RESTful API Design

Tímto (netradičně dlouhým) článkem bych se rád zamyslel nad **architekturou webových aplikačních rozhraní**. Při vývoji mého prvního Web API jsem si lámal hlavu s celou řadou otázek a zpětně jsem zjistil, že ne všechno, co jsem vytvořil je ideální. Proto jsem začal psát silně primitivní API, které se snaží na jednoduché ukázce řešit i atypické problémy, které při vývoji vyvstávají. Průběžně jsem si poznamenal na padesát otázek, které jsem průběžně řešil až jsem došel ke konečnému designu API. V této sérii popíšu můj pohled na vývoj API a vzhledem k tomu, že celá řada bodů je logicky sporných, přínos celého tohoto řešení tak je především v ucelenosti.

- Článek v PDF
- Článek v HTML

Projekt je dále k nahlednutí na codeplex, kde se budu snažit průběžně design vylepšovat.

- Web API Design na Codeplex

HTTP protokol

Začal bych samotným HTTP protokolem, který v API použijeme pro příjem RequestMessages a následné vracení ResponseMessages. Při návrhu API jsem tuto kapitolu řešil průběžně, nicméně logicky se jedná o to nejpodstatnější, protože HTTP protokol slouží k samotné komunikaci. Krátce se zde zastavím a popíšu především pár myšlenek, které usnadní některá rozhodování v průběhu vývoje.

Princip komunikace

Pokud konzument pošle nějaký požadavek, bude se jednat o některou z HTTP metod (GET, POST, PUT, PATCH...). Pošle jí na určitý endpoint a bude očekávat odpověď. Jednak ho bude zajímat výsledek jako takový (**HTTP Status Code**) a jednak bude **požadovat nějaký resource**. V případě GET requestu dostane buď single object nebo kolekci objektů. Teoreticky by mohl dostat i unifikovanou error response. Podobně je tomu v případě POST, PUT, PATCH, kde by bylo vhodné vracet v případě chyby podrobnou error response (například při nevalidním modelu).

HATEOAS

Pro implementaci celého řešení je vhodné používat HATEOAS. Myšlenkou je ve své podstatě odekorování vrácených objektů o speciální **vlastnosti odkazující na související zdroje**. HATEOAS pak umožňují i stránkování kolekcemi. V API obvykle tedy řeším

1. vrácení jednoho objektu -> **dědím** ApiEntity
2. vrácení kolekce objektů -> **používám** ApiCollection
3. vrácení kolekce výsledků -> **vracím** ApiBulkResult
4. vrácení chybové zprávy -> **vracím** ErrorModel

Link.cs

Třída `Link` je základním kamenem HATEOAS. Obsahuje odkaz na určitý zdroj. Původně jsem si myslel, že je důležité aby obsahovala i `Method` atribut ale ukázalo se, že je ve většině případů zbytečný.

```
public class Link
{
    public string Href {get; set;}
    public string Rel {get; set;}
}
```

ApiEntity.cs

Třída `ApiEntity` obaluje všechny entity a umožňuje tak připojit kolekci odkazů na související zdroje.

```
public abstract class ApiEntity
{
    public List<Link> Links {get; set;}
}
```

ApiCollection.cs

Třída `ApiCollection` slouží k odekorování kolekce vrácených dat informacemi pro stránkování. I tato třída dědí `ApiEntity` a obsahuje tak

související odkazy na zdroje.

```
public class ApiCollection<T> : ApiEntity
{
    public List<T> Data {get; set;}
    public int? TotalItems { get; set; }
    public int? TotalPages { get; set; }
}
```

ApiBulkResult.cs

Poslední objekt pro HATEOAS je `ApiBulkResult`, který je odpovědí na hromadný požadavek. Obsahuje kolekci HTTP status kódů a k nim Id primárních klíčů.

```
public class ApiBulkResult
{
    public Dictionary<object, HttpStatusCode> Results {get; set;}
}
```

Bulk Response

Některé endpointy mohou umožňovat **hromadné operace**. Pokud například konzument požaduje DELETE 10 resources, bylo by nedostačující odpovědět jedním status kódem (jeden z resources se například nemusí podařit odstranit). Proto je ideální vrátit kolekci HTTP odpovědí ke každému požadovanému odstranění samostatně. Příkladem budiž objekt `ApiBulkResult` výše.

```
DELETE /api/articles/1,2,3,5

{
    1: 404,
    2: 204,
    3: 404,
    5: 204
}
```

Řešení by mohlo být samozřejmě sofistikovanější a obsahovat i informace o příčině chyby.

Pagination

Jedna z důležitých funkcí je stránkování. **Pro účely API je ideální s touto funkcí počítat už při návrhu repositories.** Po zvážení všech možností se jeví (kvůli přenositelnosti URL, HATEOAS) jako nejideálnější řešení vkládat informace o stránkování do **QueryStringu**, např.:

```
GET     /api/articles?page=2&pagesize=20
```

s tím, že systém má výchozí nastavení na nějakém `BaseControlleru`, které může být přepsané konkrétními controllery.

Čtení Page a PageSize osobně provádím vystavením property `QueryFilter` na `BaseControlleru`.

```
protected QueryFilter QueryFilter
{
    get { return filterQuery ?? (filterQuery = ExtractFilterQueryData()); }
}

private QueryFilter ExtractFilterQueryData()
{
    var requestHelper = new RequestHelper(Request);

    int page = requestHelper.GetValueFromQueryString<int>("page");
    int pageSize = requestHelper.GetValueFromQueryString<int>("pagesize");

    var filter = new QueryFilter
    {
        Page = page != 0 ? page : DefaultPageNumber,
        PageSize = pageSize != 0 ? pageSize : DefaultPageSize
    };
```

```
    return filter;
}
```

Při dotazování na repositáře pak používám výhradně tento queryfiltr:

```
public interface IArticleRepository : IRepository<Article>
{
    PagedList<Article> Get(QueryFilter filter);
    PagedList<Article> GetByAuthorId(QueryFilter filter, int authorId);
}
```

Content negotiation

Jedna z celkem jasných oblastí se týká Content Negotiation. Protože mi přijde čistě logické nechat většinu rozhodování na straně klienta, doporučuji striktně cestu **agent-driven content negotiation**, kdy o obsahu rozhoduje klient. Mezi typické problémy patří **volba formátu** (json, xml) nebo **volba jazyka**.

Language negotiation

I kdyby rozhraní poskytovalo data jen v jednom jazyce (například autor napsal článek jen ve španělštině), dává smysl vracet formátované alespoň formáty datumů nebo například různé servisní informace. Podpora Language negotiation má vždy určitý smysl.

```
Accept-Language: de; q=1.0, en; q=0.5
```

Výše uvedená HTTP hlavička bude preferovat vrácení dat v německém formátu, případně v anglickém s nižší prioritou. Pro podporu ze strany aplikace stačí aktualizovat web.config

```
<globalization uiCulture="auto" culture="auto"/>
```

a následně vracet textové informace pomocí Resource Files,

```
ApiHttpResultMessages.resx     - výchozí EN verze
ApiHttpResultMessages.de.resx  - německá verze
ApiHttpResultMessages.cs.resx  - česká verze
```

například:

```
return Request.CreateErrorResponse(HttpStatusCode.NotFound, ApiHttpResultMessages.NotFound);
```

Pro správné formátování datumů je ideální datové typy `DateTime` převádět na `string`.

Response data format

Protože se dnes stále hojně využívají dva formáty (JSON / XML), je vhodné povolit vracení dat v požadovaném formátu ze strany agenta. Ten se na formát dotazuje pomocí HTTP hlavičky

```
Accept: text/json;
```

a pro správné fungování ve Web API aplikaci je důležité nakonfigurovat tzv. formatters. Běžně jsou tyto formatters nastavené a obvykle se hodí pouze donastavit serializaci XML:

O nastavení XML formatteru jsem psal v nedávném článku Web API XML Serializer.

Další možnosti content negotiation

Dále je možné použít HTTP hlavičky pro nastavení požadované znakové sady aj. Možností je celá řada a uvedené příklady jsou spíše ilustrativní. Více o content negatiation se můžete dočíst zde.

Range

Další užitečnou HTTP hlavičkou je Range. Její použití je široké a teoreticky by se dala použít například i pro stránkování. To však nedoporučuji,

protože stránkování je vhodnější umístit přímo do URL kvůli přenositelnosti a možnosti procházet API.

Reálné využití Range je pro určení offsetu a limitu přijímaných dat u různých **streaming APIs**. Pokud například potřebujeme streamovat video nebo soubor od určitého místa, můžeme to udělat pomocí Range s tím, že limitem si definujeme velikost bufferu (zobecněně).

```
GET /api/videos/25
Range: bytes=0-500
```

URI Design

Velká kapitola je vzhled URL :) Ukážu teď všechny možné kombinace, které snad mohou nastat a popíšu, proč je řeším tak, jak je řeším. Ještě předtím ale zmíním dvě moje myšlenky související s **pluralizací a umístěním vnořených kolekcí na správné controllery**.

Pluralizace

V naprosté většině případů dává smysl pluralizovat názvy resources v URI. Jsou ale situace, kdy je vhodnější pluralizaci opustit a využít jednotné číslo pro **zdůraznění vztahu 1:1**. Nejlepšími příklady jsou konfigurační informace.

```
GET /api/configuration
```

Předpokladem je samozřejmě existence jediné konfigurace. Obvykle pak nepoužíváme nad endpointem metody POST a DELETE. Pouze data čteme a aktualizujeme (GET, PUT/PATCH).

Teoreticky by měla podobně vypadat i URL, která vrací právě jediného autora článku (pokud článek má vždy jednoho autora).

```
GET /api/articles/25/author
```

Obvykle ale tyto endpointy vůbec nepotřebujeme a ani je nedoporučuji používat. Jsem spíše toho názoru, že všechny související objekty by měli být dostupné přímo na hlavní kolekci. Více o tomto bodě píšu níže. Vystačíme si tedy pouze se dvěma endpointy:

```
GET /api/articles/25  - přímo obsahuje informace o autorovi v response
GET /api/authors/2    - pokud chci jen autora, jdu přímo na resource authors
```

Umístění nested collections na controllerech

Uvažujme endpoint:

```
GET /api/authors/2/articles
```

Kam umístíme metodu `Get()`? `AuthorsController` nebo `ArticlesController`? Neexistuje správná odpověď ale jsem toho názoru, že správným místem je `ArticlesController`. **URI bude v obou případech stejná.** Jediný rozdíl je ale v tom, že endpoint vrací kolekci článků (dle id autora) a dává tak smysl, aby bylo zodpovědností `ArticlesController`u tento požadavek vyřídit. Kdybychom umístili Get() metodu na AuthorsController, okamžitě bychom potřebovali připojovat repositář `ArticleRepository`. A to je zbytečné. Koneckonců metoda PagedList

`GetByAuthorId(QueryFilter filter, int authorId)` je také umístěná na `ArticleRepository`.

Při generování dokumentace pomocí **ApiExploreru** pak tento endpoint najdeme pod sekcí Articles. A to je správně. Chceme totiž kolekci článků (ať už filtrovanou nebo vlastněnou kýmkoliv). Nakonec je zde celá řada "malých důvodů". Dává například smysl, že výchozí počet vrácených výsledků (PageSize) je společný pro jeden typ kolekcí, tedy v tomto případě pro články. `ArticlesController` tedy vždy vrátí stejný počet výsledků bez ohledu na to, zda se jedná o obecnou kolekci nebo o kolekci článků autora.

URI design pro HTTP metody

Teď už se pojďme podívat na jednotlivé HTTP metody a design URI podle toho, co mají dělat.

GET single object

Tady není snad co řešit:

```
GET /api/articles/2
```

nebo ano? Vlastně jsem uvažoval nad podobnou klíče. Došel jsem nakonec k závěru, že nejlepší je klíč primární ale prakticky nic nebrání tomu, aby jím byl i unikátní sloupec. Tedy osobně vidím URI

```
GET /api/authors/6
GET /api/authors/mholec
```

jako rovnocenné. Nehodí se ale už pro tvorbu vazebních tabulek M:N, kde mi přijde rozumnější pracovat s reálnými FK (viz. dále).

GET collection

Vlastně ani vrácení kolekce standardně není žádná magie.

```
GET /api/articles
```

Otázkou by mohlo být, jak seřadit výsledky. Standardně bude existovat výchozí řazení (latest first = ID / Created) s tím, že by mohl být dostupný filtr pro výběr konkrétního atributu (sortby). **Filtry a řazení budu řešit v dalším článku.**

GET nested collection

Nested collections (vnořené kolekce) používáme pokud chceme vrátit kolekce, které jsou vlastněny nějakým resourcem. Zatímco příklad výše vrátil všechny články bez ohledu na autora, můžeme požadovat vrácení kolekce článků striktně dle autora:

```
GET /api/authors/2/articles
```

Kdybych chtěl ale ještě teoreticky štítky nějakého konkrétního článku, už by se mělo jednat o nezávislou nested collection:

```
GET /api/articles/25/tags
```

Chybou by bylo zpřístupnit tuto nested collections takto:

```
GET /api/authors/2/articles/25/tags
```

protože konzument by musel znát za všech okolností ID autora (což je nežádoucí).

POST single object

Odeslání jednoho objektu je podobně jednoduché jako jeho získání:

```
POST /api/articles
```

POST collection

Odeslání kolekce objektů mi osobně přijde jako nestandardní a nenašel jsem žádné využití takové operace. Tedy až na hromadné operace související s vazebními tabulkami. Řešením by byl stejný endpoint jako pro single object ale s jiným argumentem typu kolekce a response typu `ApiBulkResult`.

POST nested single object

Odeslání nested objektu je z mého pohledu celkem užitečné ale zapeklitější než se může zdát. Jako správné pokládám:

```
př. 1: POST /api/authors/2/articles
```

na rozdíl od méně vhodného

```
př. 2: POST /api/articles
```

kde by součástí body byl body json/xml upřesňující vazbu AuthorId, tedy:

```
{
    AuthorId : 2,
    Title : "Some title here"
}
```

Toto řešení je "špatné" ze dvou důvodů

- v případě první ukázky existují dvě místa určující vazbu - URL AuthorId a body AuthorId (možnost konfliktu)
- z podstaty GET metod první endpoint evokuje "článek napsal autor č. 2", zatímco druhý endpoint evokuje "článek NEMÁ ŽÁDNÉHO autora".

Komplikace to však může být pro front-end, protože jedna operace nyní může teoreticky probíhat na dvou URL v závislosti na tom, zda článek bude či nebude mít autora. Přesto dle mého názoru

vhodná implementace je:

- cizí klíče součástí POUZE url, nikoliv request body
- POST do nested collection se připojí k řídícímu objektu (př. 1)
- POST do běžného resource je nezávislý (př. 2)

Všechno může mít i praktické výjimky. Například pokud by byl uživatel indentifikován pomocí autentizace a pak by se vazba tvořila systémově (bez URL). Tedy v takovém případě by `POST /api/articles` automaticky navázal resource na autentifikovaného uživatele. Chybou by byl endpoint ve stylu:

```
POST    /api/authors/articles
```

protože ten by říkal, vlož článek, který napsali všichni autoři.

Vazební tabulky M:N

Nad vazebníma tabulkama jsem si lámal hlavu snad nejvíce. Nakonec jsem našel celkem logické řešení, které jsem si doplnil o pravidla, která zatím celkem fungují. Předpokladem je rozdělení vazebních tabulek na M:N a M:1:N.

V případě vazby M:N je situace jednoduchá, protože taková vazba definuje složený PK z dvou cizích klíčů. Pokud budeme uvažovat například situaci "Produkt má více kategorií a v kategorii je více produktů", pak jsem dospěl k následujícím endpointům:

```
POST    /api/products/2/categories/6
DELETE  /api/products/2/categories/6
```

Zatímco první endpoint (POST) vytváří vazbu, druhý (DELETE) vazbu ruší. Odeslání POST requestu bez body je v tomto případě v pořádku. Řešil jsem také otázku, zda by vazba neměla být opačná, tedy:

```
POST    /api/categories/6/products/2
DELETE  /api/categories/6/products/2
```

a teoreticky by mohla. Dokonce by teoreticky mohli **koexistovat oba endpointy**. Nicméně udržovat dva endpointy, které dělají totéž je code smell. Kam tedy vazbu dát? Podle mého názoru na tu stranu, která je zodpovědnější za život objektů. V tomto případě se přikláním k produktům, protože samostatná existence kategorií bez produktů v reálném světě je poněkud irelevantní. Lépe si to lze představit na příklad "autoři píšou knihy a knihy mohou být napsány více autory". Kniha bez autorů je opět irelevantní (sama se nenapíše). Je tu určitá kompozice. Ale autoři existovat mohou, i když knihu ještě žádnou nemají. **Proto bych se přikláněl k tvorbě endpointů směrem od řídících objektů**.

Také si myslím, že v případě vazeb M:N NENÍ DOBRÉ umožnit tvorbu vazeb za běhu přes URL. Jednoduše proto, že tato tvorba je nedostačující. Například odesláním dotazu:

```
POST    /api/categories/2/products
```

by teoreticky měl vzniknout produkt, který se zařadí do kategorie s ID 2. V případě M:N ale můžeme potřebovat zařadit produkt do více než jedné kategorie. A pak nám jednoduše tento endpoint nestačí a stejně nezbyde než použít další (jeden z těch výše uvedených) nebo si vytvořit **endpoint specializovaný na hromadné operace**.

Vazební tabulky M:1:N

V případě vazebních tabulek M:1:N je situace trochu jiná, protože vazební tabulka obvykle nese nějakou další informaci. Uvažujme stejný příklad s autory a knihami:

```
AuthorsBooks
----------------------------
```

```
PK   int        AuthorBookId
FK   int        AuthorId
FK   int        BookId
     int        Worth
```

V této tabulce identifikujeme vztahy na základě umělého PK a připojujeme informaci o zásluhách autora na psaní knihy. Je tedy zřejmé, že budeme potřebovat vazbu i aktualizovat (nejen vytvářet a rušit). V takovém případě se přikláním k tvorbě standardního endpointu a data posílat v body requestu:

```
POST    /api/authorsbooks
PUT     /api/authorsbooks/5
DELETE  /api/authorsbooks/5
```

Vazební tabulky pro hromadné změny

Poslední případ je, když někdo na frontendu zakřížkuje který produkt patří do jakých kategorií a následně odešle celou tuto kolekci informací. Vycházel bych pak ze stejného řešení jako v případě M:1:N s tím, že v případě vazeb M:N bych si vytvořil také speciální "vazební" endpoint. Data bych poté odesílal jako array s vazbami a odpovídal bych pomocí objektu ApiBulkResult.

```
PUT     /api/productcategories/2
```

Tento endpoint by pak uměl provést změny souvisejícími s kategoriemi nad produktem ID = 2.

DELETE single object

Odstraňování jednoho objektu je celkem přímočaré a nemyslím, že by tu měl být nějaký konflikt. Delete by měl být umístěn co nejblíže k požadovanému resource:

```
DELETE  /api/articles/25
```

tedy nemá smysl odstraňovat single object jako nested:

```
DELETE  /api/authors/2/articles/25
```

protože konzumentovi může být jedno, komu daný článek patří. Autorizace požadavku by měla proběhnout nějak sofistikovaněji (nikoliv na základě URL).

DELETE many objects

Pokud chceme odstranit více objektů, dává mi smysl takový požadavek řídit celý skrze URL.

```
DELETE  /api/articles/2,5,6,7,11
```

Ids objektů se dá z requestu získat pomocí vlastního filtru:

```
[Route("articles/{articleIds}")]
[HttpDelete]
[ArrayInputParameter("articleIds")]
public HttpResponseMessage Delete(int[] articleIds) {....}
```

Definice filtru může vypadat takto:

```
public class ArrayInputParameterAttribute : ActionFilterAttribute
{
    private readonly string parameterName;
    private const int MaxParameters = 10;

    public ArrayInputParameterAttribute(string parameterName)
    {
        this.parameterName = parameterName;
        Separator = ',';
    }
```

```
    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        if (actionContext.ActionArguments.ContainsKey(parameterName))
        {
            string parameters = string.Empty;

            if (actionContext.ControllerContext.RouteData.Values.ContainsKey(parameterName))
                parameters = (string)actionContext.ControllerContext.RouteData.Values[parameterName];

            else if (actionContext.ControllerContext.Request.RequestUri.ParseQueryString()[parameterName] != null)
                parameters = actionContext.ControllerContext.Request.RequestUri.ParseQueryString()[parameterName];

            var paramValues = parameters.Split(Separator).Select(int.Parse).ToArray();

            if (paramValues.Count() > MaxParameters)
            {
                throw new HttpResponseException(actionContext.Request.CreateCustomResponse(HttpStatusCode.BadRequest, "Too many pa
            }

            actionContext.ActionArguments[parameterName] = paramValues;
        }
    }

    public char Separator { get; set; }
}
```

Součástí filtru je i omezení na maximální počet parametrů (kvůli maximální délce URL). V tomto případě je omezení nastaveno na 10 prvků v poli. Response by měla být typu `ApiBulkResult`, která jasně oznámí jak byly vyřízeny požadavky nad jednotlivými resources.

DELETE all objects

Metoda DELETE all moc časté využití nemá. Kromě toho je celkem nebezpečná. Každopádně někdy se může hodit (například. odstranit všechny štítky článku). Endpointy jsou celkem intuitivní:

```
DELETE  api/tags
DELETE  api/articles/5/tags
```

V prvním případě se odstraní zcela všechny štítky, v případě druhém pak jen všechny štítky článku s ID 5. Response by měla být opět typu `ApiBulkResult`.

PATCH / PUT rozdíl

Když budu vycházet z akademické definice, PUT aktualizuje celý resource, zatímco PATCH pouze jeho část. Zřejmě si k této definici můžeme domyslet "kromě primárních klíčů", protože v komplexních systémech bychom pak nikdy PUT nepoužili a z principu by se jednalo vždy a pouze o PATCH. Při vývoji mého API jsem tak došel k tomu, že veškeré updaty mám implementovány jako PUT metody. **Kdy tedy a hlavně jak implementovat PATCH?** Pro PATCH jsem našel dokonce dvojí použití (z hlediska podstaty).

Defacto PATCH může být užitečná servisní metoda a nebo může suplovat PUT metodu v momentě, kdy jsou některé properties povinné. Uvažujme příklad:

```
public class Author
{
    public string Firstname {get; set;}

    [Required]
    public string Email {get; set;}
}
```

V případě metody PUT metody není co řešit. Konzument musí poslat kompletní validní objekt i v případě, že chce aktualizovat jen Firstname. Tedy včetně povinného emailu. V opačném případě se mu vrátí BadRequest.

V případě metody PATCH je ale přípustné, že konzument pošle pouze Firstname. Action metoda v takovém případě může použít stejný model ale jednoduše neprovede validaci celého modelu ale každého přijatého atributu samostatně. Protože **[FromBody] může být uvedeno jen jednou**,

je možné data nabindovat například na `dynamic`:

```
public HttpResponseMessage Patch([FromBody]dynamic model)
{
    if(model.firstname != null){...}
}
```

Response Object

Jedna snad z nejrozporuplnějších otázek se týká toho, co vlastně uživateli vracet za data. Představte si následující URIs a přemýšlejte se mnou:

- při GET products/1 - mám vrátit související objekty jako např.: eshop?
- při GET products/1 - mám vrátit související kolekce jako např.: categories?
- při GET products/1 - mám někdy vrátit jen část dat (například při výpisu kolekcí) zatímco na detailu všechno?
- při PUT products/1 - mám aktualizovat související objekt, jako např.: eshop?

I toto dilema jsem nakonec rozklíčoval a uzavřel s tím, že se **nejedná o jediné správné řešení**. Přesto si myslím, že můj přístup celkově zapadá do celého konceptu.

GET vrátí vždy kompletní objekt bez cizích klíčů včetně souvisejících objektů

Pokud budeme uvažovat příklady:

```
GET     /api/articles
GET     /api/articles/5
```

pak by měl být vždy vrácet **kompletní article objekt**. Bez ohledu na to, zda se jedná o požadavek na jeden článek nebo celou kolekci. Každý article by měl být kompletní. Dále by měl obsahovat rovnou i související objekty a **neměl by obsahovat cizí klíče**. Teď zkusím popsat myšlenky, proč zrovna takto:

- je nežádoucí, aby NULL měla dvojí význam... pokud je něco NULL, je to skutečně NULL z podstaty věci a ne proto, že "na tomto endpointu to prostě neplním" = co je na modelu, to naplním
- největší režie je sestavení request message a přenos po síti... vzhledem k velikostem běžných webů (často MB) je zbytečné uvažovat nad tím, zda endpoint vrátí 100 bajtů nebo 3000 bajtů. Proto je lepší rovnou připojovat související objekty a získat tak plnohodnotný objekt. Neplatí to ale o kolekcích (viz. další bod).
- z podstaty důvodu B máme **na souvisejícím objektu vždy primární klíč** a tudíž je zbytečné (a nežádoucí) aby se tento klíč opakoval znovu v podobě cizího klíče na hlavním resource.

Příklad:

```
{
    articleId: 4,
    authorId: 2, // zbytecne, to same je na related objektu
    title: "Muj clanek",
    author: {
        authorId: 2,
        firstName: "Mirek"
    }
}
```

GET nevrací nikdy související kolekce (pouze na explicitní vyžádání)

Přestože jsem zastáncem vracení kompletních objektů včetně souvisejících resources, v případě kolekcí je situace jiná.

- pokud by resource obsahoval kolekci, ta by obsahovala i odkaz na resource a zpětně... zkrátka došlo by k zacyklení
- kolekce resource může být značně velká (např.: eshop může mít 10000 produktů) a pak vracení tolika objektů je přes síť nemyslitelné
- pokud bychom vrátili kolekci omezenou na počet záznamů, je otázka JAK provést omezení - zobrazit nové nebo oblíbené nebo nejčtenější nebo náhodné (nedeterminističnost)

A to je důvod, proč zde figurují HATEOAS a kolekce je ideální vůbec nevracet. Existují ale výjimky:

- simplifikace v podobě dictionary

- simplifikace v podobě pretty simple objektů

a předpokladem pro obě možnosti je, že taková kolekce je "rozumně omezená".

Příklady:

1. článek může mít štítky
2. produkt může mít značky
3. kniha může mít autory (knihu obvykle nepíše 1000 autorů)

Vrácená data mohou vypadat například takto:

```
title : "Muj clanek",
tags: {
    254: "Pevne disky",
    269: "Pocitacove hry"
},
```

PUT/PATCH aktualizuje vždy jen hlavní resource

Uvažoval jsem nad aktualizací resources a opět kvůli deterministočnosti jsem došel k závěru, **že PUT/PATCH metody by měly aktualizovat pouze "svůj" hlavní resource**. Související objekty by měly zůstat nedotčeny. Právě pro tento účel a pro jasné vymezení toho, co je určené k aktualizaci jsem také došel k závěru, že je lepší mít** speciální model pro request message a speciální model pro response message**. V systému tak existují tři různé modely. Příkladem může být entita reprezentující autora:

První entita je na datové vrstvě. Je to dalo by se říci POCO objekt, který slouží pro ORM. Vztahy jsou dále popsány pomocí Fluent Api ale stejně tak by bylo relevantní použít datové anotace.

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Bio { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}
```

Další entita slouží pro vrácení dat z API. Dědí třídu ApiEntity, která zajišťuje HATEOAS a je oproštěná o data související s objektově relačním mapováním. Kolekce tu nenajdeme a nebo jsou velmi simplifikované (jak jsem popisoval výše).

```
public class AuthorModel : ApiEntity
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Bio { get; set; }
}
```

Poslední třídou je model pro request message. Ten je obvykle ze všech nejjednodušší. Už neobsahuje klíč (protože klíče jsou součástí URL) a duplicita je nežádoucí. Co má ale každý InputModel navíc jsou validační pravidla (ať už Data Annotations nebo realizace IValidatableObject.

```
public class AuthorInputModel
{
    [Required(ErrorMessageResourceName = "Required", ErrorMessageResourceType = (typeof(ApiValidationMessages)))]
    public string Firstname { get; set; }

    [Required(ErrorMessageResourceName = "Required", ErrorMessageResourceType = (typeof(ApiValidationMessages)))]
    public string Lastname { get; set; }
    public string Bio { get; set; }
}
```

Výhodou je i možnost provádět rozdílné validace ze strany klienta na server a na datové vrstvě.

Error responses

Už jsem nastínil jaké je ideální vracet response objekty. V případě výskytu chyby v API se hodí uživatele informovat poněkud přesněji o tom, co se vlastně stalo. Pro tento účel používám jednoduchý `ErrorModel`.

```
public class ErrorModel
{
    public ErrorModel(HttpStatusCode statusCode, string message)
    {
        StatusCode = (int)statusCode;
        Message = message;
        ValidationErrors = new Dictionary<string, ModelErrorCollection>();
    }

    public ErrorModel(HttpStatusCode statusCode)
    {
        StatusCode = (int)statusCode;
        ValidationErrors = new Dictionary<string, ModelErrorCollection>();
    }

    public string Message { get; set; }
    public int StatusCode { get; set; }
    public Dictionary<string, ModelErrorCollection> ValidationErrors { get; set; }
    public Exception Exception { get; set; }
}
```

Error message se dá vrátit z API i bez této obstrukce na Controllerech například takto:

```
return Request.CreateErrorResponse(HttpStatusCode.NotFound, ModelState);
```

Nevýhodou je, že serializovaná response nemá takový tvar, jak by zrovna vývojář chtěl a hlavní message není lokalizovaná. Já proto používám vlastní extension metodu:

```
return Request.CreateCustomResponse(HttpStatusCode.NotFound, ModelState);
```

která vypadá na první pohled stejně ale vrací můj vlastní ErrorModel, který je lokalizovaný (language negotiation) a validation errors obsahují odkazy přímo názvů properties (např.: Title, Name, SomeId) místo defaultních obskurit (model.Title, model.Name, etc.).

Status Codes

Co se týče stavových kódů, obvykle zde není moc problém. Důležité je rozdělení chyb 4xx VS. 5xx a dodržení základních pravidel.

- vznikne-li chyba, za kterou je odpovědný konzument / klient, pak vracíme 4xx
- vznikne-li chyba, za kterou je zodpovědný server, pak se vrací některý 5xx
- pokud je všechno OK, vrací se některý 2xx kód

Celkem tvrdý oříšek vyvolávající diskuse může být ale** požadavek na odstranění zdroje** a odpovědi typu 204 (No Content) vs 202 (Accepted).

Každý požadavek na odstranění resource je prapůvodně 202 (pokud je přijat) a následně probíhá zpracování. V momentě, kdy je request vyřízen a odeslán zpět konzumentovi, je potřeba rozhodnout se, zda mu oznámíme 204 (tedy vyřízeno a žádný obsah již neexistuje) nebo 202. Můj závěr je takový, že požadavek je vyřízen (204) pokaždé:

- pokud došlo k odstranění hlavního požadovaného resource
- pokud by okamžitě po response poslal konzument další DELETE / GET na stejný response, pak by nutně dostal 404 (NotFound)

A je podle mého názoru jedno, zda

- došlo či nedošlo k dalším souvisejícím vnitřním procesům,
- došlo či nedošlo například k asynchronním I/O operacím,

jelikož to už je v ten moment klintovi jedno. Onen hlavní zdroj byl odstraněn a i kdyby se něco dalšího nepovedlo, tak se to nedozví.

Zkrátka **smysl vracet 202 má jen**:

- pokud by okamžitě po response poslal konzument GET na stejný response, pak by tento resource stále ještě možná získal,
- pokud odstranění nebylo provedeno a bylo zařazeno do nějaké Queue ke zpracování,
- pokud odstranění nebylo provedeno a čeká na vyřízení jiného požadavku,
- pokud odstranění nebylo provedeno a čeká na schválení jinou autoritou

Zároveň to všechno jsou výsledky s nezřejmým koncem a musí být vráceno 202 (Accepted, přijato s nejasnou budoucností). Chybou by bylo vrátit jiný stavový kód evokující, že něco se definitivně nezdařilo (např.: Conflicted).

Závěr

V článku jsem popsal několik mých myšlenek souvisejících s návrhem aplikačního rozhraní pomocí ASP.NET Web API. Závěrem dodávám, že toto řešení není zdaleka jediné a zdaleka né jediné správné. Každé aplikační rozhraní je velmi unikátní a ne vždy je potřeba implementovat všechny endpointy, podporovat hateoas nebo složitě pracovat se třemi druhy modelů v aplikaci. Zkrátka dobrý design je obvykle o tom najít takové řešení, které stojí optimální vývojářské úsilí a splní požadavky klientů. Zasadit pak všechny požadavky do funkčního konceptu a implementovat je nemusí být jednoduché. Tento článek ukazuje celkem komplexní design, který již většinu typických problémů řeší.