Сопоставление адреса URL или **URL matching** представляет процесс сопоставления запроса с конечной точкой. Данный процесс основывается на пути запроса и полученных в запросе заголовках. Данный процесс проходит ряд этапов:

1. Сначала выбираются все конечные точки, шаблон маршрута которых совпадает с путем запроса
2. Далее из полученного на предыдущем этапе набора конечных точек удаляются те, которые не соответствуют ограничениям маршрута
3. Затем из полученного на предыдущем этапе набора конечных точек удаляются те, которые не удавлетворяют политике объекта MatcherPolicy (вкратце: класс **MatcherPolicy** позволяет определить порядок сравнения конечных точек и адреса URL)
4. И в самом конце применяется объект **EndpointSelector** для выбора из полученного на предыдущем этапе списка конечной точки, которая в конечном счете и будет обрабатывать запрос

Приоритет конечных точек зависит от двух факторов:

* Порядок следования в наборе конечных точек
* Приоритетность шаблона маршрута

Приоритетность шаблонов маршрута зависит от специфичности шаблона. Специфичность шаблона определяется на основе следующих критериев:

* Шаблон маршрута с большим количеством сегментов более специфичен, чем шаблон меньшим количеством сегментов
* Сегмент с текстовым литералом (статический сегмент) более специфичен, чем сегмент с параметром маршрута
* Сегмент с параметром, к которому применяется ограничение маршрута, более специфичен, чем сегмент с параметром без ограничения
* Комплексный сегмент более специфичен, чем сегмент с параметром с ограничением
* Параметр catch-all (параметр, который соответствует неопределенному количеству сегментов) наименее специфичен

Если в конечном счете осталось две и более конечных точек, которые соответствуют запрошенному адресу, и соответственно система маршрутизации не может выбрать, какая из этих конечных точек должна обрабатывать маршрут, то генерируется исключение.

Кроме конечных точек запрос в конвейере обработки могут обрабатывать и другие компоненты middleware. При этом надо учитывать общий процесс обработки запроса и вызова конечных точек.

Так, если приложение содержит конечные точки, то система маршрутизации на основе процессе URL matching или сопоставления адреса URL с шаблонами маршрута выбирает для обработки определенную конечную точку. Если в приложении есть такая конечная точка, которая соответствует запросу, то компонент middleware **Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware** устанавливает у объекта HttpContext конечную точку для будущей обработки запроса, которую можно получить с помощью метода **HttpContext.GetEndpoint()**. Кроме того, устанавливаются значения маршрута, которые можно получить через коллекцию **HttpRequest.RouteValues**

Конфигурация

Важную роль в приложении играет конфигурация, которая определяет базовые настройки приложения. Приложение ASP.NET Core может получать конфигурационные настройки из следующих источников:

* Аргументы командной строки
* Переменные среды окружения
* Объекты .NET в памяти
* Файлы (json, xml, ini)
* Azure
* Можно использовать свои кастомные источники и под них создавать провайдеры конфигурации

**Интерфейс IConfiguration**

Данный интерфейс содержит следующие компоненты:

* **this [string key]**: индексатор, через который можно получить по ключу значение параметра конфигурации. Стоит отметить, что и ключ, и значение параметра конфигурации представляет собой объект типа string
* **GetChildren()**: возвращает набор подсекций текущей секции конфигурации в виде объекта IEnumerable<IConfigurationSection>
* **GetReloadToken()**: возвращает объект IChangeToken, который применяется для отслеживания изменения конфигурации
* **GetSection(string key)**: возвращает секцию конфигурации, которая соответствует ключу key
* Свойство **Providers** возвращает коллекцию применяемых провайдеров конфигурации. Каждый провайдер конфигурации представляет объект IConfigurationProvider
* Метод **Reload()** перезагружает значения из всех применяемых источников конфигурации

Эта конфигурация загружается из разных источников в следующем порядке:

* Сначала в качестве провайдера конфигурации устанавливается объект класса **ChainedConfigurationProvider**, который добавляет объект IConfiguration в качестве источника конфигурации и который фактически соединяет все далее применяемые провайдеры конфигурации в одну цепочку.
* Далее загружается конфигурация из файла **appsettings.json**
* Затем загружается конфигурация из файла **appsettings.[Environment].json**, где [Environment] передает название окружения, например, **appsettings.Production.json** и **appsettings.Development.json**, благодаря чему мы можем задать конфигурацию для разных состояний проекта
* Если проект запущен в окружении с именем "Development", загружается конфигурация App secrets (секретные настройки)
* Затем загружаются переменные среды
* В конце загружаются аргументы командной строки

Как мы видим, объект конфигурации может использовать одновременно сразу несколько источников и провайдеров конфигурации. Все подключемые источники конфигурации считываются в том порядке, в котором они были добавлены. Если разные источники конфигурации содержат одинаковые ключи, то используется значение, последнего добавленного источника конфигурации. Например, в данном случае значения из appsettings.Develoment.json перекроют значения с теми же ключами из appsettings.json.

АНАЛИЗ КОНФИГУРАЦИИ

Для работы с конфигурацией интерфейс IConfiguration определяет следующие методы:

* **GetSection(name)**: возвращает объект IConfiguration, который представляет только определенную секцию name
* **GetChildren()**: возвращает все подсекции текущего объекта конфигурации в виде набора объектов IConfiguration
* **GetReloadToken()**: возвращает токен - объект IChangeToken, который используется для уведомления при изменении конфигурации
* **GetConnectionString(name)**: эквивалентен вызову GetSection("ConnectionStrings")[name] и предназначается непосредственно для работы со строками подключения к различным базам даных
* **[key]**: индексатор, который позволяет получить по определенному ключу key хранящееся значение