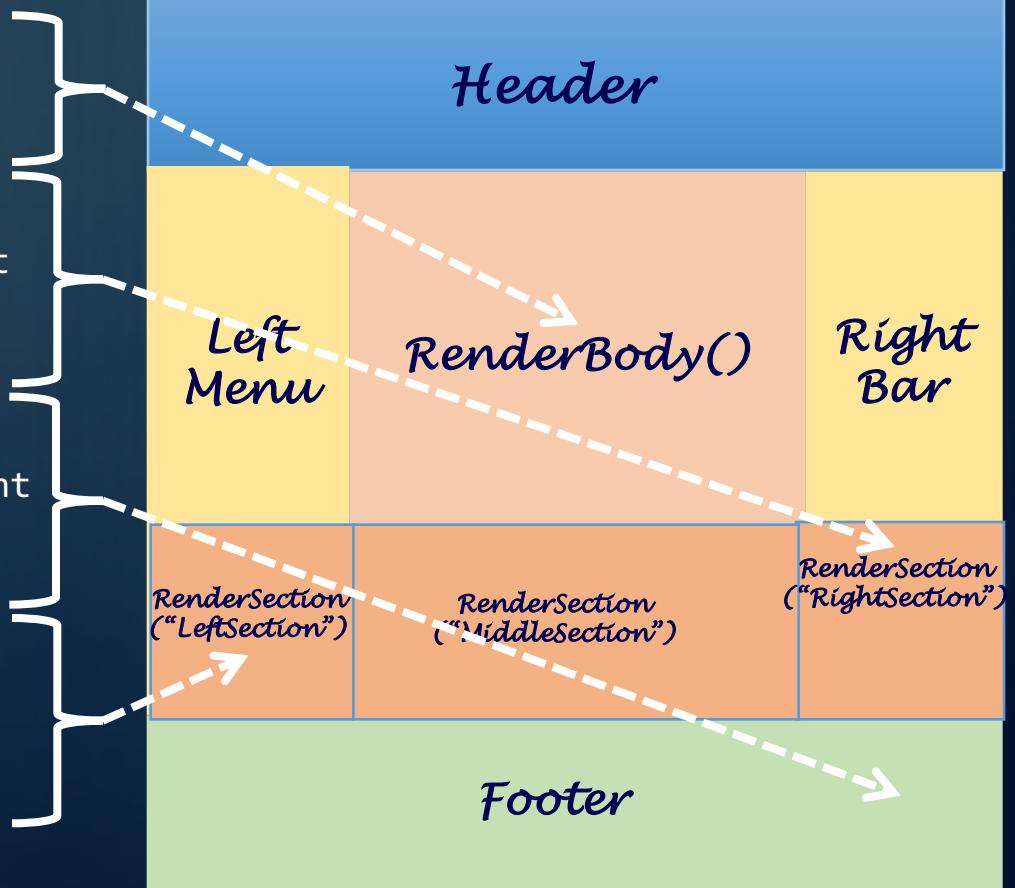


Layout-представления

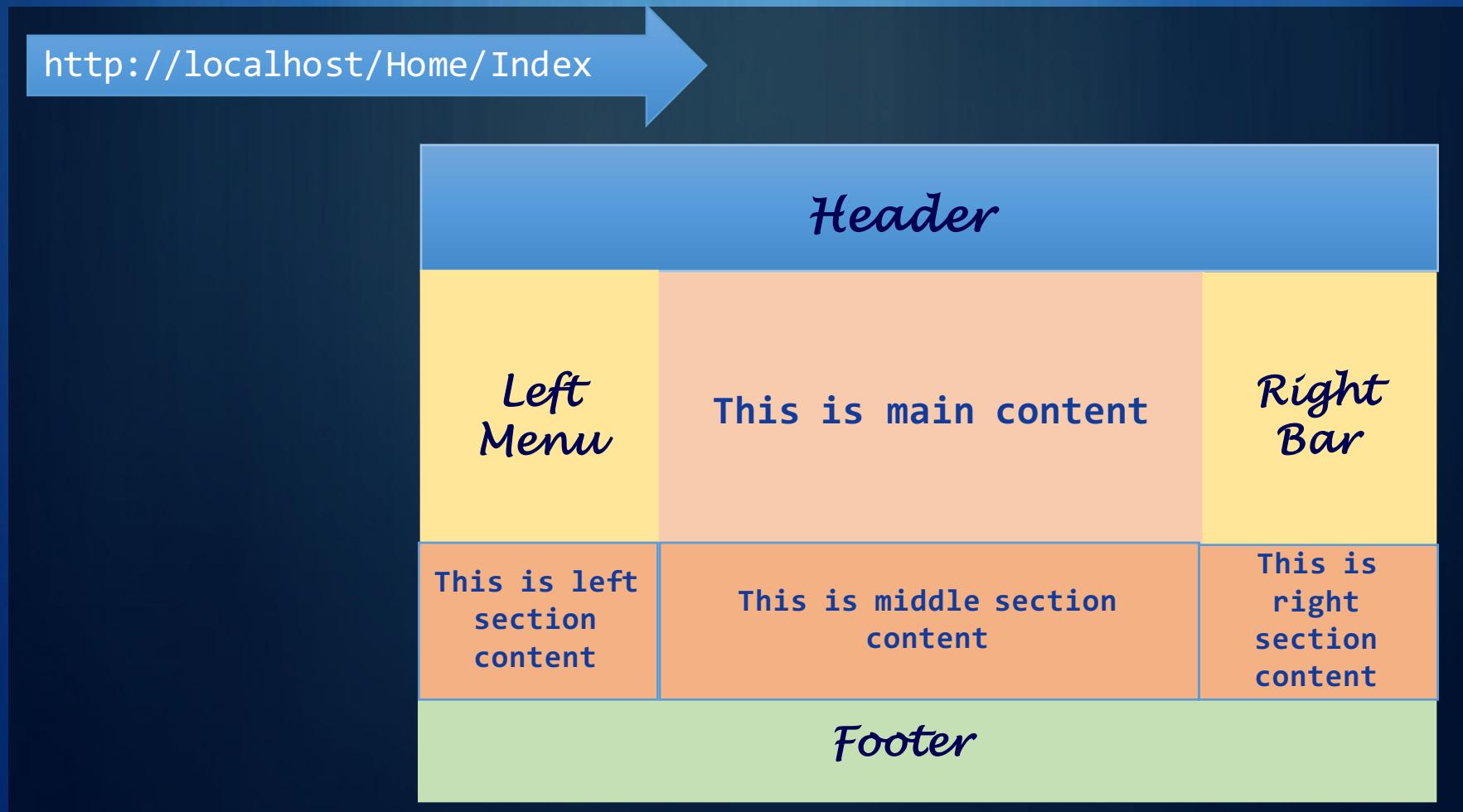
Index.cshtml

```
<div>
    This is main content
</div>
@section RightSection{
<text>
    This is right section content
</text>
}
@section MiddleSection{
<text>
    This is middle section content
</text>
}
@section LeftSection{
<text>
    This is left section content
</text>
}
```

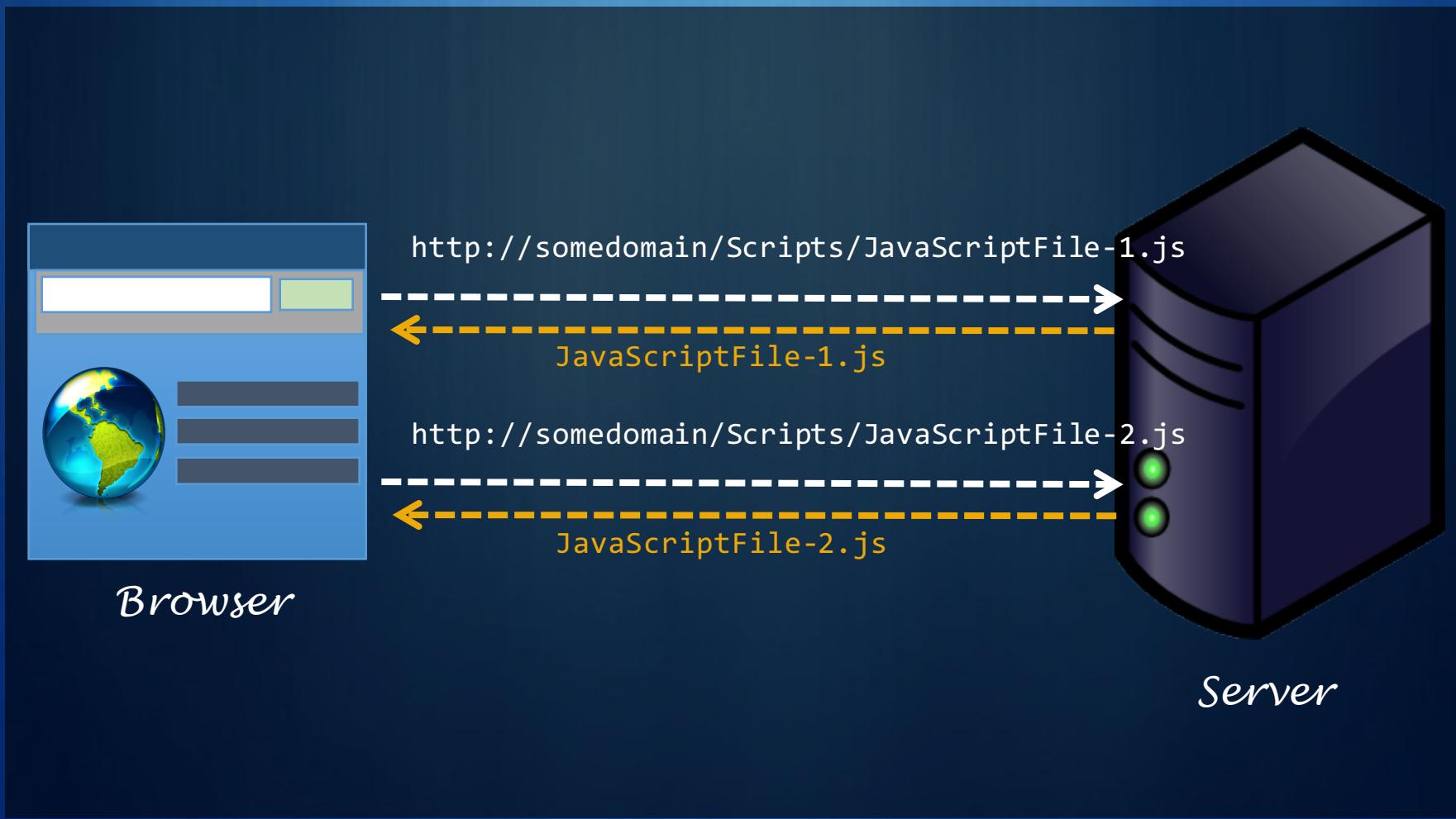
_Layout.cshtml



Layout-представления



Бандлы и минификация



Бандлы и минификация



!

Example

Бандлы и минификация

JavaScript

```
sayHello = function(name){  
    var msg = "Hello, " + name;  
    alert(msg);  
}
```

*JavaScript function
will be optimized into*

Minified JavaScript

```
sayHello = function(n){var t="Hello, "+n;alert(t);}
```



Example

Html-хелперы

Представления используют html-разметку для визуализации содержимого. Однако фреймворк ASP.NET MVC обладает также таким мощным инструментом как **HTML-хелперы**, позволяющие генерировать html-код.

- Внутренние хелперы
- Внешние хелперы
- Встроенные (built-in)-хелперы

Html-хелперы. Внутренние хелперы

Внутренние хелперы похожи на обычные определения методов на языке C#, начинающихся с тега `@helper`

```
@helper CreateList(IEnumerable<string> list)
{
    <ul>
        @foreach (var b in list)
        {
            <li>@b</li>
        }
    </ul>
}
```

Хелперы наиболее полезны, если приходится многочисленно создавать сложную, но однотипную html-разметку



Example

Html-хелперы. Внешние хелперы

В качестве альтернативы можно использовать **внешние вспомогательные методы HTML**, которые выражаются как методы расширения C#

```
public static MvcHtmlString CreateList(this HtmlHelper html,
    IEnumerable<string> items)
{
    var ul = new TagBuilder("ul");
    foreach (string item in items)
    {
        var li = new TagBuilder("li");
        li.SetInnerText(item);
        ul.InnerHtml += li.ToString();
    }
    return new MvcHtmlString(ul.ToString());
}
```



Example

Html-хелперы. Свойства класса HtmlHelper

Свойство	Описание
RouteCollection	Возвращает набор маршрутов, определенных в приложении.
ViewBag	Возвращает данные объекта из ViewBag , который был передан методом действия в представление, вызвавшее данный вспомогательный метод.
ViewContext	Возвращает объект ViewContext , который обеспечивает доступ к информации о запросе и процессе его обработки (и который мы опишем далее в этой главе). Свойство ViewContext наиболее полезно для создания контента, который адаптируется к текущему запросу

Html-хелперы. Свойства класса ViewContext

Свойство	Описание
Controller	Возвращает контроллер, обрабатывающий текущий запрос.
HttpContext	Возвращает объект HttpContext для текущего запроса.
IsChildAction	Возвращает true, если вызвавшее вспомогательный метод представление визуализируется дочерним действием.
RouteData	Возвращает данные маршрутизации для запроса.
View	Возвращает экземпляр реализации IView, которая вызвала вспомогательный метод.

Html-хелперы. Класс TagBuilder

Член	Описание
InnerHtml	Свойство, которое позволяет записать содержимое элемента как строку HTML. Значение, присвоенное этому свойству, не будет закодировано, что означает, что с его помощью можно создавать вложенные элементы HTML.
SetInnerText	Устанавливает текстовое содержимое элемента HTML. Параметр string кодируется для безопасности отображения.
AddCssClass	Добавляет класс CSS к элементу HTML.
MergeAttribute	Добавляет атрибут к элементу HTML. Первый параметр - это имя атрибут, второй - его значение. Параметр bool определяет, нужно ли заменить существующий атрибут с таким же названием.

Html-хелперы. Управление кодировкой строк

В MVC Framework для защиты от вредоносного ввода применяется автоматическое кодирование, которое позволяет гарантировать безопасность добавления данных на страницу

Браузеру запрещается интерпретировать значения данных как действительную (допустимую) разметку, так как это является основой для распространенной формы атак, при которой злоумышленники пытаются изменить поведение приложения, пытаясь добавлять свой собственный код HTML или разметку JavaScript. Razor автоматически кодирует значения данных, когда они используются в представлении

Поскольку вспомогательные методы должны генерировать HTML, то они пользуются высоким уровнем доверия со стороны движка представления - и в силу этого требуют более пристального внимания.



Example

Html-хелперы. Встроенные хелперы

Фреймворк MVC уже предоставляет большой набор встроенных html-хелперов, которые позволяют генерировать ту или иную разметку, главным образом, для работы с формами

Наиболее полезные (и наиболее часто используемые) вспомогательные методы - это `Html.BeginForm` и `Html.EndForm`. Они создают теги формы HTML и генерируют для нее допустимый атрибут `action`, основываясь на механизме маршрутизации приложения

В MVC определен широкий набор хелперов ввода практически для каждого html-элемента

Html-хелперы. Встроенные хелперы

Элемент HTML	Пример
Html.CheckBox	<code>Html.CheckBox("cccheckbox", false)</code> Вывод:<input id="cccheckbox" name="cccheckbox" type="checkbox" value="true" /><input name="cccheckbox" type="hidden" value="false" />
Html.Hidden	<code>Html.Hidden("chidden", "val")</code> Вывод:<input id="chidden" name="chidden" type="hidden" value="val" />
Html.RadioButton	<code>Html.RadioButton("cradiobutton", "val", true)</code> Вывод: <input checked="checked" id="cradiobutton" name="cradiobutton" type="radio" value="val"



Example

Html-хелперы. Встроенные хелперы

Элемент HTML

Пример

`Html.Password("cpassword", "val")`

`Html.Password`

Вывод: `<input id="cpassword" name="cpassword" type="password" value="val" />`

`Html.TextArea("ctextarea", "val", 5, 20, null)`

`Html.TextArea`

Вывод: `<textarea cols="20" id="ctextarea" name="ctextarea" rows="5"> val</textarea>`

`Html.TextBox("ctextbox", "val")`

`Html.TextBox`

Вывод: `<input id="ctextbox" name="ctextbox" type="text" value="val" />`

Html-хелперы. Встроенные хелперы

Элемент HTML

Пример

Html.DropDownList

```
Html.DropDownList("countires", new  
SelectList(new string[] {"Russia", "USA",  
"Canada", "France"}), "Countries")
```

Вывод: <select id="countires"
name="countires"><option
value="">Countries</option>
<option>Russia</option>
<option>USA</option>
<option>Canada</option>
<option>France</option>
</select>

Html.Label

```
Html.Label("Name")
```

Вывод: <label for="Name">Name</label>

HTML-хелперы. Строго типизированные хелперы

Кроме базовых хелперов в ASP.NET MVC имеются их двойники - строго типизированные хелперы. Этот вид хелперов принимает в качестве параметра лямбда-выражение, в котором указывается то свойство модели, к которому должен быть привязан данный хелпер. Важно учитывать, что строго типизированные хелперы могут использоваться только в строго типизированных представлениях, а тип модели, которая передается в хелпер, должен быть тем же, что указан для всего представления с помощью директивы `@model`.

HTML-хелперы. Строго типизированные хелперы

Базовые HTML-хелперы

```
Html.CheckBox("checkbox", false)
```

```
Html.Hidden("hidden", "val")
```

```
Html.RadioButton("radiobutton",  
    "val", true)
```

```
Html.Password("password", "val")
```

```
Html.TextArea("textarea", "val",  
    5, 20, null)
```

```
Html.TextBox("textbox", "val")
```

Строго типизированные HTML-хелперы

```
Html.CheckBoxFor(x => x.IsApproved)
```

```
Html.HiddenFor(x => x.SomeProperty)
```

```
Html.RadioButtonFor(  
    x => x.IsApproved, "val")
```

```
Html.PasswordFor(x => x.Password)
```

```
Html.TextAreaFor(x => x.Bio,  
    5, 20, new{})
```

```
Html.TextBoxFor(x => x.Name)
```

HTML-хелперы. Шаблонные хелперы

Шаблонные вспомогательные методы позволяют только указать свойство, которое необходимо визуализировать, не уточняя при этом, какой элемент HTML для него требуется - MVC Framework выясняет это самостоятельно. Это более гибкий подход к отображению данных пользователю, хотя он и требует немного больше внимания и осторожности.

HTML-хелперы. Шаблонные хелперы

- **Display** создает элемент разметки, который доступен только для чтения, для указанного свойства модели: `Html.Display("Name")`
- **DisplayFor** создает строго типизированный аналог хелпера `Display`:
`Html.DisplayFor(e => e.Name)`
- **Editor** создает элемент разметки, который доступен для редактирования, для указанного свойства модели: `Html.Editor("Name")`
- **EditorFor** строго типизированный аналог хелпера `Editor`:
`Html.EditorFor(e => e.Name)`
- **DisplayText** создает выражение для указанного свойства модели в виде простой строки: `Html.DisplayText("Name")`
- **DisplayTextFor** строго типизированный аналог хелпера `DisplayText`:
`Html.DisplayTextFor(e => e.Name)`

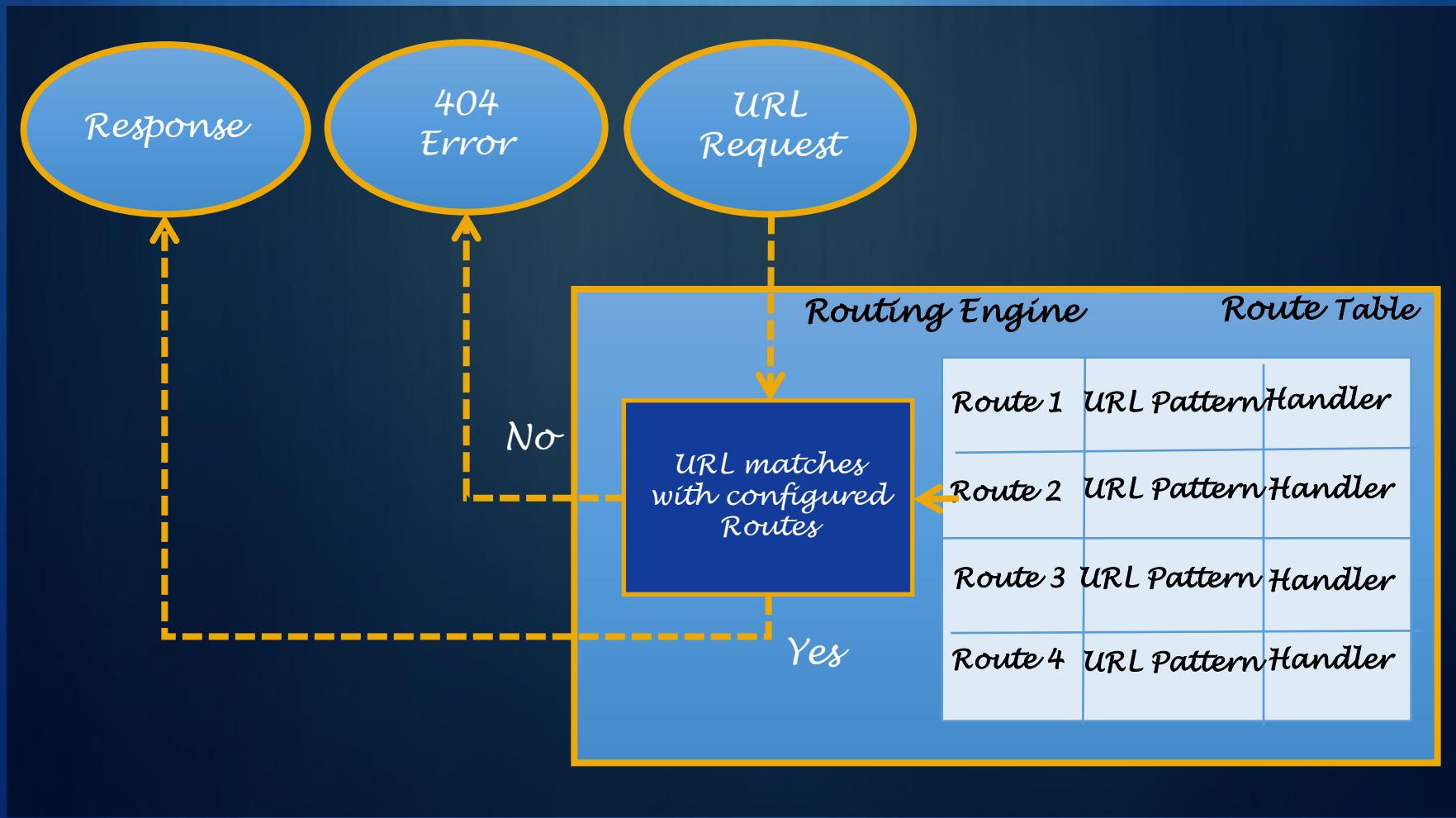
HTML-хелперы. Шаблонные хелперы

Существует несколько шаблонов, которые позволяют сгенерировать разом все поля для определенной модели:

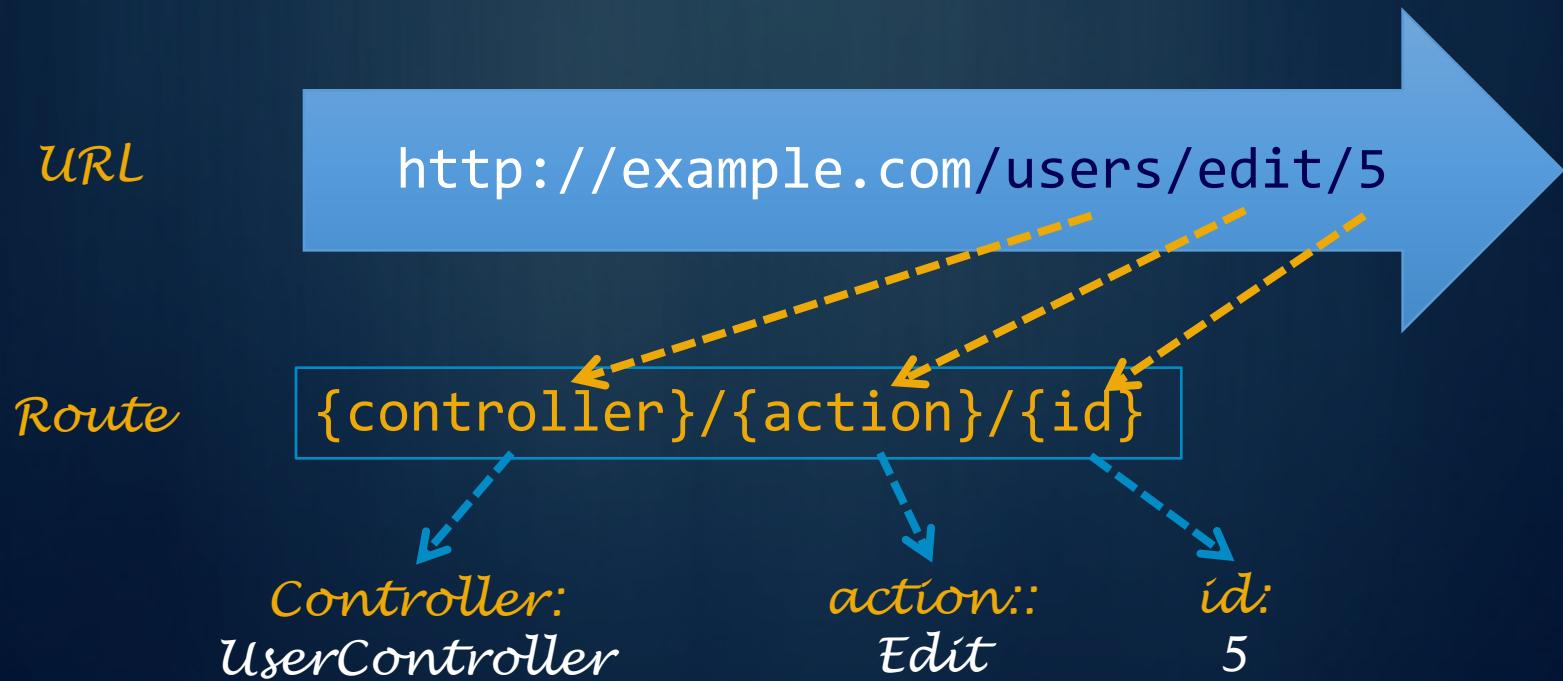
- **DisplayForModel** – создает поля для чтения для всех свойств модели:
`Html.DisplayForModel()`
- **EditorForModel** – создает поля для редактирования для всех свойств модели:
`Html.EditorForModel()`

Маршрутизация

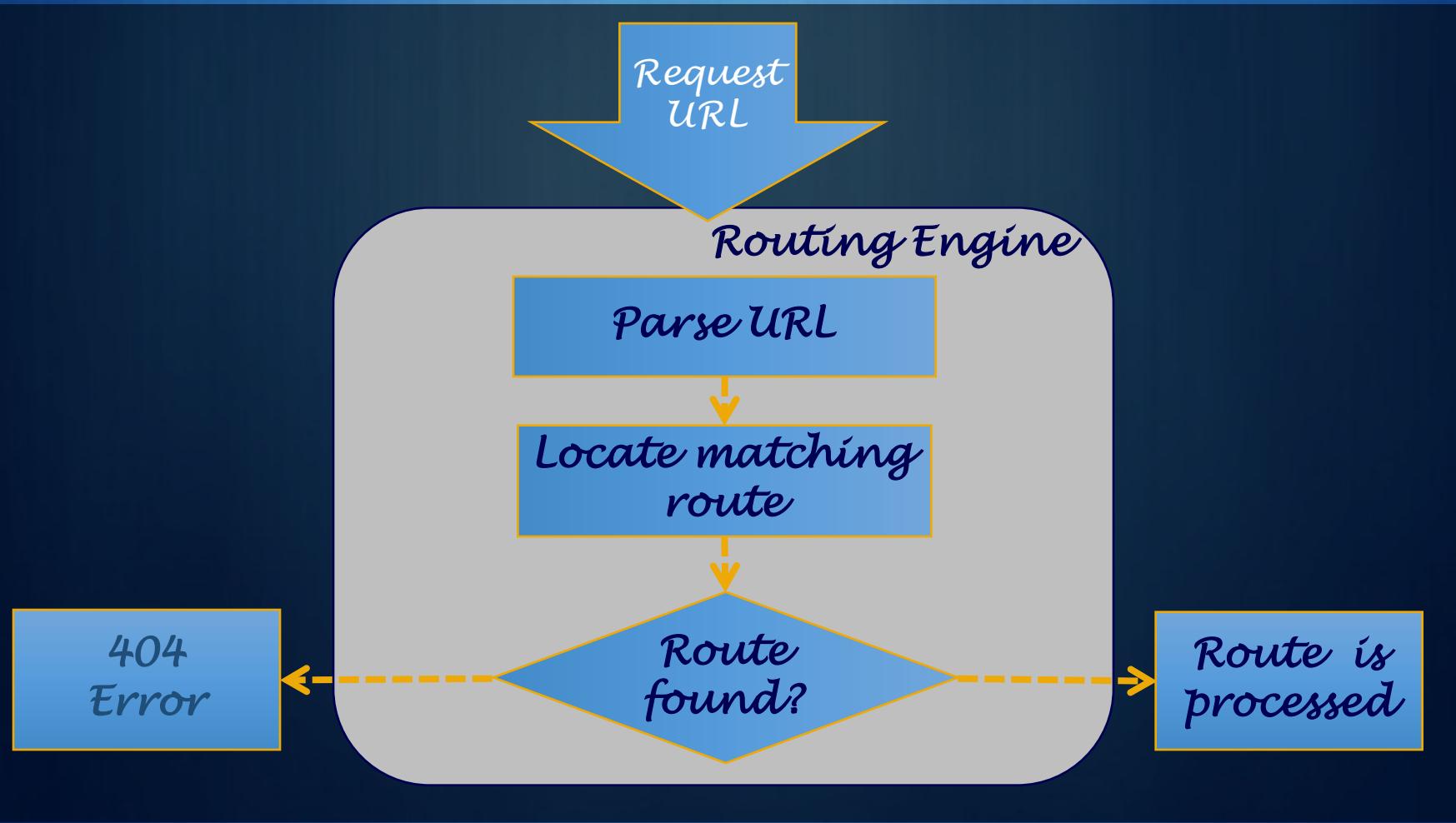
Введение в маршрутизацию



Введение в маршрутизацию



Введение в маршрутизацию



Введение в маршрутизацию

```
routes.MapRoute(  
    name: "Guestbook",  
    url: "Guestbook/{id}"  
    defaults: new {  
        controller = "Guestbook",  
        action = "Index",  
        id = UrlParameter.Optional});
```

Route name

URL Pattern

Defaults for Route

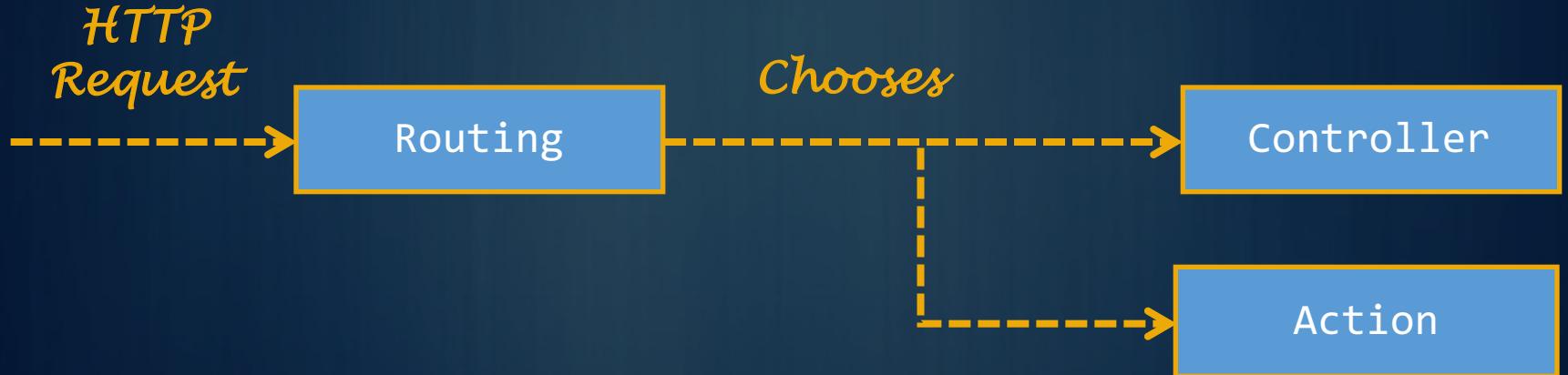
URL	Controller	Action	Id
http://localhost/Guestbook/123	GuestbookController	Index	123
http://localhost/Guestbook/Index/123	GuestbookController	Index	123
http://localhost/Guestbook?Id=123	GuestbookController	Index	123

Входящая и исходящая маршрутизация

Инфраструктура маршрутизации разбивает URL на сегменты, исходя из логики приложения. Она должна владеть двумя направлениями:

- Входящая маршрутизация – сопоставление запросов с контроллером, действием и любыми дополнительными параметрами
- Исходящая маршрутизация – построение URL-адресов, которые соответствуют схеме URL из контроллера, действия и любых дополнительных параметров

Входящая и исходящая маршрутизация



Входящая маршрутизация описывает, как URL-адрес вызывает действие контроллера. HTTP-запрос поступает в конвейер ASP.NET и передается по роутам, зарегистрированным в приложении ASP.NET MVC. Каждый роут имеет возможность обработать запрос, и затем соответствующий роут укажет, какой контроллер и действие будут использоваться.

Входящая и исходящая маршрутизация



Исходящая маршрутизация описывает механизм генерации URL для ссылок и других элементов на сайте, где используются зарегистрированные роуты

Создание схемы URL-адреса

Рекомендации проектирования схем URL:

- Создавайте простые, чистые URL.
- Создавайте интуитивно понятные URL.
- Дифференцируйте запросы с помощью URL-параметров.
- Не открывайте id из баз данных везде, где это возможно.
- Страйтесь добавлять дополнительную информацию.

<http://example.com/blog/post-1/hello-world>

http://example.com/eventmanagement/events_by_month.aspx?year=2011&month=4

vs

<http://example.com/events/2011/04>

Создание схемы URL-адреса

URL	Описание
<code>http://example.com/events</code>	Отображает все события
<code>http://example.com/events/<year></code>	Отображает все события за определенный год
<code>http://example.com/events/<year>/<month></code>	Отображает все события за определенный месяц
<code>http://example.com/events/<year>/<month>/<date></code>	Отображает все события за определенный день

Валидация

Валидация модели

Проверка достоверности – это процесс, при котором выясняется, подходят ли полученные данные для привязки модели, и если это не так, пользователь получает информацию для устранения ошибки.

Процесс проверки:

- Проверка корректности полученных данных.
- Уведомление пользователя о результате проверки, чтобы помочь исправить ошибки.

Типы проверок:

- На стороне сервера.
- На стороне клиента.

Валидация модели

Способы валидации данных:

- Явная валидация данных.
- Осуществление валидации в model binder-е. DefaultModelBinder имеет несколько полезных методов (OnModelUpdated, SetProperty), которые можно переопределить и с их помощью добавить валидацию. Затем этот binder нужно зарегистрировать в Application_Start в Global.asax.
- Валидация при помощи метаданных.
- Самовалидирующаяся модель (self-validating model). В этом случае валидационная логика будет частью самой нашей модели. Достигается это при помощи реализации интерфейса IValidatableObject.
- Кастомный провайдер валидации (custom validation provider). Достигается путем наследования от класса ModelValidationProvider и переопределения метода GetValidators. Затем необходимо зарегистрировать провайдер в Application_Start в Global.asax

Явная валидация данных

- Заключается в проверке значений, полученных после привязки данных, и регистрации обнаруженных ошибок с помощью свойства **ModelState**.
- **ModelState** - специальное временное хранилище информации типа **ModelStateDictionary**, которое доступно через свойство контроллера. В **ModelState** хранятся все параметры запроса и информация об ошибках конвертирования, возникших при привязке модели.
- Метод **AddModelError** позволяет указать название свойства, с которым возникли проблемы, и сообщение для пользователя. Если вместо названия свойства указана пустая строка, то ошибка относится ко всей модели сразу.
- Используя метод **IsValidField**, можно проверить, смог ли model binder привязать значение к этому свойству.
- Свойство **ModelState.IsValid** вернет `false`, если были зарегистрированы ошибки или если при привязке данных возникли какие-то проблемы.

Явная валидация данных

Helper-методы для отображения валидационных сообщений пользователю

- метод `Html.ValidationSummary` - может вывести все ошибки, которые были зарегистрированы. Существуют перегруженные методы. Например, можно указать, чтобы отображались только ошибки, относящиеся ко всей модели сразу.
- метод `Html.ValidationMessageFor` - отображает сообщение об ошибке для конкретного свойства.

Валидация при помощи метаданных

MVC поддерживает использование метаданных для валидации модели.

- Преимуществом использования метаданных является то, что эти правила будут проверяться везде, где будет осуществляться привязка данных. При явной реализации валидации эти правила применялись только в конкретном action-методе.
- Метаданные представляют собой валидационные атрибуты, которые **DefaultModelBinder** распознает и применит.
- Осуществляют валидацию не только **на сервере**, но и **на клиенте**.

Валидация при помощи метаданных

Compare

Пример: `[Compare("OtherProperty")]`

Два свойства должны иметь одинаковые значения. Это полезно, когда необходимо ввести некоторую информацию дважды, например, email или пароль

Range

Пример: `[Range(10, 20)]`

Численное значение (или другой тип, реализующий `IComparable`) должно принадлежать указанному диапазону

RegularExpression

Пример: `[RegularExpression("pattern")]`

Строковое значение должно удовлетворять указанному регулярному выражению – значение должно полностью удовлетворять шаблону

Required

Пример: `[Required]`

Значение не должно быть пустым или состоять только из пробелов

Валидация при помощи метаданных

StringLength

Пример: [StringLength(10)]

Строковое значение должно быть не длиннее указанной максимальной длины. Можно также указать минимальную длину: [StringLength(10, MinimumLength=2)]

Все валидационные атрибуты имеют свойство **ErrorMessage**. С его помощью можно задать свое сообщение об ошибке.

Создание пользовательских валидационных атрибутов

- Путем **наследования** от класса `ValidationAttribute`. От него наследуются все валидационные атрибуты. Основной метод, который необходимо переопределить - это метод `IsValid`.
- Путем **наследования** от уже **существующего** валидационного атрибута и расширения его функциональности.

Создание пользовательских валидационных атрибутов

```
public class MustBeTrueAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        return value is bool && (bool)value;
    }
}
```

```
public class FutureDateAttribute : RequiredAttribute
{
    public override bool IsValid(object value)
    {
        return base.IsValid(value) &&
               (DateTime)value > DateTime.Now;
    }
}
```



Example

Клиентская валидация

MVC поддерживает **unobtrusive client-side validation** – валидационные правила выражаются в виде дополнительных атрибутов к html тегам, которые обрабатывает **javascript**-библиотека **jQuery validation library**, которая включена в состав MVC и выполняет всю работу.

Клиентская валидация контролируется настройками в **Web.config**:

```
<appSettings>
    <add key="ClientValidationEnabled" value="true"/>
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
```

Включить выключить клиентскую валидацию можно также для конкретного представления в блоке Razor

```
Html.EnableClientValidation(false);
Html.EnableUnobtrusiveJavaScript(false);
```

Клиентская валидация

Для работы на клиенте нужно подключить **бандлы**:

```
bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(  
    "~/Scripts/jquery.unobtrusive*",
    "~/Scripts/jquery.validate*"));  
  
@Scripts.Render("~/bundles/jqueryval")
```



Example

Фильтры

Использование фильтров

Фильтры – это атрибуты .NET, которые добавляют дополнительные этапы в конвейер обработки запроса

```
public class AdminController : Controller
{
    // ... instance variables and constructor
    [Authorize]
    public ViewResult Index()
    {
        // ...rest of action method
    }
    [Authorize]
    public ViewResult Create()
    {
        // ...rest of action method
    }
    // ... other action methods
}
```

Использование фильтров

```
[Authorize(Roles = "admin")]
public class ExampleController : Controller
{
    [ShowMessage]
    [OutputCache(Duration = 60)]
    public ActionResult Index()
    {
        // ... action method body
    }
}
```

Использование фильтров

Тип фильтра	Интерфейсы	Реализация по умолчанию	Описание
Фильтр авторизации	IAuthorizationFilter	AuthorizeAttribute	Запускается вначале, перед любым другим фильтром или методом действия
Фильтр действия	IActionFilter	ActionFilterAttribute	Запускается до и после метода действия
Фильтр результата	IResultFilter	ActionFilterAttribute	Запускается до и после выполнения результата действия
Фильтр исключений	IExceptionFilter	HandleErrorAttribute	Запускается только в том случае, если другой фильтр, метод действия или результат действия генерирует исключение

Фильтры авторизации

Фильтры авторизации - это фильтры, которые запускаются первыми, перед любым другим фильтром или методом действия. Эти фильтры осуществляют политику авторизации, гарантируя, что методы действий могут быть вызваны только пользователями, имеющими право доступа. Фильтры авторизации реализуют интерфейс `IAuthorizationFilter`

```
public interface IAuthorizationFilter
{
    void OnAuthorization(AuthorizationContext filterContext);
}
```

Более безопасный подход - создать подкласс класса `AuthorizeAttribute`, который будет содержать самый сложный код и облегчит написание пользовательского кода авторизации

Фильтры авторизации

```
public class CustomAuthAttribute : AuthorizeAttribute
{
    private bool localAllowed;
    public CustomAuthAttribute(bool allowedParam)
    {
        localAllowed = allowedParam;
    }
    protected override bool AuthorizeCore(HttpContextBase httpContext)
    {
        if (httpContext.Request.IsLocal)
        {
            return localAllowed;
        }
        else
        {
            return true;
        }
    }
}
```

Фильтры авторизации

Класс `AuthorizeAttribute` используется как основа для пользовательского фильтра, у которого есть своя собственная реализация метода `AuthorizeCore`, который используется для выполнения общих задач авторизации.

Используя `AuthorizeAttribute` напрямую, можно определить правила авторизации с помощью двух доступных свойств этого класса

Название	Тип	Описание
Users	string	Разделенный запятыми список имен пользователей, которым разрешен доступ к методу действия.
Roles	string	Разделенный запятыми список названий ролей. Чтобы получить доступ к методу действия, пользователь должен обладать по крайней мере одной из этих ролей.

Фильтры исключений

Если при вызове метода действия было выброшено необработанное исключение, будет запущен фильтр исключений. Исключения могут поступать из:

- другого фильтра (фильтра авторизации, действия или результата)
- самого метода действия
- при выполнении результата действия

Фильтры исключений

Фильтры исключений должны реализовывать интерфейс `IExceptionFilter`

```
public interface IExceptionFilter
{
    void OnException(ExceptionContext filterContext);
}
```

Когда появится необработанное исключение, будет вызван метод `OnException`, параметром которого является объект `ExceptionContext`, наследующий `ControllerContext` и имеющий ряд полезных свойств, с помощью которых можно получить информацию о запросе

Фильтры исключений. Свойства ControllerContext

Название	Тип	Описание
Controller	ControllerBase	Возвращает объект контроллера для данного запроса
HttpContext	HttpContextBase	Обеспечивает доступ к информации о запросе и доступ к ответу
IsChildAction	bool	Возвращает true, если это дочернее действие
RequestContext	RequestContext	Предоставляет доступ к объекту HttpContext и данным маршрутизации, хотя и то, и то доступно через другие свойства
RouteData	RouteData	Возвращает данные маршрутизации для данного запроса

Фильтры исключений. Свойства ExceptionContext

В дополнение к свойствам, унаследованным от класса `ControllerContext`, класс `ExceptionContext` определяет некоторые дополнительные свойства, которые также полезны при работе с исключениями

Название	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет подробную информацию о методе действия
Result	ActionResult	Результат для метода действия; фильтр может отменить запрос, установив для этого свойства иное значение, кроме null
Exception	Exception	Необработанное исключение
ExceptionHandled	bool	Возвращает true, если другой фильтр отметил это исключение как обработанное

Фильтры исключений

```
public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        if (!filterContext.ExceptionHandled &&
            filterContext.Exception is ArgumentOutOfRangeException)
        {
            filterContext.Result = new RedirectResult("~/Content/
                                         RangeErrorPage.html");
            filterContext.ExceptionHandled = true;
        }
    }
}

[RangeException]
public ActionResult RangeTest(int id)
{ ... }
```



Example

Фильтры исключений

```
public class RangeExceptionAttribute: FilterAttribute, IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        if (!filterContext.ExceptionHandled &&
            filterContext.Exception is ArgumentOutOfRangeException)
        {
            var value = (int)((ArgumentOutOfRangeException)
                filterContext.Exception).ActualValue;
            filterContext.Result = new ViewResult()
            {
                ViewName = "RangeError",
                ViewData = new ViewDataDictionary<int>(value)
            };
            filterContext.ExceptionHandled = true;
        }
    }
}
```



Example

Встроенная обработка исключений. Свойства HandleErrorAttribute

Название	Тип	Описание
ExceptionType	Type	Тип исключения, который обрабатывается данным фильтром. Это свойство также будет обрабатывать типы исключений, которые наследуют от указанного, но будет игнорировать все другие. По умолчанию для свойства <code>ExceptionType</code> указано значение <code>System.Exception</code> , что означает, что оно будет обрабатывать все стандартные исключения.
View	string	Имя представления, которое рендерится данным фильтром. Если значение не задано, то по умолчанию используются следующие пути: <code>/Views/Имя_контроллера/Error.cshtml</code> или <code>/Views/Shared/Error.cshtml</code>
Master	string	Имя используемой мастер-страницы

Встроенная обработка исключений

```
[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException),  
    View = "RangeError")]  
public ActionResult RangeTest(int id)  
{  
    if (id < 100)  
    {  
        throw new ArgumentOutOfRangeException("id", id, "");  
    }  
    else  
    {  
        ViewBag.Message = string.Format("value of id : {0}", id);  
    }  
    return View("Index");  
}  
<system.web>  
    <customErrors mode="On"/>  
    ...  
</system.web>
```



Example

Фильтры действий

Фильтры действий позволяют проконтролировать входной контекст запроса при доступе к действию, а также выполнить определенные действия по завершению работы метода действий. Фильтр действий должен реализовать интерфейс **IActionFilter**:

```
public interface IActionFilter
{
    void OnActionExecuting(ActionExecutingContext filterContext);
    void OnActionExecuted(ActionExecutedContext filterContext);
}
```

*called before
calling the action*

*called after
calling the action*

Фильтры действий. Свойства ActionExecutingContext

Свойство	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет информацию о вызываемом методе действия
Result	ActionResult	Результат метода действий

Фильтры действий. Свойства ActionExecutedContext

Свойство	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет информацию о вызываемом методе действия
Result	ActionResult	Результат метода действий
Canceled	bool	Хранит значение, показывающее, отменен ли вызов действия. Если имеет значение true, если вызов действия был отменен другим фильтром
Exception	Exception	Возвращает исключение, выбрасываемое данным методом действий или другим фильтром
ExceptionHandled	bool	Хранит значение, показывающее, обработано ли исключение

Фильтры результатов

Фильтры результатов во многом похожи на фильтры действий, поскольку так же могут срабатывать как до возвращения результата действия, так и после. Фильтры результатов реализуют интерфейс **IResultFilter**:

```
public interface IResultFilter
{
    void OnResultExecuting(ResultExecutingContext filterContext);
    void OnResultExecuted(ResultExecutedContext filterContext);
}
```

before

after

Фильтры действий и результатов

Фильтры действий и результатов объединены в одну реализацию - абстрактный класс `ActionFilterAttribute`, который объединяет черты обоих фильтров:

```
public abstract class ActionFilterAttribute : FilterAttribute,  
    IActionFilter,  
    IResultFilter  
{  
    public virtual void OnActionExecuting(ActionExecutingContext  
        filterContext) {}  
    public virtual void OnActionExecuted(ActionExecutedContext  
        filterContext) {}  
    public virtual void OnResultExecuting(ResultExecutingContext  
        filterContext) {}  
    public virtual void OnResultExecuted(ResultExecutedContext  
        filterContext) {}  
}
```



Example

Регистрация фильтров

Существует три уровня подключения фильтров:

- Глобальный уровень (**Global Level**)
- Уровень контроллера (**Controller level**)
- Уровень метода действия (**Action method level**)

Регистрация фильтров

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    }
}

public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

The diagram illustrates the call flow between the code snippets. A dashed arrow points from the closing brace of the Application_Start() method to the opening brace of the RegisterGlobalFilters() method in the FilterConfig class. Another dashed arrow points from the opening brace of the RegisterGlobalFilters() method to the Add() method call, indicating the flow of execution from the application's global filter configuration setup into the specific filter registration.

Global Level

Регистрация фильтров

```
[HandleError] <----- Controller level
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}

public class HomeController : Controller
{
    [HandleError] <----- Action method level
    public ActionResult Index()
    {
        return View();
    }
}
```

Использование встроенных фильтров

Фильтр `RequireHttps` - заставляет использовать протокол HTTPS, а браузер перенаправит пользователя на то же действие, только с префиксом `https`, применяется только к GET-запросам.

Фильтр `OutputCache` - сообщает MVC-фреймворку кэшировать вывод метода действия, чтобы полученный контент можно было в дальнейшем использовать повторно, что может увеличить производительность, особенно когда идет речь о выборке из базы данных, которая может занимать значительное время. С помощью параметра `Duration` мы можем настроить время (в секундах):

```
[OutputCache (Duration=360)]
public ActionResult Index()
{
    //.....
}
```

Использование встроенных фильтров

Фильтр `ValidateAntiForgeryToken` предназначен для противодействия подделке межсайтовых запросов, производя верификацию токенов при обращении к методу действия.

```
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid && WebSecurity.Login(model.Email,
        model.Password, persistCookie: model.RememberMe))
    {
        return RedirectToAction(returnUrl);
    }

    ModelState.AddModelError("", "Invalid login or password");
    return View(model);
}
```

Введение в AJAX

Что такое AJAX

AJAX (аббревиатура от «Asynchronous Javascript And Xml») — технология обращения гибкого взаимодействия между клиентом и сервером без перезагрузки страницы.

За счет этого уменьшается время отклика и веб-приложение по интерактивности больше напоминает десктоп.

Несмотря на то, что в названии технологии присутствует буква X (от слова XML), использовать XML вовсе не обязательно. Под AJAX подразумевают любое общение с сервером без перезагрузки страницы, организованное при помощи JavaScript

Какие технологии включает AJAX

AJAX - это набор технологий, которые поддерживаются веб-браузерами. AJAX использует:

- HTML в качестве "каркаса"
- CSS для оформления
- DOM для извлечения или изменения информации на странице
- Объект XMLHttpRequest для асинхронного обмена данными с сервером
- JavaScript для связи перечисленных выше технологий между собой

Что можно сделать с помощью AJAX

Элементы интерфейса

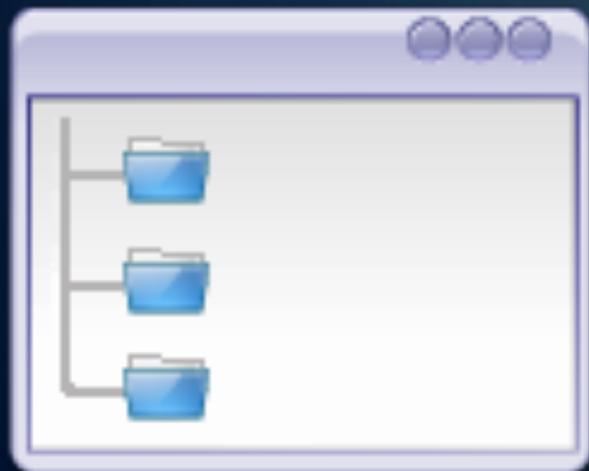
В первую очередь AJAX полезен для форм и кнопок, связанных с элементарными действиями: добавить в корзину, подписаться, и т.п. Сейчас — в порядке вещей, что такие действия на сайтах осуществляются без перезагрузки страницы.



Что можно сделать с помощью АЈАХ

Динамическая подгрузка данных

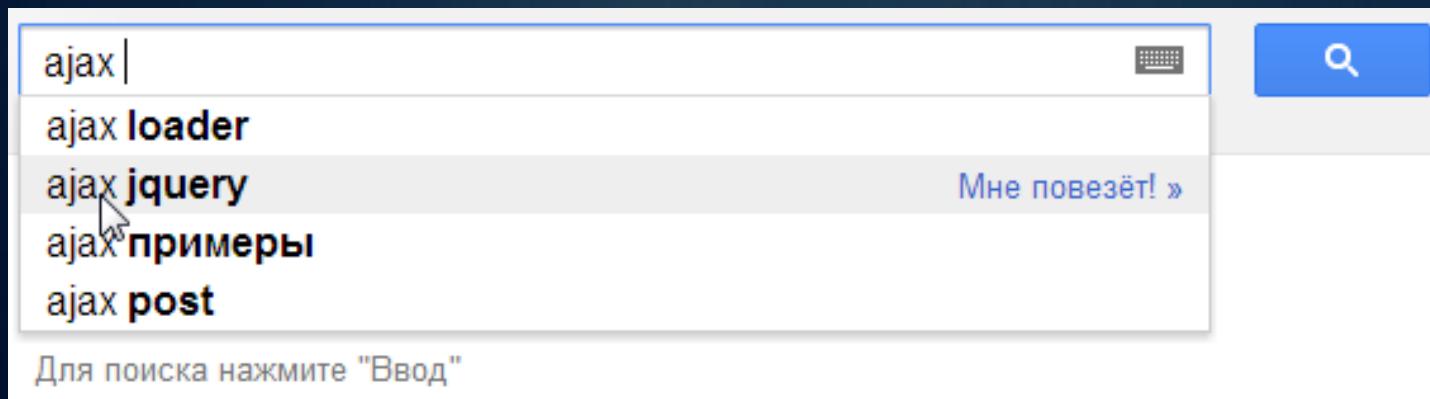
Например, дерево, которое при раскрытии узла запрашивает данные у сервера.



Что можно сделать с помощью AJAX

Живой поиск

Живой поиск — классический пример использования AJAX, взятый на вооружение современными поисковыми системами. Пользователь начинает печатать поисковую фразу, а JavaScript предлагает возможные варианты, получая список самых вероятных дополнений с сервера.



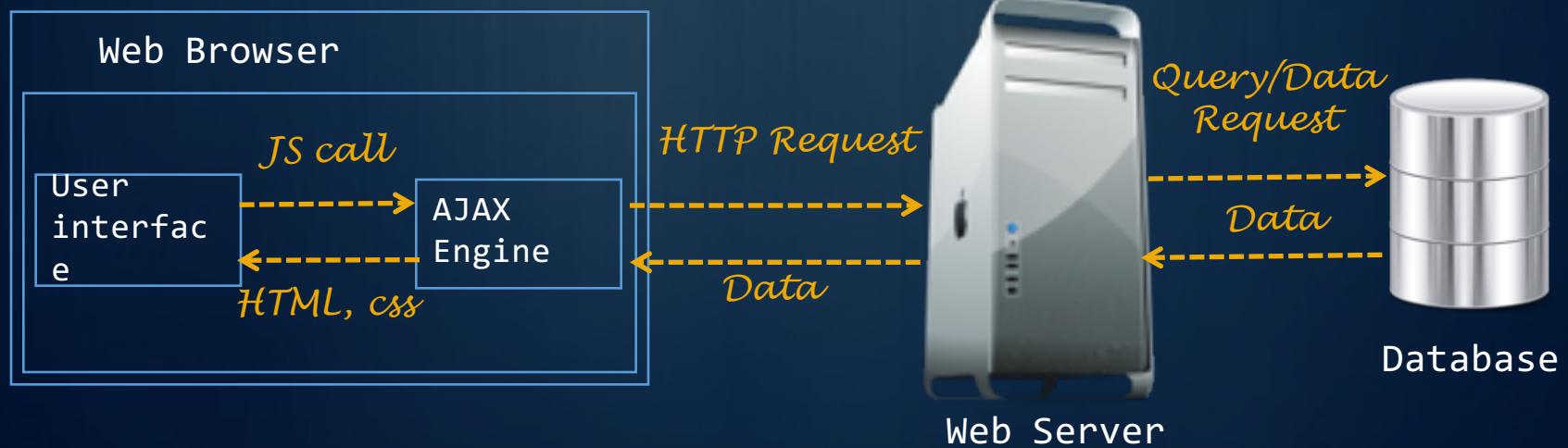
Как работает AJAX

Традиционное веб-приложение



Как работает AJAX

Веб-приложение с применением AJAX

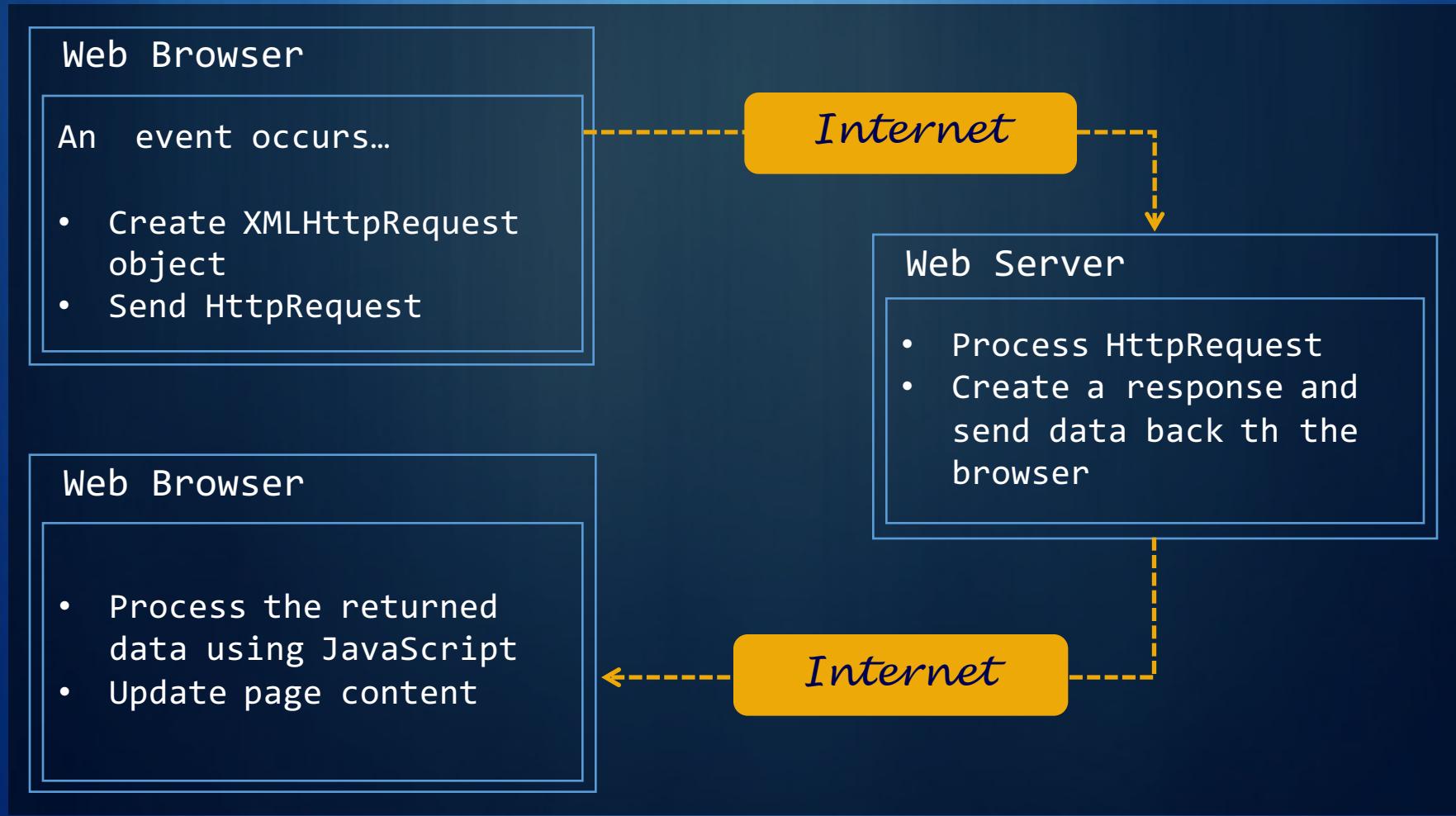


Как работает AJAX

Технически, с помощью AJAX можно обмениваться любыми данными с сервером, для этого обычно используются форматы:

- JSON — для отправки и получения структурированных данных, объектов
- XML — если сервер работает в формате XML
- HTML/текст — можно загрузить с сервера код HTML или текст для показа на странице
- Бинарные данные, файлы — гораздо реже (в современных браузерах для них есть удобные средства)

Как работает AJAX



Основы XMLHttpRequest

Объект **XMLHttpRequest** (кратко его называют, «XHR») дает возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы. Несмотря на слово «XML» в названии, XMLHttpRequest может работать с любыми данными, а не только с XML.

<http://www.w3.org/TR/XMLHttpRequest/>

Основы XMLHttpRequest

Создание объекта XMLHttpRequest

Отправка запроса на сервер с помощью методов open и send

Получение сервером запроса, обработка и отправление ответа

Получение и обработка ответа сервера с помощью responseText и onreadystatechange

Основы XMLHttpRequest

Синтаксис для создания экземпляра объекта XMLHttpRequest:

```
var xhr;  
/* Если объект XMLHttpRequest существует, значит мы имеем дело с  
современным браузером Chrome, Firefox, Safari, Opera или IE7 и выше.  
*/  
if (window.XMLHttpRequest){  
    xhr = new XMLHttpRequest();  
}  
/* Если же объект XMLHttpRequest не существует значит мы имеем дело  
с IE6 и нам придется воспользоваться специальным синтаксисом */  
else {  
    xhr = new ActiveXObject("Microsoft.XMLHTTP");  
}
```

Основы XMLHttpRequest

Настроить: open – xhr.open(method, URL, async, user, password)

Основные параметры запроса:

- **method** – HTTP-метод. Как правило, используется GET либо POST, хотя доступны и TRACE/DELETE/PUT и т.п.
- **URL** – адрес запроса, можно использовать не только **http/https**, но и другие протоколы, например **ftp://** и **file://**. При этом есть ограничения безопасности, называемые «Same Origin Policy»: запрос со страницы можно отправлять только на тот же **протокол://домен:порт**, с которого она пришла
- **async** – если установлено в **false**, то запрос производится синхронно, если **true** – асинхронно
- **user, password** – логин и пароль для HTTP-авторизации, если нужны



Вызов метода open не открывает соединение, а лишь настраивает запрос!

Основы XMLHttpRequest

Отослать данные: `send – xhr.send([body])`

Метод `send` открывает соединение и отправляет запрос на сервер. В `body` находится тело запроса. Не у всякого запроса есть тело, например у `GET`-запросов тела нет, а у `POST` – основные данные как раз передаются через `body`.

Отмена: `abort – xhr.abort()` прерывает выполнение запроса

Основы XMLHttpRequest

Ответ: **status, statusText, responseText, responseXML**

Основные свойства, содержащие ответ сервера:

- **status** HTTP-код ответа: 200, 404, 403 и так далее. Может быть также равен 0, если сервер не ответил или при запросе на другой домен
- **statusText** текстовое описание статуса от сервера: OK Not Found, Forbidden и т. д.
- **responseText** текст ответа сервера
- **responseXML** если сервер вернул XML, снабдив его правильным заголовком Content-type: text/xml, то браузер создаст из него XML-документ. По нему можно будет делать запросы `xhr.responseXml.querySelector("...")` и другие. Используется редко, так как обычно используют не XML, а JSON. То есть, сервер возвращает JSON в виде текста, который браузер превращает в объект вызовом `JSON.parse(xhr.responseText)`

Основы XMLHttpRequest

Событие `readystatechange` происходит несколько раз в процессе отсылки и получения ответа. При этом можно посмотреть «текущее состояние запроса» в свойстве `xhr.readyState`.

Состояния по спецификации:

```
const unsigned short UNSENT = 0; // начальное состояние  
const unsigned short OPENED = 1; // вызван open  
const unsigned short HEADERS_RECEIVED = 2; // получены заголовки  
const unsigned short LOADING = 3; // загружается тело (получен  
очередной пакет данных)  
const unsigned short DONE = 4; // запрос завершён
```

Запрос проходит их в порядке $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow 3 \rightarrow 4$, состояние 3 повторяется при каждом получении очередного пакета данных по сети.

Основы XMLHttpRequest

XMLHttpRequest умеет как указывать свои HTTP-заголовки в запросе, так и читать присланные в ответ.

Для работы с HTTP-заголовками есть 3 метода:

- `setRequestHeader(name, value)` устанавливает заголовок `name` запроса со значением `value`. В целях безопасности и для контроля корректности запроса существуют ограничения на заголовки - нельзя установить заголовки, которые контролирует браузер, например Referer или Host и ряд других. Отменить `setRequestHeader` невозможно, повторные вызовы лишь добавляют информацию к заголовку
- `getResponseHeader(name)` возвращает значение заголовка ответа `name`, кроме Set-Cookie и Set-Cookie2
- `getAllResponseHeaders()` возвращает все заголовки ответа, кроме Set-Cookie и Set-Cookie2

Основы XMLHttpRequest

Современная спецификация предусматривает следующие события по ходу обработки запроса:

- **loadstart** – запрос начат
- **progress** – браузер получил очередной пакет данных, можно прочитать текущие полученные данные в `responseText`
- **abort** – запрос был отменен вызовом `xhr.abort()`
- **error** – произошла ошибка
- **load** – запрос был успешно (без ошибок) завершен
- **timeout** – запрос был прекращен по таймауту
- **loadend** – запрос был завершен (успешно или неуспешно)

Используя эти события можно более удобно отслеживать загрузку (`onload`) и ошибку (`onerror`), а также количество загруженных данных (`onprogress`). Событие **readystatechange** появилось гораздо раньше, еще до появления текущего стандарта. В современных браузерах от него можно отказаться в пользу других, необходимо лишь, учесть особенности IE8-9

<http://learn.javascript.ru/ajax>

Ненавязчивый AJAX ASP.NET MVC

Применительно к ASP.NET MVC использование AJAX вылилось в целую концепцию под названием "ненавязчивого AJAX" и ненавязчивого JavaScript (unobtrusive Ajax/JavaScript). Смысл этой концепции заключается в том, что весь необходимый код JavaScript используется не на самой веб-странице, а помещается в отдельные файлы с расширением *.js. А затем с помощью тега <script> на веб-странице дается ссылка на данный файл кода.

AJAX-хелперы

Хелпер	Описание
Ajax.ActionLink	Создает гиперссылку на действие контроллера, по нажатию на которую происходит ajax-запрос к этому действию
Ajax.RouteLink	Похож на хелпер Ajax.ActionLink, только ссылка создается на определенный маршрут, а не на действие контроллера
Ajax.BeginForm	Создает html-форму, которая отправляет ajax-запросы к определенному действию определенного контроллера
Ajax.BeginRouteForm	Похож на Ajax.BeginForm, только ajax-запросы направляются не к действию контроллера, к по определенному маршруту

AJAX-хелперы

Хелпер	Описание
Ajax.GlobalizationScript	Создает ссылку на скрипт, который содержит информацию о культуре
Ajax.JavaScriptStringEncode	Кодирует строку для использования в JavaScript

Спасибо за внимание!

Обратная связь

Надеюсь, что Вы найдете этот материал полезным.

Если Вы нашли ошибки или неточности в этом или знаете, как его улучшить, пожалуйста, сообщите мне по электронному адресу: anzhelika_kravchuk@epam.com с пометкой [ASP.MVC Training Course Feedback]

Спасибо.