

Routing

ASP.NET Web API 2

Routing em ASP.NET Web API 2

Existem 2 tipos de routing em ASP.NET Web API 2: Conventions-based routing e Attribute routing.

Conventions-based routing usa o nome do **controller** e os nomes + parâmetros (ID) dos **métodos** do Controller (GetAgentes, PutAgentes, PostAgentes, DeleteAgentes) para determinar a rota (ex: GET api/agentes, PUT api/agentes/5, etc.). É simples, mas limitado.

Attribute routing é uma forma de definir rotas *custom* directamente no controller. É muito mais flexível e poderoso do que o primeiro método, mas implica colocar os atributos [Route] e [HttpGet] (ou [HttpPost, HttpPut, ...]) nos métodos que queremos *expor* ao cliente.

Conventions-based routing

Configurado no WebApiConfig.cs através do “[MapHttpRoute](#)”, tal como no MVC. O template por defeito adiciona o prefixo “api/” seguido do nome do controller, e um parâmetro para o ID, que é opcional.

O **prefixo** do nome do controller (“AgentesController” > “Agentes”) indica o URL a usar (“api/agentes”). O **prefixo** “Get”, “Put”, “Post”, ou “Delete” indica o método HTTP ao qual o método vai responder. O resto do nome do método é irrelevante.

Problema principal: É muito limitado naquilo que permite fazer. Se quiser fazer algo como obter as multas de um agente, [tenho que configurar uma rota no WebApiConfig.cs](#). Felizmente, a versão 2 da Web API introduziu Attribute Routing.

Attribute Routing

Tal como conventions-based routing, é também configurado no WebApiConfig.cs, através do método “[MapAttributeRoutes](#)”. Todas as outras configurações de rotas são feitas nos controllers e nos seus métodos.

Todos os métodos que quero expor usam o [\[Route\]](#) e o [\[HttpGet\]](#), [\[HttpPost\]](#), etc. No [\[Route\]](#) também posso definir parâmetros (placeholders).

No controller, é comum usar-se o [\[RoutePrefix\]](#) para definir o caminho base **comum** a todos os métodos do controller. Permite-nos “não repetir” a parte inicial (ex: “api/agentes”) em todos os [\[Route\]](#).

Attribute Routing - Placeholders e Parâmetros

O [\[Route\]](#) permite que se coloquem parâmetros. Os parâmetros são colocados entre chavetas (ex: {id}), e os valores dos placeholders no URL são mapeados aos parâmetros no método, **através do nome**.

Exemplo: Se tenho um [\[Route\("{id}/multas"\)\]](#), o “id” será colocado no parâmetro “id” do método do controller (ex: int **id**).

Parâmetros podem ser **opcionais** se tiverem um “?” após no nome (ex: {nome?}), mas isto é pouco usado (é costume usar a query string nestes casos). Se o valor não estiver definido, o parâmetro fica com o valor **null**. Placeholders também podem ter constraints, caso queira restringir situações em que só são permitidos números (ou regex).

Attribute Routing - Vantagens e Desvantagens

Vantagem: A configuração das rotas fica junto à lógica (no controller).

Vantagem: Muito mais flexível e poderoso, com parâmetros opcionais, constraints, e caminhos custom (nota: conventions-based routing também permite isto, mas é mais difícil).

Vantagem: Pode ser usado em conjunto com conventions-based routing. Pode ser usado para os [casos mais complexos](#), e usar conventions-based para o resto.

Desvantagem: É fácil fazer asneira, porque temos controlo total sobre os URLs. Por “asneira” é mais “má prática” segundo os princípios REST (fica para outra aula...).

Mais informações

Attribute Routing in ASP.NET Web API 2:

https://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2?__hstc=250130137.d519671a34e3a56a4c773f89949c75cd.1488067200081.1488067200082.1488067200083.1&__hssc=250130137.1.1488067200084&__hsfp=528229161

View Models

Web API 2 e MVC

View Models

View Models são models. A única diferença é que são Modelos *especializados* para enviar ou receber dados para/de uma View, e não são (geralmente) usados numa BD.

São usados quando queremos enviar mais dados para uma view, sem fazer uso da ViewBag (sugestão: usem a ViewBag apenas para coisas simples), ou quando queremos receber dados diferentes daquilo que são os objetos da base de dados (mais ou menos campos, ou dados diferentes), situações que poderão causar problemas de validação.

Em algumas situações, são até usados por questões de segurança, para prevenir ataques de Overposting (também conhecido como Mass Assignment), problema comum em frameworks que fazem uso de Model Binding (ASP.NET MVC, Web API 2, Spring, etc.).

View Models - Casos de Uso

- Quando crio um agente, necessito da Fotografia. A fotografia é uma string na classe “Agente”, mas o upload é um HttpPostedFileBase. Em vez de receber o ficheiro de upload por parâmetro, uso um View Model que valida a presença e o tipo da imagem.
- Quando “passo uma multa”, necessito de saber quem é o agente, o condutor, e a viatura. Em vez de passar os dados por ViewBag (o template default faz isto), uso um View Model com dados para as drop down lists, preencho-o, e uso-o. O código fica mais resistente a refactorings.
- Quando quero “limitar” os dados recebidos do cliente (ex: quando edito um agente, não posso editar o seu nome).

View Models - Casos de Uso

- Quando não tenho controlo, ou não quero mexer, nas classes do modelo (exemplo: estou a usar a abordagem “Database First”). Por vezes, trabalhamos com código que “não podemos mexer”, logo View Models são adequados para este fim, visto que posso ter controlo total sobre eles.
- Quando faço páginas que necessitam de agregar dados de várias fontes, ter um View Model que guarde esses dados todos num único objeto torna a construção da View mais fácil.
- Em Web APIs, quando quero omitir ou modificar os dados disponibilizados, por questões de segurança, desempenho, ou conveniência.

View Models - Vantagens e Desvantagens

Vantagem: Torna a criação de Views com múltiplos dados mais fácil (não tenho que usar a ViewBag)

Vantagem: Ajuda-me a prevenir ataques de Overposting, porque me obriga a passar manualmente os dados para a base de dados (para as classes do modelo).

Vantagem: Permite-me construir validações mais complexas, sem estar a “poluir” o meu modelo ou controllers. As views ficam também mais resistentes a alterações do modelo.

(Grande) Desvantagem: Escrevo mais código, por vezes duplicado. Herança não ajuda muito aqui. Ferramentas como o [Automapper](#) são por vezes usadas para mitigar isto.

Secções Razor

MVC

Secções Razor

O Razor é dos poucos motores de templating que permite usar secções para controlar onde colocar partes dos conteúdos de uma View.

Uma view de Layout pode definir secções através do [`@RenderSection\(\)`](#). Cada secção tem um nome, e pode ser opcional ou obrigatória (required).

Views que usam um Layout podem usar isto para colocar partes dos seus conteúdos fora da zona do `@RenderBody()` (ex: [scripts antes do fim do body](#), styles no head, modais do Bootstrap)

Nota: A não ser que se use “Layout = ...” numa View, o valor por defeito está definido no [ViewStart.cshtml](#) (é usado por todas as views).

Partial Views

MVC

Partial Views

Partial Views são Views. A única diferença de uma Partial View para uma View normal é que **layouts e secções não estão disponíveis**. Qualquer View (**excepto views com @RenderBody()**) pode ser utilizada como Partial View.

Partial Views são usadas quando queremos **partilhar código** HTML entre views. Servem como *componentes*; peças de Lego.

Numa view posso invocar uma Partial View através do “`@Html.Partial("NomeDaView")`”.

Também podem ser usadas em Controllers; em vez de “return View()” faço “return PartialView()”.

Partial Views - Use Cases

Se quiser partilhar código entre duas views (ex: um formulário), criaria uma Partial View com os campos do formulário comuns.

Quando o código de uma View está a ficar muito grande, usar Partial Views pode ser útil para organizar o código e torná-lo mais legível.

Se quisesse retornar HTML para usar com AJAX, usaria uma Partial View devolvida de um Controller, e usaria \$.ajax para dinamicamente “enxertar” o HTML na página onde fosse necessário.

Namespaces e Views

MVC

Namespaces e Views

O “[Web.config](#)” que está dentro da pasta das Views é usado para configurar as Views de Razor dentro dessa pasta.

Dentro deste ficheiro, existe uma secção que pode ser usada para adicionar [namespaces que queremos que sejam importados automaticamente](#) em todas as Views.

A vantagem de fazer isto é que não temos que dizer o [nome completo da classe](#). Alternativamente, podia-se usar um “@using” (como em C#).

Nota: Modificar este ficheiro implica **recompilar o projeto**, e por vezes, fechar e reabrir as Views. Se isso não funcionar, deve-se fechar e reabrir o Visual Studio.