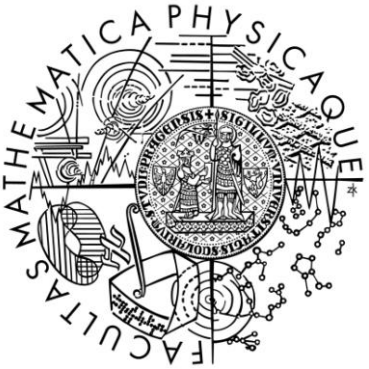
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Václav Nádraský

Redakční systém s podporou dynamicky generovaného obsahu

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Pavel Ježek

Studijní program: Informatika

Studijní obor: Softwarové systémy

Praha rok 2012

Na tomto místě bych rád poděkoval svému vedoucímu práce Mgr. Pavlu Ježkovi, který mi byl vždy k dispozici a poskytnul velmi cenné rady a zkušenosti.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 12. 4. 2012 podpis

Název práce: Redakční systém s podporou dynamicky generovaného obsahu

Autor: Václav Nádraský

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Pavel Ježek, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Předmětem práce je návrh a vytvoření redakčního systému, který je snadno rozšiřitelný a umožňuje vytvářet webové aplikace z předem připravených komponent, které je možné libovolně umisťovat do obsahu stránek. Tyto komponenty mohou obsahovat i složitou aplikační logiku, která je ale nezávislá na rozložení ovládacích prvků na stránce. Redakční systém dovoluje tvořit obsah obsahující libovolné informace z databáze bez nutnosti programovat nebo psát SQL dotazy.

Klíčová slova: redakční systém, CMS, dynamický obsah

Title: CMS Supporting Dynamically Generated Content

Author: Václav Nádraský

Department: Department of Software Engineering

Supervisor: Mgr. Pavel Ježek, Department of Distributed and Dependable Systems

Abstract: The topic of this thesis covers design and development of content management system which is easily extensible. It allows creating websites out of components which can be placed at any place in a web site. These components can contain a complex application logic which is independent of a layout of user controls. Content management system also contains a component allowing to place any data from any database into web site content without need to program or to create SQL queries.

Keywords: content management system, dynamic content

Obsah

1 Úvod

Pod pojmem redakční systém si většina internetových uživatelů představuje softwarový nástroj, který umožňuje snadnou tvorbu internetového obsahu bez větší nutnosti umět programovat či dokonce umět značkovací jazyk HTML. Softwarů takového typu existuje celá řada. Většina z nich se ale zaměřuje pouze na správu statického obsahu ve formě článků či textových příspěvků. Takové systémy pak nelze žádným způsobem využít pro situace, kdy například potřebujeme postavit webový portál či aplikaci, která by byla napojena na nějakou databázi, ze které by se měly vypisovat různé informace. Typickým příkladem jsou například malé firmy, které si v počátcích svého podnikání nechaly vyvinout nějaký jednoduchý informační systém, ve kterém by si evidovaly vše potřebné k jejich podnikání. Jejich webové prezentace se pak většinou skládaly pouze ze statických stránek nebo byly v lepším případě vedené v některém z redakčních systémů. V dnešní době je však velkým trendem umožňovat svým zákazníkům čím dál více služeb pomocí webového rozhraní s co nejaktuálnějšími informacemi. Tohoto však nelze docílit jinak než přímým napojením redakčního systému či webové aplikace na databázi využívané interním informačním systémem, ve kterém se právě takto aktuální informace nacházejí. Cílem této práce je tedy vytvoření redakčního systému, který by umožňoval podobně jednoduše tvorbu statického obsahu stejně jako většina existujících redakčních systémů, ale s tím rozdílem že by zároveň poskytoval jednoduché prostředky, jak do obsahu webu dostat informace uložené právě v některé z databází využívaných jiným informačním systémem. Takovýto redakční systém by se dal také velmi dobře využít v situacích, kdy například vyvíjíme nějaký obecně použitelný informační systém, který chceme nabízet více zákazníkům. Jako příklad lze uvést třeba rezervační systém pro cestovní kanceláře, který umožňuje jednoduchou správu zájezdů, nakoupených kapacit v hotelech a jejich on-line prodej na webu. U takovéhoto systému se při nasazení u zákazníka (tedy cestovní kanceláře) předpokládá, že jedním z jeho důležitých požadavků bude právě webový portál pro prodej zájezdů, který se ale typicky má odlišovat od konkurence. Redakční systém, který by umožnil rychlou tvorbu webu, jehož hlavním obsahem mají být právě informace tvořené z dat uchovávaných v databázi rezervačního systému, by velmi výrazně zjednodušil (a většinou tak i zlevnil) uvedení takovéhoto webu do provozu. Konkurenceschopnost by se ještě výrazně zvedla, pokud by tento systém umožňoval vytváření dynamického obsahu bez nutnosti programování. Tvorba webu by se tak omezila pouze na vytvoření samotného obsahu pomocí HTML a používání dynamických komponent pro výpisy a vyhledávání zájezdů či jejich rezervace. Tento přístup mimo to i snižuje prostor pro zanesení nových chyb v důsledku programování nového

kódu. Výsledkem použití zamýšleného redakčního systému by tak byla rychle postavená a stabilní aplikace.

1.1 Existující řešení a jejich nevýhody

Při prozkoumávání trhu s redakčními systémy je jasné, že redakčních systémů existuje obrovské množství. Jak bylo ale řečeno v úvodu, drtivá většina z nich dokáže pracovat jen s jednoduchým obsahem, který je ve své podstatě statickým, nebo naopak jsou zaměřeny jen na konkrétní typ webových aplikací. Příkladem mohou být redakční systémy pro internetové obchody, které ale vyžadují, aby prodávané produkty byly udržovány v jejich databázi. Existují ale i výjimky, které se velmi přibližují požadavkům, které byly nastíněny v úvodu – tedy propojení redakčního systému s databází jiného informačního systému. Asi nejznámější a nejpropracovanějším systémem je systém Kentico CMS (1). Tento systém nabízí téměř všechny funkce jako klasický redakční systém a zároveň umožňuje doprogramovat libovolné rozšíření. Jedná se o komplexní a dobře propracované řešení, které se velmi dobře hodí pro situaci popsanou v úvodu, kdy máme informační systém a chceme umožnit libovolné propojení s existující databází a umožnit tak vytvářet dynamický obsah webu. Kentico CMS je založen na platformě ASP.NET a rozšiřování systému o nové komponenty se provádí hlavně pomocí vytváření nových uživatelských ovládacích prvků. Takovýto uživatelský prvek je pak možné libovolně využívat na stránkách a dosáhnou tak znuvupoužitelnosti v případě, kdy chceme danou komponentu použít i u jiného zákazníka. Velmi zajímavou vlastností systému Kentico je možnost si nadefinovat databázové entity pomocí průvodce, který umí za uživatele automaticky vytvořit příslušnou tabulku v databázi a zároveň i rovnou vytvořit nové komponenty pro výpis entity, její vložení, editaci a smazání. Dovoluje tedy velmi jednoduše a rychle vytvářet jednoduché webové aplikace i uživatelům, kteří nemají například žádnou znalost jazyka SQL. Při bližším seznámením s tímto systémem ale zjistíme, že ač vyniká ve své jednoduchosti, při specifičtějším požadavku jakým je například možnost napojení na existující databázi, musí stejně většinu práce odvést programátor sám. Další nevýhodou je jeho cena, která i v případě nejnižší konfigurace začíná na téměř dvou tisících dolarech. V případě, kdy chceme postavit jednoduchý web, který by jen vypisoval několik málo informací z existující databáze, je tato investice zbytečně vysoká a nemůže se vyplatit.

Výše popsané důvody nás tedy dovedli k přesvědčení, že by bylo dobré vytvořit úplně nový redakční systém, který by byl zaměřen právě na oblasti, které na dnešním trhu chybějí. Implementace komplexního systému jakým je například Kentico a jeho doplnění o chybějící funkcionalitu se však jeví jako velmi náročný úkol přesahující rozsah této práce. Cílem tedy bude vytvořit systém, jenž se zaměřuje pouze na netypickou funkcionalitu, který by byl

zároveň velmi dobře rozšiřitelný, aby se případné chybějící vlastnosti a komponenty daly jednoduše doplnit.

1.2 Napojení na informační systém Magic2G

Celá práce bude navrhována a koncipována tak, aby se výsledný systém dal propojit a zakomponovat do komerčního systému Magic2G (zkráceně M2G) vyvíjený společností MagicWare, s. r. o. Jedná se o rozsáhlý databázový informační systém, jehož relační struktura databáze se skládá z téměř 700 tabulek. Jeho primární zaměření je tvorba a prodej produktů cestovního ruchu – tedy prodej pobytových a poznávacích zájezdů, samostatných hotelů či letenek. Největším benefitem pro jeho uživatele je on-line rezervační systém, který nabízí okamžitou informaci o dostupnosti kapacit požadovaného produktu. Systém se neomezuje jen na tento segment trhu a začíná se postupně nasazovat i v situacích, které s cestovním ruchem mají pramálo společného. Jeho důležitou součástí byl také velmi jednoduchý redakční systém umožňující vytváření celého webového portálu z online administračního rozhraní, které umožňovalo přidávat do webu nové stránky. Obsah stránek byl ukládán v databázi jako HTML, do kterého bylo možné vkládat XML tagy, které při zpracování stránky byly nahrazovány ASP.NET uživatelskými ovládacími prvky (podobný princip jako v systému Kentico). Chování těchto ovládacích prvků bylo možné upravit pomocí klasických XML atributů, které byly při zpracování obsahu nastaveny do příslušných C# vlastností daného prvku. Systém tak umožňoval jednoduché doplnění obsahu webu o dynamické části, které ale vždy musel vytvořit některý z programátorů. Jednou z typických komponent, byl i dynamický objekt (DO)[1] s názvem `mw:ProductList`[2], který zastával funkci pro výpis seznamu zájezdů. Fungování tohoto objektu spočívalo v zavolání uložené procedury v databázi, která na vstupu očekávala seznam filtračních parametrů a která vrátila seznam produktů k zobrazení. Tyto produkty se pak předaly komponentě, která z nich generovala HTML tabulku s obsahem mírně modifikovaným parametry objektu, předanými jako XML atributy. Tento přístup byl zcela v duchu redakčního systému, který cílil na znovu-použitelnost DO a tím tak urychloval a zlevňoval vytváření webu. Časem se ale ukázalo, že každý zákazník si výpis svých zájezdů představuje často úplně jinak, než jeho konkurence a často dokonce i po obsahové stránce. SQL dotaz tak neúměrně rostl o všechny požadované vlastnosti všech zákazníků a i parametry upravující vzhled řádek tabulky přibývaly s každým novým zákazníkem. Zde se tedy naplno projevil problém naznačený v úvodu. Tato

[1] Pojmenování dynamický objekt se v systému Magic2G používá pro ovládací prvky, které je možné vkládat jako dynamické komponenty na web.

[2] Všechny dynamické objekty v systému Magic2G se do obsahu stránky vkládají jako XML tag s prefixem mw (zkratka pro MagicWare).

práce tedy vznikla primárně kvůli výše popsanému problému a její velká část se tedy zaměřuje na nalezení jeho řešení a jeho zobecnění do podoby popsané v předchozí kapitole vytyčující cíl umět vypisovat libovolné entity z databáze bez nutnosti programování nových dynamických objektů.
I přesto, že celá koncepce bude navrhována s přihlédnutím na integraci do systému Magic2G, jedním z hlavních cílů je umožnit existenci redakčního systému jako samostatné aplikace, která se bude dát napojit na libovolný informační systém. Zobecňování řešení ale bude často omezováno kompromisem z důvodů zpětné kompatibility s již existujícím redakčním systémem. Napojení na M2G bude pro účely této práce spočívat jen v ukázkovém propojení na část databáze používané informačním systémem týkající se produktů, jelikož zdrojové kódy systému jsou chráněny autorským zákonem. Následující kapitola vytyčuje nejdůležitější cíle, které by výsledný redakční systém měl splňovat s přihlédnutím požadavky primárně z předchozí kapitoly a částečně vycházející ze zkušeností s procesem vývoje ve společnosti MagicWare a požadavkem na finální integraci systému s informačním systémem Magic2G.

1.3 Požadované cíle

- Základem návrhu by měl být systém pro tvorbu webových aplikací s možností přidávat nové stránky a editovat stávající obsah bez nutnosti rekompilace nebo i restartu aplikace. Principu bude vycházet z již existující koncepce používané v redakčním systému Magic2G.
- Uživatel, který vytváří obsah, by měl mít URL plně pod kontrolou. Měl by mít tedy možnost URL měnit i za běhu aplikace bez narušení existujících odkazů. Mimo to by měla být k dispozici i možnost nějak umožnit nadefinovat dynamickou část URL adres – například pokud by uživatel chtěl zobrazit stránku s detailem o nějakém produktu, systém by měl umět rozpoznat, o který produkt se jedná jen na základě URL adresy. Například z URL adresy `egypt.cestovka.cz/al-nabila`, by měl systém poznat, že se například jedná o hotel „Al Nabila“ v destinaci Egypt. Z uvedeného příkladu je patrné, že systém by měl umět i dynamicky vytvářet a rozpoznávat domény třetího a vyššího řádu. Takováto správa URL je totiž nejdůležitějším nástrojem pro SEO[3] optimalizace celého webu.
- Hlavním cílem je tvoření dynamického obsahu z dat uložených v relační databázi. A to bez nutnosti programování – tedy aby toto bylo zpřístupněno i uživatelům, kteří mají malé nebo i žádné zkušenosti

[3] Search Engine Optimization. SEO optimalizacím webu se obecně říká technikám, jak zajistit co nejlepší výsledek stránek při hledání relevantního obsahu pomocí internetových vyhledávačů. Obsáhlejší definici SEO lze najít například na stránkách Wikipedie (5). Podrobněji popsané techniky používané v praxi například v (4).

s programováním aplikací napojených na databáze. Pokud tohoto docílíme, výsledkem například bude velké zjednodušení případného nasazení stejného informačního systému jinému zákazníkovi, který ale bude požadovat jiný obsah webu. Z procesu nasazení tohoto informačního systému se totiž bude moci přenést velká část požadavků z programátora na HTML kodéra[4] či designéra s mírnými zkušenostmi s HTML a CSS – bude tak vyřešen problém popsaný v předešlé kapitole.

- Mimo dynamických výpisů dat z databáze by měl systém obsahovat i možnost rozšířit systém o libovolně složitou aplikační logiku, která je ale v maximální možné míře oddělena od vzhledu, resp. struktury generovaného HTML. Častým problémem, který doprovází vývoj webových aplikací, je totiž ten, že většina aplikačních programátorů se nesoustředí na vzhled aplikace, protože ten má na starosti například některý z jeho kolegů designérů či kodérů. Před nasazením konkrétní funkční části se pak nezřídka stává, že se programátorovi požadavek vrací s tím, že je nutné pouze zpřeházet generované HTML nebo například jen přidat ke konkrétním HTML tagům nějaké CSS třídy. Takovéto velice jednoduché požadavky a úpravy často ale vyžadují vystavení nové verze aplikace a znemožňují tak plynulé zpracování požadavku kladeného na HTML kodéra. Umožnit designérovi upravit nebo i kompletně změnit generované HTML, by dovolilo tuto situaci zcela eliminovat a usnadnil by se tak celý proces vývoje.
- Administrace by měla být primárně přes webové rozhraní, jak je tomu u jiných redakčních systémů běžné. Očividné výhody plynoucí z webového rozhraní kazí ale fakt, že všechny existující redakční systémy s tímto rozhraním nutí uživatele přizpůsobit se jejich webovému editoru, který ale v drtivé většině nabízí jen velmi omezené funkce. Často se pak stává, že si uživatelé kopírují obsah z tohoto jednoduchého webového editoru do uživatelsky mnohem přívětivějšího desktopového. V něm pak provedou požadované úpravy a pak tento pozměněný obsah znovu zkopírují zpět do webového rozhraní, kde poté klinout na tlačítko pro uložení. Odstranění této zjevné nevýhody, by proto mohlo být jedním z důležitých cílů, jenž by dával zajímavý náskok oproti současné situaci na poli dnešních redakčních systémů. Redakční systém obsažený v informačním systému Magic2G již existuje, a protože zapadá do celkového designu používaného v celém M2G, bude webová administrace této práce pouze doplňkovou funkcí.

[4] V praxi se při vývoji webových aplikací často dělí vývojáři na programátory a HTML kodéry podle úrovně znalosti programovacích jazyků. Programátoři tak implementují aplikační logiku a vzhled aplikace (tedy zdrojový kód webové stránky) se dává na starosti HTML kodérům, kteří nemají dostatek zkušeností nebo znalostí s vývojem aplikací, ale kteří velmi dobře ovládají jazyk HTML a CSS.

- Při nasazení zamýšleného redakčního systému v situaci, kdy již existuje nějaká databáze, ze které se mají vypisovat informace na webu, by datová vrstva zajišťující přístup k těmto datům měla umožňovat využití nějakého nástroje, který by na základě relační struktury této databáze uměl vygenerovat podklady potřebné pro její fungování. Velmi se tím tak usnadní případné prvotní nasazení.
- V systému M2G již existuje velké množství existujících dynamických objektů, které byly a jsou často rozšiřovány o nové funkce. V praxi se pak ukazuje, že ne vždy je jasné, co je a co není v konkrétní verzi systému k dispozici – zvláště pak v situacích, kdy například dokumentace objektů není zcela v souladu s naprogramovanou funkcionalitou (v praxi bohužel velmi častá a politováníhodná situace). Systém by měl umožňovat nějakou formu systémové nápovědy, ze které by bylo jasně patrné, co lze v systému využívat a jaké mají například dynamické objekty parametry. Tato nápověda by v ideálním případě mohla nahradit často neaktualizovanou verzi dokumentace.

2 Analýza řešení

V celé této kapitole bude popsán základní návrh aplikace a to ze dvou rozdílných pohledů. Prvním z nich bude z pohledu koncového uživatele, který bude redakční systém používat – tedy dle nejvyšší úrovně abstrakce. Bude zde rozebráno, jak nejlépe navrhnout funkčnost tak, aby vytváření webů bylo co nejjednodušší a snadno pochopitelné – často ale zde bohužel bude muset být zachována zpětná kompatibilita se systémem M2G. Hlavní náplní první části však bude návrh koncepce pro tvorbu dynamického obsahu, který bude hrát klíčovou roli v celkovém návrhu implementace systému. Druhý pohled pak bude z pohledu tvůrce – tedy implementátora navržené koncepce. Hlavním cílem této kapitoly je určení a návrh hlavních komponent systému, ze kterých se bude skládat a které bude nutné implementovat.

2.1 Řešení uživatelského prostředí a fungování redakčního systému

2.1.1 Struktura webu

Struktura webu v již existujícím redakčním systému Magic2G se rozděluje do tzv. sekcí a stránek. Princip tohoto rozdělení je totiž založen na postupu tvorby webu, při kterém se většinou postupuje tak, že se vytvoří jakási šablona, která určuje záhlaví a zápatí stránek, které je většinou shodné pro nějakou ucelenou část webu. Do této šablony se pak vkládá obsah jednotlivých stránek. Tento způsob návrhu právě reflektuje rozdělení na tzv. sekce a stránky. Sekce odpovídá šabloně, zatímco stránka odpovídá obsahu do šablony vkládaného celý princip je vidět na obr. 1.

Obr. 1 Obsah sekce jako šablona obsahu stránky

Při bližším pohledu také zjistíme, že na stejném principu většinou funguje i struktura URL adres stránek. Stránky spadající pod jednu sekci většinou sdílejí stejnou URL a jen ji upřesňují. Příklad můžeme uvést třeba na webu pro některou z cestovních kanceláří. Ta může prodávat klasické pobytové zájezdy pod URL adresou `www.cestovka.cz/pobytovky/` zatímco speciální golfové zájezdy může prodávat pod adresou `www.cestovka.cz/golf/`. URL adresu stránky lze tedy zadávat pouze její upřesňující částí oproti URL adrese sekce, jak je naznačeno na obr. 2. Hlavní URL adresa webu – tedy například `www.cestovka.cz`, je v redakčním systému M2G uložena jako součást aplikační konfigurace v souboru `web.config`. URL adresa sekce je tedy upřesňující částí této hlavní neměnné URL adresy. Každá sekce a stránka tak obsahuje mimo název a obsah i vlastnost definující její URL adresu. Složením základní URL adresy webu, URL adresy sekce a URL adresy stránky je tak sestavena kompletní URL adresa stránky, která je zadávána do adresního řádku webového prohlížeče.

Obr. 2 Hierarchická struktura URL sekcí a stránek

Uživatelé nemívají problém rozdělení na sekce a stránky pochopit, protože se jedná o rozdělení přirozeně plynoucí z návrhu webového designéra. Při návrhu se však ještě nabízela alternativa umožnit hierarchické rozdělení typu sekce – sekce - … - stránka. Tuto variantu jsme ale opustili z důvodu její zbytečné složitosti.

Při úvahách o možném rozšíření struktury webu jsme ještě dospěli k závěru, že by bylo vhodné ještě nadefinovat jednu úroveň – a to projekt, který by sjednocoval několik sekcí. Důvodem je například to, že se často stává, že si uživatel přeje, aby některé sekce byly přístupné pod úplně jinou doménou než jiné (reálným příkladem může být například web cestovní kanceláře Alex `www.ckalex.cz` a její druhý web `www.svatbyvrecku.cz`, jenž sdílejí dynamický obsah – tedy napojení na databázi). Dalším důvodem pro toto rozdělení je zároveň i častý požadavek, aby jeden projekt, byl přístupný pod více URL – opět můžeme uvést reálný příklad, kdy `www.seniorijedoukmori.cz` vede na web `www.ckalex.cz` – zde se ale nejedná o pouhé přesměrování – systém

totiž musí rozpoznat veškeré stránky i sekce pod oběma URL a v případě nalezení jen přesměrovat s HTTP stavem 301 na hlavní URL, kterou je `www.ckalex.cz`.

2.1.2 Statický a dynamický obsah stránek

Obsahem sekcí a stránek by měla být definice obsahu formou HTML – tedy to, na co jsou HTML kodéři a designéři zvyklí a také z důvodu zpětné kompatibility se systémem Magic2G, který takto obsah již ukládá. Při zpracování stránky není navíc takovýto obsah nutné nijak zpracovávat, protože ho lze rovnou zapsat na výstup. V systému Magic2G obsahem sekcí a stránek je HTML, které je určeno ke vložení do těla HTML tagu `body` – HTML kodér se tedy nemusí neustále psát se základní strukturou HTML stránky (tedy definováním `DOCTYPE` a strukturou HTML tagů `html` - `head` - `body`). Mimo vlastní obsah je součástí každé stránky také definice pro HTML, které je určené pro vložení to těla HTML tagu `head`, kam by mohl HTML kodér případně definovat přilinkované soubory nebo jinou logiku patřící právě do hlavičky HTML stránky.

Součástí obsahu musí ale nepochybně také být nějaká informace o tom, jak a kde zobrazovat dynamický/aplikační obsah nebo například i místo, kam se do obsahu sekce má vložit obsah stránky. V systému Magic2G je toto řešeno pouhým „doplněním“ HTML značek o nové XML tagy, které se při zpracování stránky nahrazují požadovaným HTML, na které může (ale také nemusí) být navěšena aplikační logika dynamického objektu. Příklad obsahu stránky v systému Magic2G s vyhledávacím formulářem je vidět na obr. 3.

```
<h1>Vyhledání Vaší dovolené</h1>
<mw:SearchProductForm NextPageID="123"
  ShowProductName="true" ProductNameTitle="Název hotelu:"
  ShowPriceRange="true" PriceRangeTitle="Cenový rozsah:"
  SubmitButtonText="Vyhledat mojí dovolenou" />
```

Obr. 3 Příklad dynamického objektu v systému Magic2G

Je na něm vidět definice statického HTML ve formě nadpisu první úrovně a zároveň použití dynamického objektu `mw:SearchProductForm`, který vkládá ASP.NET ovládací prvek obsahující vyhledávací formulář pro vyhledávání zájezdů. Jak je ale z obrázku patrné, HTML kodér zde nemá moc šancí jak vzhled formuláře ovlivnit. Pouze pomocí parametrů jako je například `NextPageID`, `ShowProductName` nebo `SubmitButtonText`, je mu umožněno si nadefinovat, na jakou stránku se má formulář po vyplnění přesměrovat, jaké vyhledávací prvky má formulář obsahovat, nebo text, který má být uveden na tlačítku pro odeslání formuláře. Vygenerované HTML je ale natvrdo součástí ovládacího prvku, ke kterému má přístup pouze programátor. HTML kodér se tak nemůže rozhodnout, zdali použít například tabulkový layout či layout

postavený na kontejnerových elementech – toto rozhodnutí totiž už učinil programátor dynamického objektu `mw:SearchProductForm`.
Jako cíl jsme si však vytyčili vytvořit systém umožňující HTML kodérovi mít plnou kontrolu nad generovaným HTML. Stávající jednoduché řešení dynamických objektů v systému Magic2G toto však nijak neumožňuje. Zde je tedy nutné navrhnout nový způsob, jak definovat dynamické objekty. Z důvodu zpětné kompatibility je určitě vhodné zůstat u již zavedeného principu vkládání dynamického obsahu pomocí XML tagů. Mimo jiné i proto, že HTML kodéři jsou již seznámeni s principem, na kterém je celé HTML postaveno – tedy že tag = funkce/objekt a atribut = modifikace chování.
V duchu HTML se jako řešení nabízí využít strukturu značkovacích jazyků – tedy umožnit mít dynamickým objektům obsah. Tento obsah by pak mohl být použit jako šablona HTML, do které by se jen umisťovaly pro fungování důležité prvky, jakým je například ASP.NET ovládací prvek `TextBox` pro zadání vyhledávané hodnoty nebo tlačítko pro odeslání formuláře. Příklad takovéhoto nastavení může být vidět na obr. 4, kde je vložen dynamický obsah, který se chová stejně jako ten na obr. 3, jen s tím rozdílem, že zde si zde HTML kodér sám určil, jak bude vypadat layout a zároveň si i tím určil, co bude obsahem vyhledávacího formuláře – tedy že bude existovat jeden ASP.NET ovládací prvek pro zadání názvu hotelu a dva další prvky pro zadání rozsahu požadované ceny zájezdu.

```
<h1>Vyhledání Vaší dovolené</h1>
<mw:SearchForm NextPageID="123"
  <div>
    <strong>Název hotelu:</strong>
    <span><mw:Input ParameterName="ProductName" /></span>
  </div>
  <div>
    <strong>Cenový rozsah:</strong>
    <span>
      <mw:Input ParameterName="PriceFrom" />
      -
      <mw:Input ParameterName="PriceTo" />
    </span>
  </div>
  <mw:SubmitButton Text="Vyhledat mojí dovolenou" />
</mw:SearchForm>
```

Obr. 4 Příklad definice dynamického objektu s vlastním HTML layoutem

Princip je tedy založený na tom, že stejně jako doposud by existoval nějaký dynamický objekt (`mw:SearchForm`), který by obsahoval parametry ovlivňující jeho chování (například `NextPageID` určující, kam se má formulář přesměrovat) a jeho obsah by určoval, jak se má objekt vyrenderovat (tzn., jak má vypadat HTML layout) a co má přesně obsahovat. Toto řešení zdá se být dostatečným pro splnění našeho cíle. Nicméně má jednu velkou

nevýhodu. HTML kodér by při každém použití formuláře musel znovu a znovu definovat, jak má formulář vypadat. Toto u tak specifického formuláře, jakým je `mw:SearchForm`, který se zpravidla na každém webu liší, nevadí. Zjevná kontraproduktivita nastává například u případného formuláře sloužícího pro přihlašování koncových uživatelů do newsletterů, který by se téměř vždy skládal pouze z jednoho ASP.NET ovládacího prvku pro zadání emailové adresy a jednoho tlačítka pro registraci. Bylo by zbytečné, aby zde HTML kodér musel znovu a znovu definovat obsah. Pro řešení tohoto problému nemusíme chodit daleko – stačí se podívat zpět na obr. 3. Pokud bychom dovolili, aby dynamické objekty mohly mít nějaký výchozí obsah, mohlo by použití například zmíněného objektu pro přihlašování do newsletterů vypadat tak jako na obr. 5.

```
NewsletterForm s výchozím obsahem:
<mw:NewsletterForm />

NewsletterForm s uživatelským definovaný obsahem:
<mw:NewsletterForm>
  Váš email: <mw:EmailAddress />
  <mw:SubmitButton Text="Registrovat" />
</mw:NewsletterForm>
```

Obr. 5 Příklad využití výchozího obsahu vs. uživatelsky definovaného

První použití objektu `mw:NewsletterForm` je definováno nepárovým tagem – tedy s výchozím obsahem, který by mohl definovat jako doposud programátor. Druhé použití je již definováno párovým tagem, kde si obsah definuje sám HTML kodér – tedy, že obsah má být tvořen popiskem, jedním ovládacím prvkem pro vložení emailové adresy a tlačítkem pro zaregistrování.

2.1.3 Dynamické výpisy záznamů z databáze

Podobným způsobem, jaký byl použit pro definování HTML layoutu aplikačních formulářů, by mohly jít nadefinovat různé výpisy záznamů z databáze – mělo by se jen určit, co se má vypisovat a jak má vypadat tzv. šablona pro jednotlivý záznam. Na základě této šablony by se měl pak na výstupu zobrazit seznam konkrétních záznamů. Příklad této koncepce je zobrazen na obr. 6.

```
<mw:DynamicList Source="Product">
  Seznam produktů
  <ul>
    <mw:Template>
      <li><mw:Field Type="ProductName" /></li>
    </mw:Template>
  </ul>
</mw:DynamicList>
```

Obr. 6 Příklad definice šablony pro výpis dynamického obsahu z databáze

Je zde vidět opět dynamický objekt – tentokrát například s názvem `mw:DynamicList`, jehož parametr `Source` by mohl určovat, co se má z databáze vypisovat (tedy z jaké tabulky). Obsah objektu by pak opět určoval jakousi šablonu HTML použitou pro výpis. Oproti formulářům je zde ale nutné definovat, která část šablony se má použít pro zopakování pro každý ze záznamů. Toto by například mohlo být řečeno opět dynamickým objektem s příznačným názvem `mw:Template`, který by jen ohraničoval tu část, která se má opakovat pro každý z nalezených záznamů. Jednotlivé typy vlastností záznamů (tzn. hodnot ze sloupců tabulky), by mohly být vypisovány dynamickým objektem `mw:Field`, jehož atribut `Type`, by obsahoval název vlastnosti, která se má místo něj na výstupu objevit – na obrázku je například použita vlastnost uchovávající název produktu.

2.1.4 Dynamické části URL

Poslední důležitou částí je návrh definice dynamických částí URL adres. Jelikož zadávání částí URL (tedy jedné z vlastností stránek nebo sekcí) je řešeno pouze textově, je nutné umožnit uživateli definovat určitou část tak, aby mohla být systémem interpretována jako dynamická – tedy část URL, jejíž hodnota je řízená nějakým URL parametrem. Cílem je tedy místo URL adresy například pro detail hotelu:

`http://www.cestovka.cz/detail-hotelu/?ProductID=123`

vytvořit adresu, která obsahuje název hotelu přímo v adrese a nevyžaduje tak URL parametr `ProductID`. Například:

`http://www.cestovka.cz/detail-hotelu/al-nabila/`

Pro toto je nutné vymyslet způsob / syntaxi, jak toto bude zadáváno. Stejně jako dynamické objekty i komponenty, které budou sloužit k překladu dynamických částí URL, by měly být nastavovatelné – tedy umožnit nějakým způsobem modifikovat jejich chování. Vzhledem k tomu, že uživatel je již seznámen s principem, kterým jsou definovány dynamické objekty ve stránce, můžeme využit těchto jeho znalostí a aplikovat stejný princip i v definici dynamických částí URL – tedy zadávání formou XML tagu. Příklad nastavení části URL adresy pro stránku je uveden na obr. 7,

```
detail-hotelu/<mw:Product Type="Name" />
```

Obr. 7 Příklad definice dynamicky překládané části URL adresy stránky

kde je vidět způsob zadání URL adresy pro detail hotelu, jehož URL obsahuje název hotelu, jež může být například definován jako objekt `mw:Product` s parametrem určujícím, co přesně má být obsahem URL.

2.2 Implementační analýza

2.2.1 Určení cílové platformy

Cílová platforma je díky požadavku zpětné kompatibility se systémem Magic2G předem definována – tedy ASP.NET a jazyk C#. Kdybychom však neměli požadavek na zpětnou kompatibilitu, určitě by cílovou platformou byl rovněž framework .NET, jež je dle mého názoru jednou z nejvyspělejších a nejrychleji se rozvíjejících vývojových platforem vůbec, která zaručuje, že i za několik let bude kód použitelný a přenositelný na novější verze. Jako databáze bude primárně použit Microsoft SQL Server 2008 (MSSQL) – rovněž z důvodu, že i systém Magic2G běží na tomto serveru.

2.2.2 Zpracování HTTP požadavku

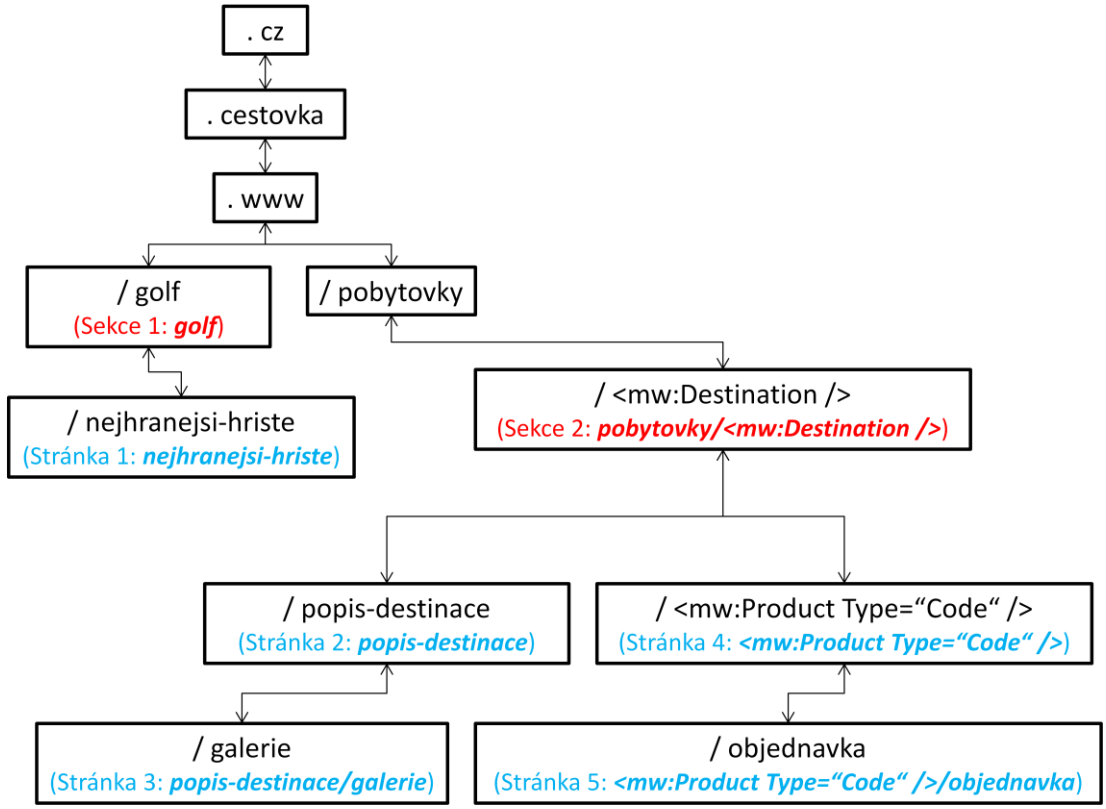
Zpracování HTTP požadavku bude muset projít několika fázemi:

1. Rozpoznáním URL adresy a překladu jejích dynamických částí do URL parametrů
2. Parsováním obsahu nalezené sekce a stránky a identifikace dynamických objektů
3. Vytvořením konkrétních instancí dynamických objektů
4. Vyrenderováním HTML obsahu definovaného pomocí šablon

Jednotlivé fáze budou blíže rozebrány v následujících kapitolách 2.2.3 - 2.2.5.

2.2.3 Rozpoznávání URL

První fáze bude vyžadovat implementovat vyhledávací strom reprezentující kompletní mapu URL adres celého projektu. Strom URL adres (jehož příklad je na obr. 8) vychází z principu, že každá část URL adresy má svoji váhu – nejdůležitější je doména prvního řádu – tedy nejpodstatnější část URL adresy, která bude v kořenovém uzlu vyhledávacího stromu. Další patra stromu pak budou reprezentovat jednotlivá doménová jména, po jejichž vyčerpání se postupuje po jednotlivých částech cesty oddělované lomítky směrem od domény prvního řádu.



Obr. 8 Ukázka stromu pro rozpoznávání URL adres

Každý uzel, který bude reprezentovat nějakou stránku (modrá barva) nebo sekci (červená barva) v sobě bude obsahovat i identifikátor této stránky či sekce.

Rozpoznávání bude muset počítat i s možností dynamicky tvořenými částmi URL adres – tedy situace, kdy je místo textu definován XML tag (například na obrázku je tímto tagem `mw:Destination` v sekci s ID 2). Uzly stromu budou muset umět tyto XML tagy parsovat a dle jejich typu umět vytvářet příslušnou komponentu, která bude spolupracovat s databází a překládat daný text na některý z URL parametrů. Překlad textu na samostatné URL parametry má tu výhodu, že zbytek systému nemusí ani vědět, že nějaká část URL byla dynamicky přeložena – systém toto bude interpretovat, jako kdyby v URL adrese byl například uveden parametr `ProductID` s hodnotou ID produktu, jenž je reprezentován daným textem v URL adrese.

Při opačném postupu, kdy bude potřeba sestavit kompletní URL stránky, bude možné využít stejného stromu určeného pro rozpoznávání. Bude se pouze postupovat opačným směrem, kdy se přistoupí rovnou k uzlu, který reprezentuje danou stránku a průchodem stromu směrem ke kořeni se bude sestavovat celá URL adresa. Dynamické části adresy budou využívat kolekci parametrů, které se mají připojit k výsledné adrese. Například při uvedeném parametru `ProductID` bude možné na základě jeho hodnoty dohledat text určený pro překlad URL daného produktu a tento vložit do sestavované adresy.

2.2.4 Parsování obsahu stránky

Po rozpoznání URL adresy a konkrétní stránky bude nutné zjistit obsah sekce, ve které se stránka nachází, a tento obsah předat ke zpracování. Pro zpracování bude třeba vytvořit parser, který bude muset umět vyhledávat a parsovat XML tagy reprezentující jednotlivé dynamické objekty. Následně na základě názvů těchto tagů bude zodpovědný za správné vytvoření objektu reprezentující daný tag a jeho začlenění do seznamu ASP.NET ovládacích prvků zpracovávané ASP.NET stránky. Parser bude muset také počítat s možností syntaktické chyby, kdy se uživatel například překlepne v názvu nebo neukončí hodnotu parametru uvozovkami či zapomene na ukončovací tag. Žádná z těchto chyb by neměla být fatální a parser by ji nejen měl umět rozpoznat a ignorovat tako chybně zadané objekty, ale také by měl o ni uživateli dát nějakým způsobem vědět. A protože HTML stránky neprocházejí žádnou formou kompilace, nejlepší způsob, jak tomu dosáhnou, je vygenerovat nějakou chybovou hlášku přímo do výstupního HTML kódu dané stránky.

2.2.5 Vytváření dynamických objektů a jejich výstup

Každý z rozpoznaných objektů bude zodpovědný za vytvoření požadovaného ASP.NET ovládacího prvku. Tento se následně předá parseru, který ho vloží do seznamu ovládacích prvků zpracovávané ASP.NET stránky. Stejným způsobem bude parser přistupovat ke statickému HTML jen s tím rozdílem, že bude rovnou vytvářet `asp:LiteralControl`, který bude rovnou vkládat do kolekce ovládacích prvků ASP.NET stránky. HTML renderované dynamickými objekty si pak budou řídit jednotlivé ovládací prvky sami. Díky tomuto principu bude možné implementovat dynamické objekty jak jednoduchým ASP.NET uživatelským ovládacím prvkem (tedy přístup zpětně kompatibilní se stávajícím redakčním systémem v M2G), tak libovolnou třídou, která využívá ASP.NET ovládací prvek jen jako prostředek k renderování obsahu. Tato poslední metoda zároveň umožní docílit vytváření formulářových aplikací, které nemají předem jasně danou strukturu generovaného HTML. Jednoduše proto, že si sám HTML kodér řekne, kde na stránce se bude vyskytovat konkrétní textové pole (ovládací prvek `asp:TextBox`) a či konkrétní tlačítko, tak jak bylo nastíněno na obr. 4 v předešlé kapitole 2.1.2. Programátor se tak bude moci zaměřit pouze na aplikační logiku a HTML layout může zcela přenechat na HTML kodérovi, jenž bude vytvářenou komponentu používat.

2.2.6 Komponentový systém a generovaná nápověda

Komponentový systém pro tvorbu dynamického obsahu musí obsahovat i možnost nějakým způsobem nadefinovat nápovědu k nově vytvářeným komponentám tak, aby měl HTML kodér k dispozici co nejvíce informací o

tom, jak daná komponenta pracuje. Tento systém nápovědy by měl být koncipován tak, aby při vývoji nové komponenty bylo možné už jen z její definice poznat, jaké například parametry očekává, jaký obsah podporuje a jakým způsobem ji lze na webu použít. Zároveň by ale se ale nemělo jednat o příliš složité a invazivní řešení narušující přehlednost kódu. I v době, kdy je například programátor tlačen blížícím se termínem zpracování k co nejrychlejšímu vyvinutí komponenty, by ho vytvoření nápovědy nemělo nijak omezovat nebo zdržovat. Tento princip zároveň omezí i šanci k zastarání nápovědy oproti změnám, které mohou v budoucnu nastat. Jako řešení se k tomuto účelu nabízí využití reflexe a deklarativního způsobu zápisu parametrů pomocí .NET atributů. Reflexe nám v tomto případě zaručí aktuálnost nápovědy po každém nasazení nové verze systému a atributy zároveň umožní přirozený popis funkcionality, který vyžaduje minimální úsilí ze strany programátora. Mimo tyto zjevné výhody má systém nápovědy založený na reflexi i tu vlastnost, že v případě, kdy by byl redakční systém nasazován v různých verzích na různé serverové instance, by měl uživatel vždy k dispozici seznam toho, co je a co není na dané instanci k dispozici.

2.2.7 Dynamické výpisy záznamů z databáze

Poslední část návrhu uživatelské části aplikace je princip, kterým bude fungovat dynamický výpis obsahu z databáze pomocí dynamického objektu `mw:DynamicList`, který byl navržen v předešlé kapitole 2.1.3 na obr. 6. Celý návrh stojí na myšlence, že jediné, co by měl HTML kodér definovat, je název databázové tabulky a seznam vlastností, které bude chtít vypsat. Dynamický objekt `mw:DynamicList` bude tedy muset fungovat tak, že si zjistí, které všechny vlastnosti jsou v šabloně pro výpis obsaženy, a tyto předá datové vrstvě k jejich zpracování. Datová vrstva musí jen na základě daného seznamu vlastností sestavit SQL dotaz a jeho výsledek předat zpět dynamickému objektu `mw:DynamicList`, který pro každý záznam vyrenderuje zadanou šablonu. Implementačně se bude šablona skládat ze seznamu objektů, které všechny budou implementovat metodu na vyrenderování HTML pro jeden předaný záznam. `mw:DynamicList`, který sám o sobě bude jedním ASP.NET ovládacím prvkem, pak bude při renderování v cyklu jednoduše iterovat všemi záznamy a všemi objekty a postupně tak tvořit výsledné HTML, jenž bude zapisovat na svůj výstup. Bude tedy založen na podobném principu, jako funguje třída `asp:Repeater`. Aby byl `mw:DynamicList` v praxi použitelný, bude muset zároveň obsahovat i velké množství filtračních parametrů, díky kterým bude možné výpisy omezovat. Tyto filtrační parametry budou muset být součástí datové vrstvy, protože jediným vhodným způsobem je rovnou sestavovat `WHERE` klauzuli v SQL dotazech – jakékoliv pozdější filtrování záznamů po jejich získání z databáze by bylo extrémně neefektivní.

2.2.8 Volba technologie pro datovou vrstvu

Jednou z nejdůležitějších částí návrhu aplikace je implementace datové vrstvy. Z počátku se nabízelo řešení využít nějakou již existující ORM[5] technologii jakou je například velmi oblíbený Hibernate nebo nedávno uvedená technologie LINQ to SQL. Po jejich prozkoumání se však ukázalo, že ani jedna z těchto variant by nebyla tím nejvhodnějším. Hibernate má například jednu velmi nepříjemnou vlastnost a tím je vždy kompletní získání a namapování všech vlastností objektu z databáze při každém přístupu k němu. Tento přístup, kdy v každém `SELECT` dotazu jsou uvedeny všechny dostupné sloupce z tabulky, není pro redakční systém pro dynamické vypisování záznamů z databáze ideální. Důvod je jednoduchý: V existujících informačních systémech se často vyskytují tabulky s velmi velkým počtem atributů, které jsou sice podstatné pro jeho fungování, nicméně pro prezentaci na webu jsou použitelné jen některé. Přidávání všech sloupců do dotazu při každém přístupu ke stránce je tedy často zbytečné a neefektivní. V praxi při tomto přístupu nelze nikdy využít například tzv. pokrývané indexy, které velmi urychlují dotazy, jež vyžadují jen sloupce držené v těchto indexech – v takovém případě se totiž nemusí přistupovat k primárnímu souboru, kde jsou uloženy všechny atributy, a je možné veškeré informace vyčíst přímo z dat indexu. Tato vlastnost Hibernate technologie zároveň i podrývá cíl tohoto projektu – tedy dynamický výpis záznamů, kdy se dopředu nedá předpokládat nejen, jak má výpis vypadat, ale hlavně i co má výpis obsahovat. Principiálně lze tedy očekávat, že k některým typům záznamů mohou existovat desítky možná i stovky vlastností, které k nim lze na webu uvádět. Požadavek na tvorbu SQL dotazů, které obsahují jen potřebné vlastnosti, je tedy nutností, kterou musí datová vrstva splňovat.

Druhou nabízenou variantou byla technologie LINQ to SQL, která již netrpí podobným problémem jako Hibernate. Zde byla ale překážka jiného typu. LINQ to SQL je výborná technologie, kdy programátor předem ví, co má být součástí dotazu a rovnou toto tak může zapsat v prostředí jazyka C#. V případě použití v tomto projektu ale obsah dotazu předem neznáme. Bylo by proto velmi obtížné navrhnout způsob, jak převádět seznam textově definovaných vlastností zjištěných za běhu aplikace do podoby, které LINQ rozumí – tedy jako volání extension metod, jenž typicky jako parametr očekávají lambda funkci. Druhou nevýhodou je také fakt, že v případě využití LINQ to SQL nemá programátor téměř žádné prostředky jak ovlivnit obsah samotného SQL dotazu spouštěného nad databází.

[5] Objektově-relační mapování. Tato technologie se stará o automatické plnění vlastností objektů daty z databázové tabulky. Jednomu záznamu v databází tak odpovídá jen objekt v programovacím jazyce, jehož vlastnosti odpovídají sloupečkům v databázi.

Kvůli výše popsaným důvodům bude tedy nutné navrhnout úplně novou datovou vrstvu, která bude jednoduše umožňovat implementovat zamýšlenou funkcionalitu. Návrh této datové vrstvy bude o to složitější z důvodu, že cílem je i možnost vygenerovat podklady pro `mw:DynamicList` na základě již existující struktury databáze nejlépe bez nutnosti programování. Implementace datové vrstvy bude tedy zaujímat velmi důležitou a možná i nejobtížnější část celého projektu.

2.2.9 Návrh implementace datové vrstvy

Návrh datové vrstvy vychází z následujícího, velmi zajímavého faktu: V případě, kdy víme, že chceme získat nějakou informaci, například název katalogu [6], je vždy předem jasné, že se v SQL dotazu musí nějakým způsobem vyskytovat tabulka `Katalog`. V případě, kdy se `Katalog` nenachází v klauzuli `FROM`, je vždy nutné „přijoinovat" tabulku `Katalog`. Například pokud chceme název katalogu produktu, ve kterém je produkt veden, je vždy dokonce i jasné, jak takovýto JOIN má vypadat – je vidět na obr. 9.

```
SELECT k.Nazev FROM Produkt p
LEFT JOIN Katalog k ON p.KatalogID = k.ID
```

Obr. 9 Způsob získání názvu katalogu z tabulky produkt

Jediné, co tento JOIN katalogu produktu vždy vyžaduje, je, aby v dotazu byl dostupný sloupec `KatalogID` z tabulky `Produkt`. Ke stejnému faktu dojdeme i v případě vlastnosti `KatalogID` pro `Produkt`. Například pokud chceme vědět `ID` katalogu, ve kterém je veden produkt, jenž byl prodán v nějakém obchodním případě, vždy je nutné „přijoinovat" tabulku `Produkt`. Pokud využijeme tranzitivní logiky, dojdeme k závěru, že pokud chceme název katalogu, ve kterém je veden produkt prodaný v nějakém obchodním případě (zjednodušeně `ObchodniPripad => Produkt => Katalog => Nazev`), musíme do dotazu dostat tabulku `Katalog` pomocí JOINu vyžadující sloupec `p.KatalogID` z tabulky `Produkt`. Jenže `p.KatalogID` vyžaduje JOIN tabulky `Produkt` dle sloupce `op.ProduktID`. A konečně sloupec `op.ProduktID` vyžaduje, aby v dotazu byla tabulka `ObchodniPripad`. Výsledný dotaz sestavený na základě této logiky je vidět na obr. 10.

[6] V celé práci se jako příklady uvádí vztahy mezi entitami katalog, produkt a obchodní případ. Katalog je entita, která představuje virtuální katalog, ve kterém jsou vedeny jednotlivé produkty, které lze prodávat. Každý prodej je reprezentován jedním obchodním případem. Jednotlivé entity jsou ukládány v databázových tabulkách se stejným jménem a jako primární klíč obsahují umělý atribut ID. Provázání mezi entitami je tak řešeno pomocí cizích klíčů na tyto identifikátory.

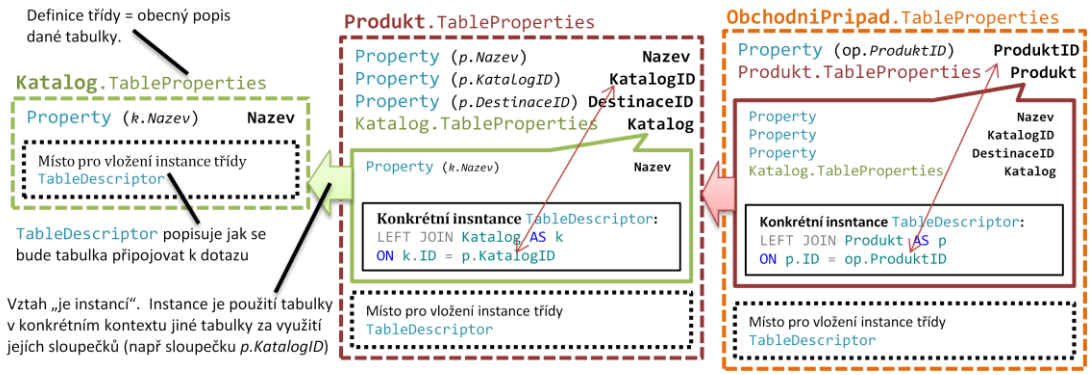
```
SELECT p.KatalogID, k.Nazev FROM ObchodniPripad op
LEFT JOIN Produkt p ON op.ProduktID = p.ID
LEFT JOIN Katalog k ON p.KatalogID = k.ID
```

Obr. 10 Způsob získání názvu katalogu z tabulky ObchodniPripad

Na základě této logiky nám vyplývá fakt, že pouze ze znalosti vlastností, které chceme ve výsledném dotazu, je už předem určeno, jak má dotaz vypadat. Při pohledu na zjednodušený zápis použitý pro sestavení dotazu na obr. 10 (tedy `ObchodniPripad => Produkt => Katalog => Nazev`) nám dává jasnou představu, jak by se postupovalo při sestavování dotazu – tedy že `ObchodniPripad` musí být uveden v klauzuli `FROM`, `Produkt` naopak říká, že v dotazu se musí vyskytovat JOIN tabulky `Produkt` z tabulky `ObchodniPripad`. `Katalog` to samé pro tabulku `Katalog` z tabulky `Produkt` a nakonec samotná vlastnost, kterou lze najít v naposledy „přijoinované“ tabulce. Přesně na tomto principu může být navržena celá architektura datové vrstvy, jež by se mohla skládat ze třech základních objektů:

- Objekt pro popsání samotné vlastnosti, jak získat sloupec z připojené tabulky (tedy SQL výraz typu `k.Nazev`) – tzv. `Property`
- Objekt, který by slučoval všechny vlastnosti, které lze z jedné tabulky získat – tzv. `TableProperties`
- Objekt, který by popisoval, jak nějakou tabulku připojit do SQL dotazu (tedy buď pomocí `FROM` nebo `JOIN` klauzule) – tzv. `TableDescriptor`

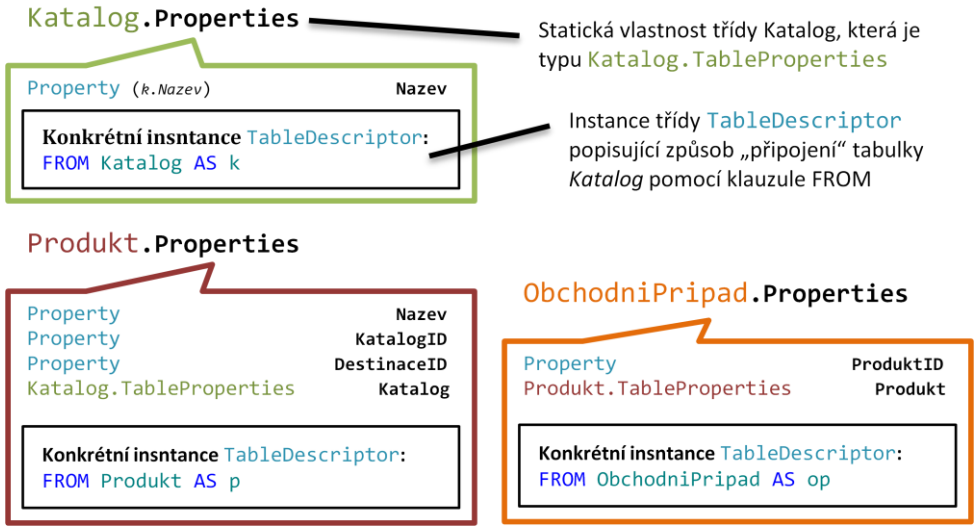
Vztahy mezi těmito třemi základními třídami jsou vidět na obr. 11, který bude popsán dále v textu.



Obr. 11 Příklad vazeb mezi hlavními objekty datové vrstvy

Pro každou tabulku v databázi by měla existovat třída `TableProperties`, ve které by byly definovány všechny sloupce – tedy vlastnosti, které jsou v dané tabulce obsaženy. Konstruktor každé z těchto tříd by očekával potomka třídy `TableDescriptor`, jenž by definoval způsob, jak danou tabulku (a tedy všechny její vlastnosti) připojit k dotazu. Na obrázku ve třídě `Produkt.TableProperties` (červená) jsou vidět vlastnosti `Nazev`, `KatalogID` a `DestinaceID` a zároveň i vlastnost `Katalog`, která je ale již instancí třídy

`Katalog.TableProperties` (zelená) s předaným deskriptorem (objekt třídy `TableDescriptor`), který popisuje, jak se tabulka `Katalog` dá připojit k tabulce `Produkt`. Stejným způsobem obsahuje třída `ObchodniPripad.TableProperties` (oranžová) vlastnost s názvem `Produkt` typu `Protukt.TableProperties` s předaným deskriptorem popisujícím, jak lze tabulku `Produkt` připojit k tabulce `ObchodniPripad`.
Každá třída `TableProperties`, by také existovala jako statická instance s předaným deskriptorem, který by místo klauzule `JOIN` obsahoval informaci, že všechny sloupce lze získat z tabulky uvedené za `FROM`, jak je vidět na obr. 12.



Obr. 12 Statické instance vlastností objektů

Hlavním úkolem datové vrstvy je nepochybně sestavení SQL dotazu a vrácení jeho výsledku z databáze. Tuto nejdůležitější část by mohla implementovat jedna základní třída `DataService`, která bude muset na základě předaných vlastností sestavit výsledný dotaz a získat jím vybraná data z databáze. Implementačně se zde nabízí dvě možnosti:

- jedna univerzální třída, která by očekávala informaci o výchozí tabulce (tedy té za `FROM`) spolu se seznamem vlastností, nebo
- pro každou tabulku by existoval potomek základní třídy `DataService`, který by již měl v sobě obsaženu informaci o výchozí tabulce, a její použití by tedy očekávalo jen předání seznamu vlastností

První možnost má tu výhodu, že by se na ní dal dobře navázat `mw:DynamicList`, jenž v návrhu předpokládá způsob zadání výchozí tabulky v parametru `Source` (pro připomenutí viz kapitola 2.2.7 obr. 6 na straně 7). Druhá možnost by v případě použití v `mw:DynamicList` znamenala vytvoření nějakého mapování mezi názvem tabulky a pro ni definovaného potomka třídy `DataService` – tedy implementaci varianty návrhového vzoru Factory. Nicméně oproti první možnosti nabízí mnoho jiných východ:

- filtrační parametry půjde implementovat přímo ve třídě pro konkrétní tabulku například jen pomocí vlastností, jež by každá představovala jednu podmínku
- odvozené třídy od `DataService` by mohly být silně typové a tím by mohlo být zajištěno jejich správné použití (tedy že do třídy pro tabulku `Katalog` nebude předána vlastnost určená pro `Produkt`)
- každý potomek by zároveň bylo i ideální místo, kde by se daly implementovat jiné metody, které souvisí s danou tabulkou – například metody typu `INSERT` nebo `UPDATE`, či jiné metody využívané aplikační logikou a vyžadující přístup do databáze

Druhá varianta se tedy jeví jako vhodnější.

Posledním důležitým objektem bude nepochybně ten, ve kterém se budou uchovávat data pro jeden záznam vrácený databází. Protože datová vrstva stojí na principu, který není typickou ORM technologií, nelze tak mapovat data vytažená z databáze na nějaký konkrétní objekt a jeho C# vlastnosti. Je nutné tedy vymyslet, jak zajistit, aby objekty reprezentující datový záznam obsahovaly jen vlastnosti vytažené z databáze. K tomuto účelu se nabízí využít zapouzdření .NET kolekce `System.Collections.Generic.Dictionary` do třídy `DataItem`. Slovník by jako klíč mohl očekávat objekt typu `Property` a hodnota by mohla být typu `object`. Třída `DataItem` by mohla veřejně implementovat `indexer`, kterým by zpřístupňovala interní slovník.

Navržená datová vrstva by mohla najít uplatnění i v jiných projektech, které vyžadují stejnou jednoduchost, jako například technologie Hibernate, ale zároveň cílí na výkonnost i z pohledu optimalizace výběru jen některých sloupců. Bylo by proto vhodné umožnit její přímé využití pro komunikaci s databází a neomezit se jen na použití v objektu `mw:DynamicList`.

Finální podoba přímého použití potomka třídy `DataService` pro obchodní případ by mohla vypadat jako na obr. 13, kde je vidět předání třech vlastností (číslo obchodního případu, název prodaného produktu a jeho katalogu) a nastavení dvou filtračních parametrů (v tomto případě pro nalezení všech obchodních případu, kde byla vydána faktura a byl v nich prodán produkt s ID 123).

```
ObchodniPripadService dataService = new ObchodniPripadService();
dataService.AddProperties(
  ObchodniPripad.Properties.Cislo,
  ObchodniPripad.Properties.Produkt.Nazev,
  ObchodniPripad.Properties.Produkt.Katalog.Nazev);
dataService.FilterParameters.ExistujeVydanaFaktura = true;
dataService.FilterParameters.ProduktID = 123;
foreach (DataItem item in dataService.GetList(DataAccess.Default))
{
  Console.WriteLine(item[ObchodniPripad.Properties.Cislo]);
  Console.WriteLine(item[ObchodniPripad.Properties.Produkt.Katalog.Nazev]);
}
```

Obr. 13 Příklad přímého použití potomka DataService

2.2.10 Mapování vlastností datové vrstvy pro mw:DynamicList

Relační struktura databáze často odráží její použití v aplikační logice celého informačního systému. Tato struktura často obsahuje i spoustu atributů a tabulek, které by neměly být z webu přístupné a ani `mw:DynamicList` by takovéto tabulky a atributy neměl rozpoznávat. U každého sloupce a tabulky bude tedy nutné evidovat, zdali je možné tyto použít v `mw:DynamicList`. Mimo tento požadavek bude také vhodné umožnit jiné pojmenování tabulek i vlastností – už jen z důvodu toho, že by všechny názvy měly být jednotné a odrážet jejich skutečné použití z pohledu uživatele. Například sloupec `OrganizatorFirmaPartyID` by se v `mw:DynamicList` měl jmenovat `OrganizerID` – HTML kodér totiž často ani neví, co část názvu `PartyID` znamená (že ukazuje na databázový vzor Party). Tyto implementačně zvolené názvy by velmi zesložiťovaly celkové použití `mw:DynamicList`.

Oba důvody nás tedy přivedly k přesvědčení, že bude vhodné implementovat nějakou formu mapování mezi vlastnostmi používané v databázové vrstvě a těmi, které bude používat HTML kodér v `mw:DynamicList`. Součástí tohoto mapování by zároveň i mělo být zpřístupnění filtračních parametrů implementovaných v potomcích `DataService` tříd.

Pro potřeby mapování bude tedy nutné vytvořit třídu `DynamicListRegister`, která bude obsahovat metody, jenž umožní vytvořit požadovaná mapování, tato třída může zároveň i sloužit jako Factory, jenž bude na základě názvu tabulky vytvářet požadovaný `DataService`. Její hlavní činnosti budou tedy následující:

- Umožnit zaregistrovat pod libovolným jménem konkrétní `DataService` – a tedy i tabulku, ze které se budou moci vypisovat informace
- Umožnit zaregistrovat pod libovolným jménem libovolnou vlastnost a filtrační parametr
- Poskytnout prostředky pro `mw:DynamicList`, které umožní vytvořit a správně nastavit `DataService` – tedy na základě textových názvů bude umět přidávat konkrétní objekty typu `Property` (vlastnosti) a nastavovat filtrační parametry libovolného typu z textové hodnoty

Implementačně jsme zde zvolili stejně jako v případě nápovědy k dynamickým objektům použití reflexe. Registrace tak bude spočívat pouze v předání textového aliasu pro `mw:DynamicList` a textového názvu vlastnosti. Toto umožní vytvořit celé mapování pomocí jednoho XML souboru a usnadní se tak jeho tvorba. Nevýhoda je zde v nemožnosti kompilace a tedy nalezení případného překlepu. Výhoda jednoduchosti v definici pomocí XML však převažuje, a tak se nám toto řešení jeví jako nejvhodnější.
Součástí mapování může být i nastavování například výchozích hodnot filtračních parametrů – například tabulka `Produkt` může obsahovat příznak, zdali je produkt viditelný na webu či nikoliv. `mw:DynamicList` by toto měl vždy respektovat, a tedy v mapování se bude moci nastavit výchozí hodnota pro parametr `JeViditelnyNaWebu` na hodnotu `true`.

2.2.11 Generátor datové vrstvy

V kapitole 2.2.9 jsme navrhli strukturu potřebnou pro dynamické sestavování SQL dotazů. Tato struktura by však dle našich cílů měla být ideálně celá generovaná na základě relační struktury databáze. Pro tento účel bude nutné vytvořit nástroj, který bude číst tuto strukturu a na jejím základě bude generovat všechny základní třídy datové vrstvy.
Generátor může být postaven pomocí technologie Windows Presentation Foundation (WPF), která umožňuje velmi rychlou a jednoduchou tvorbu desktopových aplikací. Více se o této technologii dá najít v knize „Windows Presentation Foundation Unleashed“ (2). Bude mít velmi jednoduché rozhraní, které umožní načíst všechny tabulky z databáze a určit ty, pro které se má vygenerovat datová vrstva. Na základě cizích klíčů bude možné zjistit provázání mezi tabulkami a vytvořit tak JOIN potřebný pro daný deskriptor popisující spojení dvou tabulek. Podobu tohoto JOINU bude mít uživatel možnost ovlivnit jeho přepsáním přímo v generátoru.
Součástí generovaného kódu bude také XML s mapováním pro `mw:DynamicList`. Toto mapování bude tedy moci uživatel nadefinovat přímo v generátoru – u každé tabulky a každého ze sloupců bude políčko pro zadání aliasu používaného v `mw:DynamicList`.
Generátor bude moci generovat i jednotlivé `DataService` třídy včetně filtračních parametrů – a to na základě databázového typu. Uživatel si opět bude moci pomocí zašktrtávátka určit, zdali se daný filtrační parametr vygeneruje, či nikoliv. Zároveň bude i u každého z filtrů políčko pro zadání aliasu filtru pro `mw:DynamicList`.
Veškeré generované třídy budou označeny jako `partial`, aby bylo možné je libovolně rozšiřovat a přesto mít možnost jejich generovanou část kdykoliv znovu přegenerovat.
Nastavení celého řešení (tedy všech tabulek, filtrů, sloupců, aliasů atd.) bude možné uložit pro pozdější editaci a případně přegenerování kódu. Toto

nastavení se bude ukládat do textového souboru v XML formátu, aby bylo možné tento soubor uchovávat ve verzovacím systému a snadno rozpoznávat změny či případně umožnit `merge` úprav datové vrstvy paralelně provedených dvěma uživateli.

2.2.12 Uchování dat pro potřeby redakčního systému

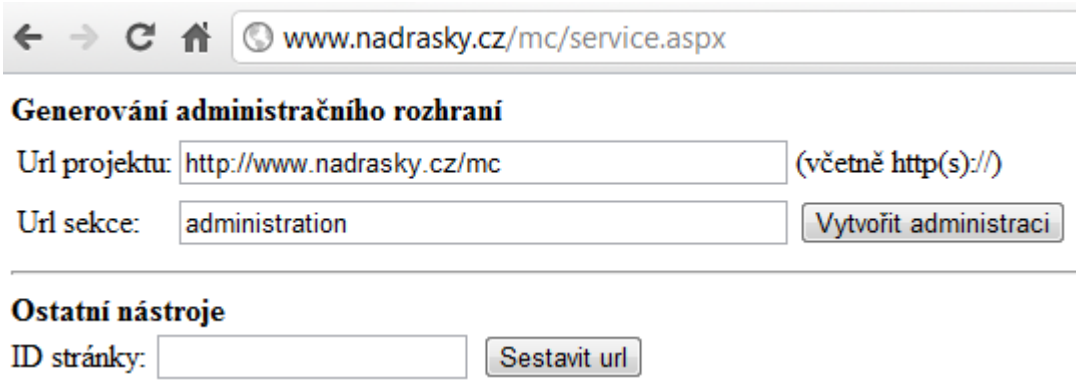
Posledním důležitým rozhodnutím je způsob, jakým budou uchovávána data potřebná pro redakční systém – tedy projekty, sekce a stránky. Zde ale opět jsme svázáni požadavkem na zpětnou kompatibilitu. V systému Magic2G jsou všechny tyto objekty uchovávány ve stejné databázi jako produkční data informačního systému a mají již pevně definovanou relační strukturu. Ačkoliv tato relační struktura zapadá do celkového návrhu databázové struktury M2G, nemusí všem vyhovovat – už jen z důvodu toho, že pojmenování tabulek i atributů je v českém jazyce. Z tohoto důvodu jsme zvolili kompromis, který spočívá v požadavku, že tabulky pro redakční systém sice musí být součástí cílového systému, jejich struktura a pojmenování však už může být libovolná. Pro napojení redakčního systému na databázi se tak bude muset vytvořit pouze mapování mezi názvy používanými redakčním systém a cílovou databází. Zároveň bude nutné při implementaci každého napojení na jinou databázi implementovat rozhraní pro operace `INSERT`, `UPDATE` a `DELETE` pro všechny entity redakčního systému.

3 Uživatelská dokumentace

Účelem kapitoly s uživatelskou dokumentací je seznámení HTML kodéra, tedy hlavního uživatele systému, jak s jeho funkčností a poskytovaným rozhraním tak s celkovou filozofií tvorby webových aplikací. Po jejím přečtení by měl být tento uživatel schopný umět správně navrhnout strukturu webu v prostředí redakčního systému a umět používat veškeré dynamické komponenty, které systém nabízí. Předpokládá, že čtenáře je seznámen s kapitolou 2.1 Řešení uživatelského prostředí a fungování redakčního systému. Mimo to se také předpokládá, že redakční systém byl již nasazen na konkrétní server a byly do něj doprogramovány veškeré dynamické komponenty a propojení s databází, ze které se mají čerpat vypisované informace. Implementace specifických dynamických komponent, stejně tak jako napojení na databázi a samotné vystavení na server jsou popsány v kapitole 4, která je určena pro programátory, jež budou systém rozšiřovat o nové dynamické objekty. Pro demonstrační účely celé této kapitoly byl systém nasazen na webový server pod výchozí URL adresou `http://www.nadrasky.cz/mc/`.

3.1 Webové administračního rozhraní

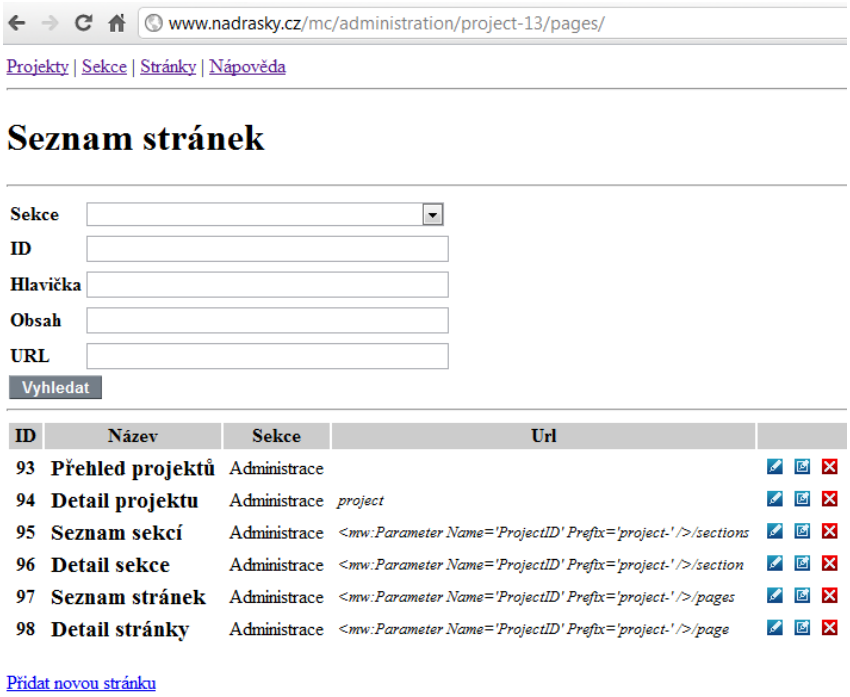
Tvorba a administrace webu se provádí z prostředí webového rozhraní, které je při prvním přístupu nejprve nutné vygenerovat. Generování tohoto rozhraní se provádí na stránce `service.aspx` (konkrétně tedy pod adresou `http://www.nadrasky.cz/mc/service.aspx`), která je vidět na obr. 14.



Obr. 14 Servisní stránka pro generování administračního rozhraní

Jediné, co je nutné pro vygenerování administrace zadat, je základní URL adresa webové aplikace včetně uvedení URL protokolu (konkrétně tedy `http://www.nadrasky.cz/mc`). Po vytvoření administrace systém automaticky přesměruje na výchozí stránku nově vytvořeného administračního rozhraní. Samotné administrační rozhraní se skládá ze třech stránek s výpisem projektů, sekcí a stránek, mezi kterými lze libovolně přecházet pomocí horního menu. Každý výpis obsahuje vyhledávací formulář a veškerou

potřebnou logiku pro vytváření, editaci a případně i mazání záznamů. Ukázka výpisu stánek je na obr. 15.

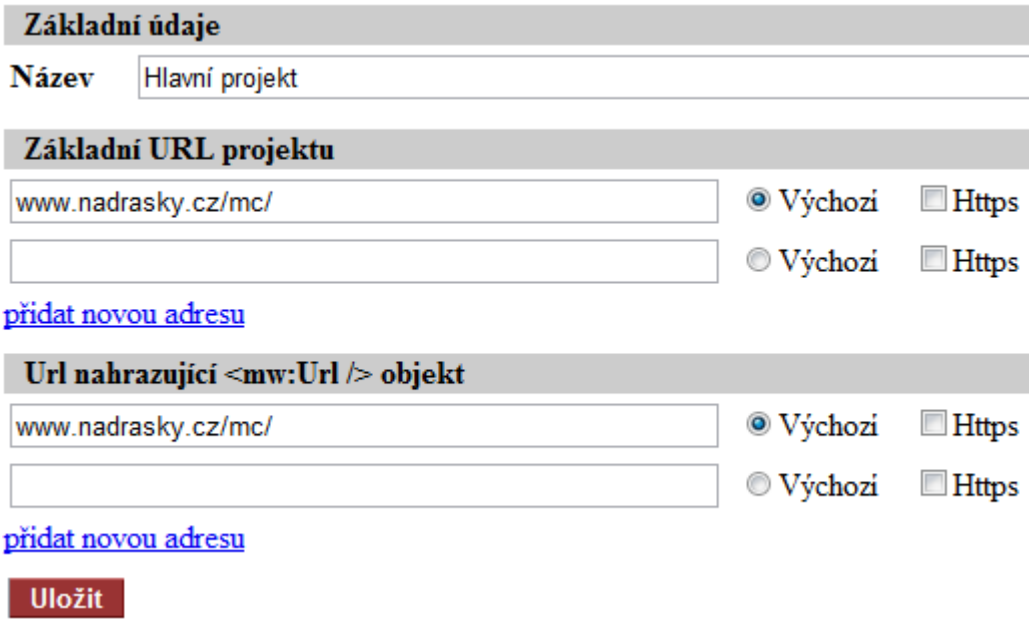


Obr. 15 Ukázka administračního rozhraní se seznamem stránek

Celé administrační rozhraní je definováno pomocí prostředků samotného redakčního systému. Tento přístup umožňuje si rozhraní libovolně přizpůsobit, či rozšířit jeho funkcionalitu. Má však jednu velkou nevýhodu – pokud si uživatel ať už chtěně nebo nechtěně toto administrační rozhraní rozbije, nezbude mu jiná možnost, než si ho vygenerovat znovu nebo vrátit nechtěné úpravy přímo v databázi, kde je redakční systém uložen. Z tohoto důvodu je lepší případné komplexnější úpravy řešit například vytvořením kopie stránky, aby bylo možné v případě chyby tuto opravit, a až po finálních a otestovaných úpravách přepsat obsah původní stránky novým.
V následujících podkapitolách budou popsány editační formuláře pro projekty, sekce a stránky a zároveň důkladněji rozebrány veškeré jejich vlastnosti již dříve zmíněné v kapitole 2.1.

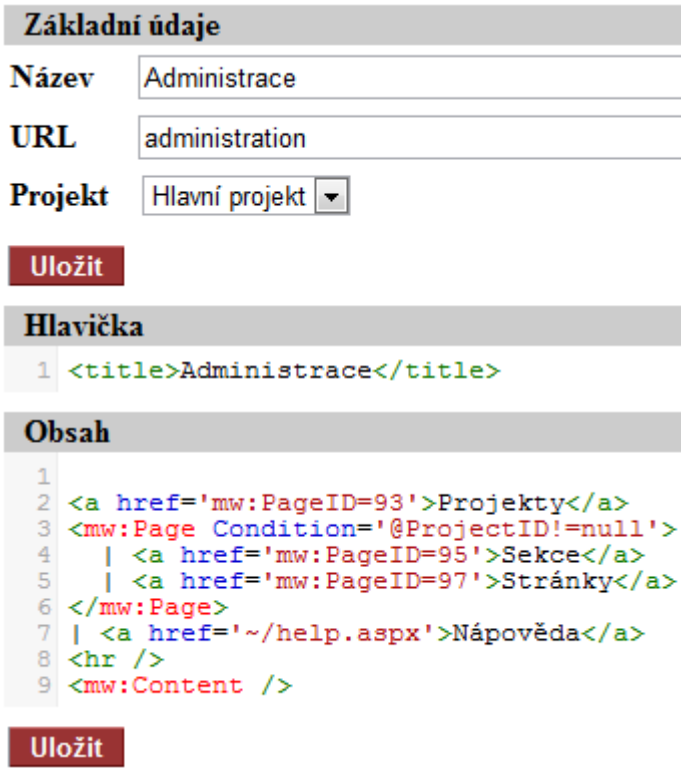
3.1.1 Editační formuláře

Jak už bylo řečeno v kapitole 2.1, projekt slouží pro nadefinování URL adresy celého webu (nebo alespoň jeho části). Editační formulář (obr. 16) obsahuje mimo povinného názvu projektu také dva seznamy s URL adresami projektu. Oba z těchto seznamů obsahují URL adresy bez uvedení protokolu (implicitně se předpokládá protokol HTTP, pokud má být URL dostupná na protokolu HTTPS, je nutné zatrhnout příslušné zaškrtávátko). Proč jsou seznamy dva a k čemu slouží je důkladněji popsáno v kapitole 3.1.2 Správa URL adres.



Obr. 16 Editační formulář pro projekt

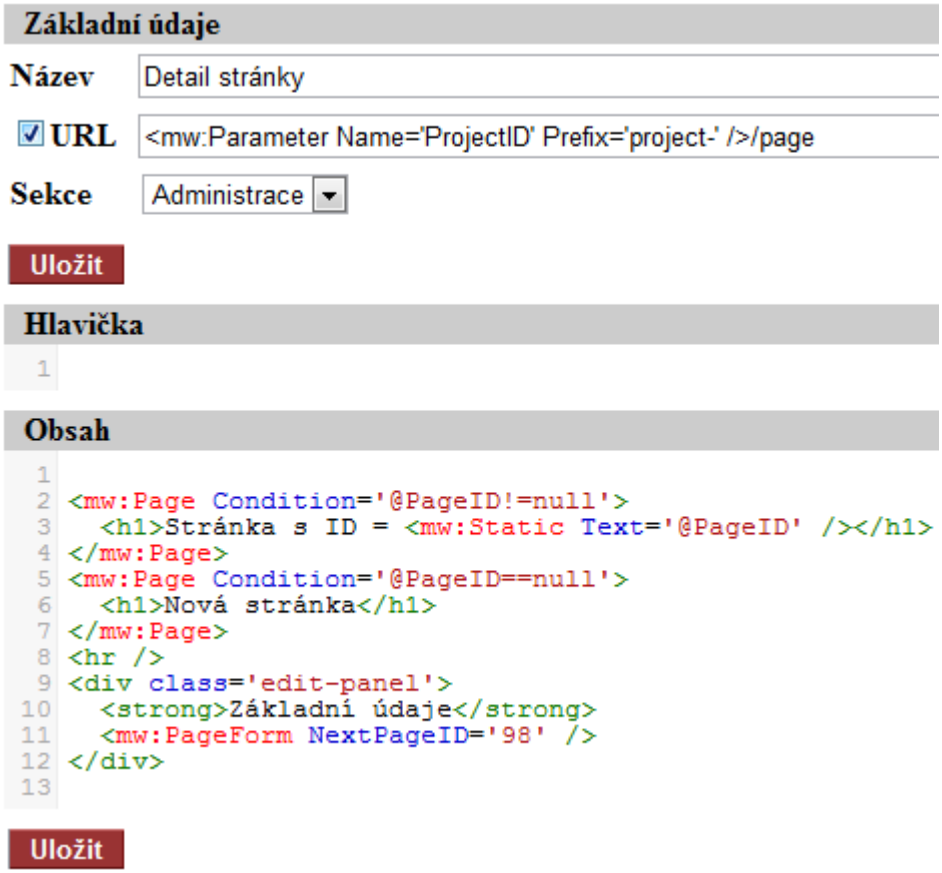
Editační formulář pro sekce z obr. 17, se skládá ze třech částí. První z nich slouží k zadání základních údajů – název, příslušnost k projektu a hlavně URL, o níž bude více napsáno v kapitole 3.1.2.



Obr. 17 Editační formulář pro sekce

Uložení změn provedených ve formuláři se provádí jedním ze dvou tlačítek Uložit. Podobným způsobem je rozložen i formulář pro editaci stránky (obr. 18). V obou je mimo základních údajů možné editovat i obsah HTML

hlavičky, která se vkládá dovnitř tagu `head`. Obsah sekce je naopak vkládán do těla HTML stránky (tedy dovnitř tagu `body`) a předpokládá se, že HTML kodér do jeho obsahu vloží i dynamický objekt `mw:Content`, jenž se při zpracování obsahu sekce nahradí obsahem aktuální stránky – o tomto objektu se lze více dočíst v kapitole 3.4.2. Pro editaci hlavičky a obsahu byla zvolena volně dostupná komponenta `CodeMirror` (3), která zpříjemňuje editaci HTML obsahu podporou pro zvýrazňování syntaxe či vyhledávání nebo nahrazování pomocí regulárních výrazů (klávesová zkratka `Ctrl+F` pro hledání, `Ctrl+G` pro nalezení dalšího výskytu a nakonec `Ctrl+Shift+F` pro nahrazování). Formulář stránky se od toho pro sekce liší v jedné podstatné věci – před polem pro zadání URL je volba, která umožňuje vytvořit stránku, která nebude mít URL a nebude tak dostupná přímo zadáním URL adresy a tím pádem na ni nemůže být ani odkazováno. Toto se hodí v případech, kdy si chce kodér vytvořit nějakou část stránky a tuto pak vkládat na různá místa webu a přitom neduplikovat obsah. Příkladem typického použití může být blok[7] obsahujícího informace například s copyrightem, jenž má být součástí patičky každé sekce.



Obr. 18 Editační formulář pro stránky

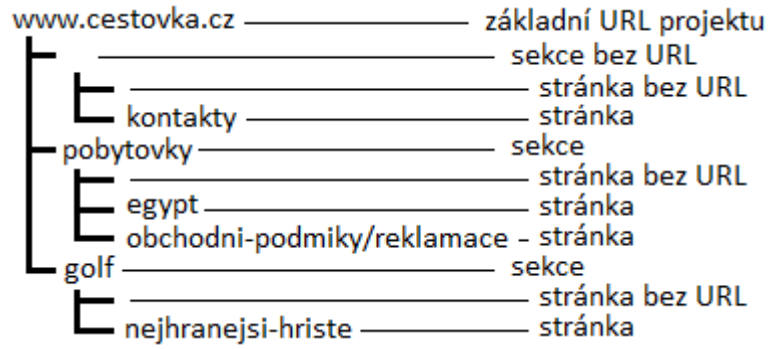
[7] Blok je označení pro stránku bez URL – tedy bez zatrženého zaškrtávátka před polem pro zadání URL adresy stránky.

3.1.2 Správa URL adres

Princip správy URL adres bude nejlepší vysvětlit na dvou konkrétních příkladech. První bude jednodušší a bude vyžadovat vytvoření webu, který bude dostupný pod jedním doménovým jménem. Druhý příklad bude mít za cíl vytvořit web, kde sekce budou muset definovat doménu třetího řádu.
Nejprve tedy jednodušší varianta webu složeného z následujících stránek:

- `www.cestovka.cz/` – výchozí stránka
- `www.cestovka.cz/kontakty/` - stránka s kontakty o cestovní kanceláři
- `www.cestovka.cz/pobytovky/` - stránka s obecnými informacemi o pobytových zájezdech
- `www.cestovka.cz/pobytovky/egypt/` - stránka s informacemi o pobytových zájezdech v egyptských letoviscích
- `www.cestovka.cz/pobytovky/obchodni-podminky/reklamace/` - stránka o podmínkách uplatnění případné reklamace
- `www.cestovka.cz/golf/` - informace o golfových zájezdech
- `www.cestovka.cz/golf/nejhranejsi-hriste/` - seznam nejhranějších hřišť

V tomto případě je jasné, že základní URL celého webového projektu bude `www.cestovka.cz`. Tato adresa zároveň ale musí sloužit i jako adresa výchozí stránky z výchozí sekce. V případě, kdy má mít sekce stejnou adresu jako projekt, je nutné, aby svoji URL neměla vyplněnou. To samé platí i pro stránku libovolné sekce. Celková struktura a nastavení URL adres je vidět na následujícím obrázku:



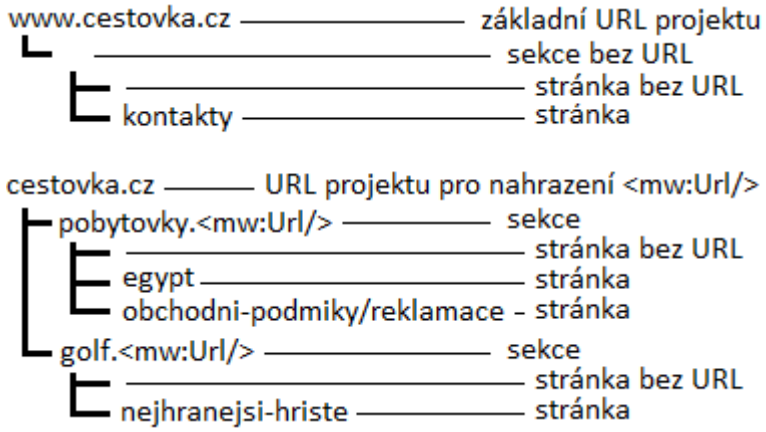
Obr. 19 Příklad jednoduché struktury URL sekcí a stránek

Druhá varianta vyžaduje definovat jednotlivé sekce jako domény třetího řádu, tedy například pro stejný obsah jako v případě jednoduché varianty by se jednalo o následující strukturu URL:

- `www.cestovka.cz/`
- `www.cestovka.cz/kontakty/`
- `pobytovky.cestovka.cz/`
- `pobytovky.cestovka.cz/egypt/`
- `pobytovky.cestovka.cz/obchodni-podminky/reklamace/`

- `golf.cestovka.cz/`
- `golf.cestovka.cz/nejhranejsi-hriste/`

V této variantě je nutné využít druhého seznamu URL adres projektu, jak je nastíněno na obr. 20.



Obr. 20 Příklad struktury URL sekcí jako domény 3. řádu

Z obrázku je patrné, že všechny sekce, jenž ve své adrese obsahují XML značku `mw:URL`, jsou zařazeny pod URL projektu z druhého seznamu – tedy pod „URL nahrazující mw:Url blok“. První seznam adres tedy slouží k nadefinování URL pro sekce, které neurčují doménové jméno, zatímco druhý seznam je právě pro sekce, které upravují doménovou část URL adresy. Každý seznam zároveň může obsahovat více než jednu URL adresu a to z toho důvodu, aby bylo možné přistoupit k webu z více domén – například `www.nejlepsicestovka.cz` může směrovat na stejný web jako `www.cestovka.cz`. Aby však nebyl takovýto web internetovými vyhledávači penalizován kvůli duplicitnímu obsahu, je nutné určit, která URL je ze seznamu výchozí. Systém poté při požadavku na zobrazení jiné než výchozí URL adresy jednoduše přesměruje na tu výchozí s HTTP stavem `301 – moved pernametly`.

3.1.3 Aplikace MagicCoding

Aplikace MagicCoding slouží pro snadnou editaci obsahu stránek a sekcí v libovolném desktopovém editoru. Její instalace spočívá v pouhém rozbalení souborů z `MagicCoding.zip` do libovolného adresáře, kde má uživatel právo zápisu. Její spuštění pomocí souboru `MagicCoding.exe` zobrazí ikonu v oznamovací oblasti, která slouží jako hlavní menu aplikace (menu se vyvolá pomocí pravého tlačítka myši). Při prvním spuštění je nejprve nutné zaregistrovat URL protokol `MagicCoding` pomocí položky „Register URL Protocol“ v hlavním menu. Registrace vyžadující administrátorská oprávnění je důležitá proto, aby systém věděl, kterou aplikaci má spustit při požadavku na editaci stránky nebo sekce. V případě přesunutí aplikace na jiné umístění, je nutné provést registraci znovu. Při prvním spuštění je zároveň nutné

nadefinovat cestu a parametry editoru zamýšleného pro editaci. Toto nastavení se rovněž provádí z hlavního menu pomocí položky „Open Settings“. Pokud je aplikace spuštěna, je v tu dobu možné ve webovém administračním rozhraní využívat tlačítka pro editaci stránky nebo sekce v externím editoru (viz obr. 21).

Obr. 21 Tlačítko pro editaci stránky / sekce v externím editoru

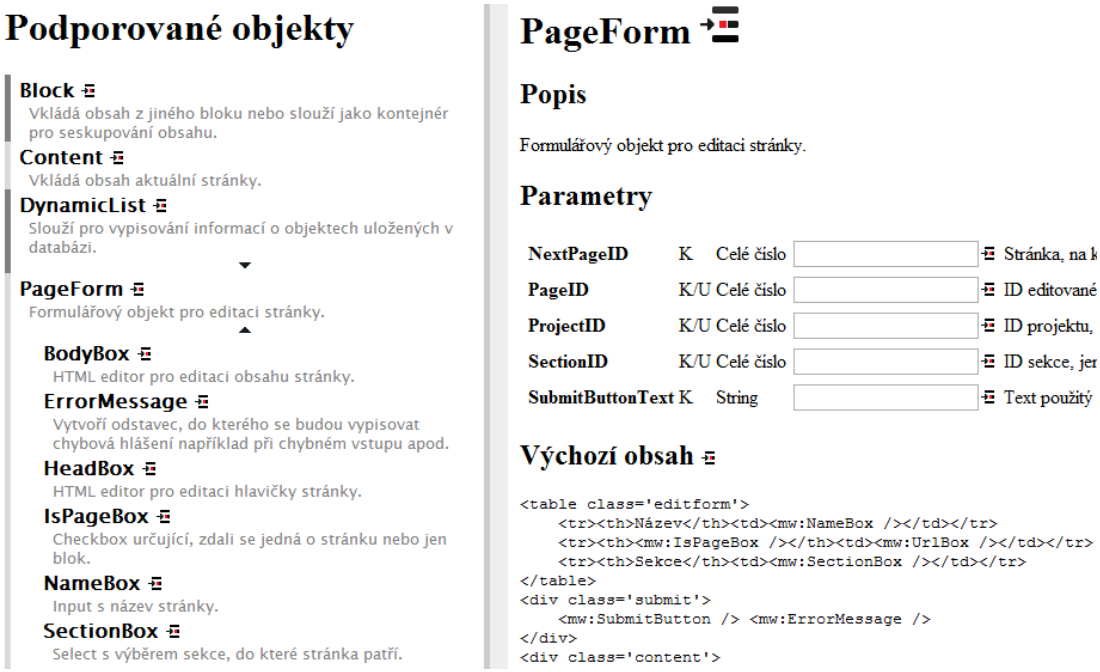
Jeho stisknutím (prohlížeč bude vyžadovat povolení) se stáhne hlavička a obsah stránky / sekce do souboru na disk do adresáře `MonitoredFolder` a tento soubor se otevře v požadovaném desktopovém editoru. Jakékoliv úpravy, které se provedou v souboru, jsou automaticky po uložení odeslány na server, o čemž vždy aplikace informuje bublinou v oznamovací oblasti. Pro správné fungování je nutné zanechat vložené oddělovače pro hlavičku stránky a sekce. V opačném případě by systém nedokázal rozeznat, který kód je určen do hlavičky a který do obsahu.

3.2 Dynamické objekty

Celý redakční systém stojí na možnosti vkládat dynamický obsah do webových stránek. Veškerá tato funkcionalita je umožněna pomocí tzv. dynamických objektů, jejichž návrhem se zabývala kapitola 2.1.2 Statický a dynamický obsah stránek. Dynamické objekty lze používat jak v hlavičce, tak obsahu libovolné stránky či sekce. Jejich použití spočívá v pouhém vložení příslušného XML tagu přímo do zdrojového kódu mezi HTML značky. Aby systém dokázal dynamické objekty odlišit od klasického obsahu, je nutné před každý název uvádět prefix „`mw:`“. Při vlastním renderování stránky jsou všechny dynamické objekty nahrazeny příslušným obsahem, pro něhož byly určeny. Chování každého dynamického objektu lze ovlivňovat pomocí parametrů, které se zadávají jako XML atributy k příslušnému tagu. Systém umožňuje ohraničovat hodnotu parametrů jak pomocí uvozovek, tak pomocí apostrofů. Na rozdíl ale od XML standardu je umožněno tyto ohraničující znaky vkládat pomocí jejich zdvojení do vnitřku příslušné hodnoty. Některé objekty dovolují nastavovat hodnotu svých parametrů i z hodnoty stejně pojmenovaného parametru uvedeného v URL adrese. Většinou se jedná například o filtrační parametry, které nějakým způsobem omezují výpis nebo jinak ovlivňují chování objektu.

Seznam všech objektů včetně jejich popisu je možné najít v automaticky generované nápovědě, která je vždy dostupná na stránce `help.aspx` a na níž také vede odkaz „Nápověda“ z webového administračního rozhraní. Ukázku stránky s nápovědou lze najít na obr. 22. V levém menu systém při zobrazení načte všechny dynamické objekty, které rozpoznává, spolu s jejich krátkým

popisem. V případě, kdy dynamický objekt podporuje rozpoznávání vnořených objektů, je možné tyto zobrazit pomocí rozevírací šipky pod tímto objektem. Pro každý objekt lze zobrazit podrobnou nápovědu v pravé části okna kliknutím na název objektu. V detailu je možné najít veškeré parametry objektu spolu s jejich popisem a způsobem použití (vedle jejich typu i například to, zdali systém umožňuje předat hodnotu jen z kódu nebo i z URL parametru). Mimo to je zde také zobrazen výchozí obsah u těch objektů, které podporují vnořený obsah (jako je například výchozí obsah objektu `mw:PageForm` z obr. 22).



Obr. 22 Nápověda k dynamickým objektům

Systém nápovědy neplní pouze informační funkci. Pomocí značky z obr. 23 je možné metodou Drag & Drop rychle „přetáhnout“ konkrétné obsah do otevřeného editoru.

Obr. 23 Drag & Drop značka pro rychlou tvorbu kódu

Například přetažením ze značky vedle názvu dynamického objektu je na cílové místo v editoru vložen kód pro rychlé použití dynamického objektu (například pro objekt `mw:Block` to je kód `<mw:Block PageID=”” />`. Stejně tak je umožněno rychlé nastavování všech parametrů objektů, kdy stačí pouze vyplnit hodnotu a opět přetáhnout příslušnou značku na místo určení, kam je zkopírován text obsahující název parametru včetně jeho hodnoty. Systém v tomto případě umí automaticky nahradit případné apostrofy nebo uvozovky

jejich zdvojením a uživatel tak nemusí myslet na syntakticky špatně zadané hodnoty.

3.3 Ukázková webová aplikace

V následujícím textu bude popsána funkčnost celého systému pomocí tvorby jednoduchého webového portálu pro výpis zájezdů z databáze použité jako ukázkové napojení na systém Magic2G. Portál by se měl skládat ze třech stránek:

- Výchozí stránka by měla obsahovat stromovitý seznam všech zemí a destinací, v nichž existuje nějaký produkt. Z tohoto seznamu by se mělo jít dát přejít na seznam všech hotelů v dané zemi nebo destinaci (následující stránka).
- Seznam všech zájezdů s jednoduchým filtračním formulářem a možností libovolného seřazení. Z každého hotelu bude možné překliknout se na stránku s jeho detailem.
- Detail zájezdu bude tvořit jednoduchý výpis veškerých informací, které jsou o něm v databázi vedeny.

3.4 Základní dynamické objekty

Kapitoly níže si kladou za cíl popsat dynamickou funkcionalitu systému, která je dosažena pomocí základních dynamických objektů, jež jsou součástí každého nasazení a jež tvoří základ celého redakčního systému.

3.4.1 Odkazy a mw:PageID

V redakčních systémech není zvykem odkazy zadávat natvrdo pomocí URL adresy a systém MagicContent není výjimkou. V případě změny adresy stránky by se totiž musely všechny linky, které na ní vedou, přepsat. Obyčejné odkazy na stránky se tak tvoří jednoduchou nestandardní značkou `mw:PageID=id_stránky` tak, jak je ukázáno na obr. 24. Zpracování obsahu stránky totiž předchází proces, jenž v kódu najde všechny výskyty tohoto tvaru a nahradí je příslušnou URL adresou.

```
<a href="mw:PageID=99">Hlavní stránka</a> |
<a href="mw:PageID=100">Seznam všech hotelů</a>
```

Obr. 24 Způsob tvoření odkazů na stránky

3.4.2 Obsah vkládaný pomocí mw:Content a mw:Block

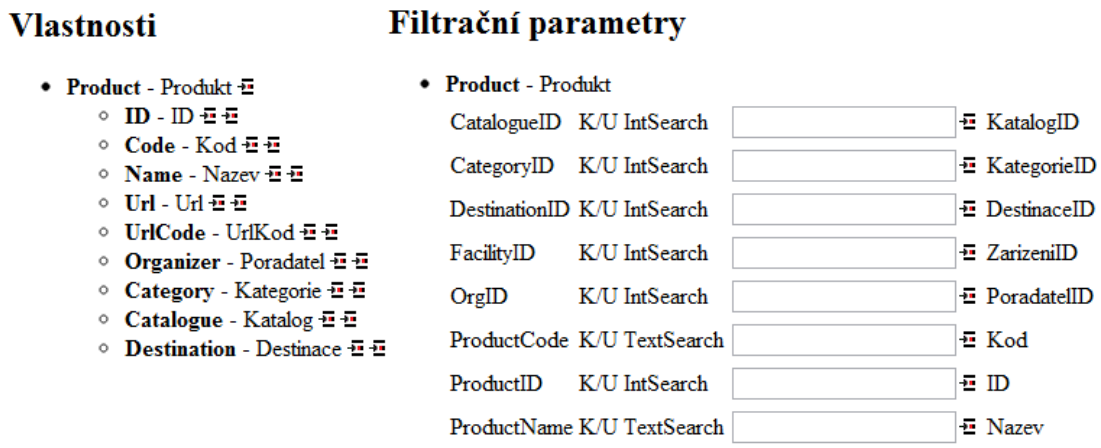
Dynamický objekt `mw:Content` se při zpracování obsahu nahrazuje obsahem aktuální stránky (tedy té, která byla rozpoznána z URL). Jeho použití je tedy určeno pouze pro sekce a typicky pouze jednou.

V případě, kdy mají být nějaké části webu používané na více místech, je více než vhodné umístit tento kód do nějakého bloku (tedy stránky bez URL) a

tento kód pak pomocí objektu `mw:Block` vkládat na libovolné místo. Objekt má jediný parametr `PageID`, který očekává ID bloku, jenž má za sebe nahradit.

3.4.3 Dynamický obsah pomocí mw:DynamicList

Nejdůležitějším a zároveň i nejsložitějším dynamickým objektem je nepochybně `mw:DynamicList`, který je určen pro výpis informací z databáze. V prvním kroku, je vždy důležité si rozmyslet, ze které tabulky se mají informace vypisovat, jelikož je to nejpodstatnější parametr (jeho název je `Source`), který `mw:DynamicList` vyžaduje. Seznam všech tabulek, které je možné pomocí `mw:DynamicList` vypsat lze najít v nápovědě k tomuto objektu v seznamu vlastností (relevantní část této nápovědy je na obr. 25.). Seznam vlastností (co jsou, viz níže) je strukturován právě pod tabulky (seznam se rozevře po kliknutí na název tabulky), ze kterých lze dané vlastnosti vypsat. Tučně jsou uvedeny názvy používané pro `mw:DynamicList` zatímco obyčejným písmem jsou vypsány ekvivalentní názvy použité přímo v databázi.



Obr. 25 Nápověda pro dynamický objekt mw:DynamicList

Ukázková webová aplikace se skládá celkem ze třech stránek a na každé z nich mají být nějakým způsobem zobrazeny informace z databáze. Začneme tedy trochu nezvykle od stránky s detailem hotelu. Tato stránka by se měla skládat z nadpisu tvořeného názvem hotelu, pod kterým by měly být informace o destinaci, ve které se hotel nachází, o katalogu, ve kterém je veden, nebo i o pořadateli, jenž zájezd do daného hotelu organizuje. `mw:DynamicList` je dynamickým objektem, jenž vyžaduje obsah – je ho proto nutné vždy vkládat jako párový XML tag. Pro tento účel lze použít Drag & Drop značku vedle názvu tabulky, ze které chceme informace vypisovat. Obsahem objektu je libovolná HTML šablona, která může být doplněna libovolnými dynamickými objekty, které jsou podporovány. Nejčastějším dynamickým objektem používaným v šablonách `mw:DynamicListu` je

nepochybně `mw:Field`. Tento objekt očekává jako parametr `Type` název vlastnosti, jejíž hodnotu má vypsat. Všechny vlastnosti, které lze pro danou tabulku použít je možné nají v nápovědě (viz obr. 25), kde je možno znovu využít Drag & Drop značky (první slouží k přetáhnutí pouze názvu vlastnosti, zatímco druhá v sobě obsahuje objekt `mw:Field` s přednastaveným parametrem Type na danou vlastnost). Některé vlastnosti – například pořadatele – lze rozbalit a najít pod ní veškeré dostupné vlastnosti týkající se pořadatele (některá z nich – typicky název – je vždy výchozí a není tak vždy nutné specifikovat konkrétní podřízenou vlastnost a je možné tak vložit například vlastnost `Destination` bez nutnosti specifikovat `Destination.Name`). Celý kód detailu stránky s hotelem tedy může vypadat jako na obr. 26.

```
<mw:DynamicList Source="Product" >
  <h1><mw:Field Type="Name" /></h1>
  <div>
    <strong>Destinace:</strong> <mw:Field Type="Destination" />
  </div>
  <div>
    <strong>Katalog:</strong> <mw:Field Type="Catalogue" />
  </div>
  <div>
    <strong>Pořadatel:</strong><br />
    Název: <mw:Field Type="Organizer.Name" /><br />
    ICO: <mw:Field Type="Organizer.ICO" /><br />
    DIC: <mw:Field Type="Organizer.DIC" />
  </div>
</mw:DynamicList>
```

Obr. 26 Kód stránky s detailem hotelu

Dalším krokem k vytvoření ukázkového portálu je stránka se seznamem všech hotelů. Na rozdíl ale od stránky s detailem, která vypisovala informace pouze o jednom záznamu, bude zde nutné vytvořit a označit kus kódu, který bude sloužit jako šablona pro tvorbu HTML pro jednotlivé záznamy. K tomuto účelu slouží dynamický objekt `mw:Template`, jehož obsah je právě tato šablona. Dynamický obsah pro výpis hotelů může vypadat například tak, jak je vidět na obr. 27. Jako v předchozím případě i zde jsou použity dynamické objekty `mw:Field` pro výpis konkrétních vlastností hotelu, nicméně zde jsou umístěni uvnitř dynamického objektu `mw:Template`, který se postará o to, aby jeho obsah byl zduplikován tolikrát, kolik obdrží od svého nadřazeného objektu záznamů k vyrenderování. Za povšimnutí zde ještě stojí způsob, kterým se dá uvnitř obsahu `mw:DynamicList` vytvářet odkazy na jiné stránky. K tomuto účelu slouží parametr `NextPageID`, kterému stačí předat ID stránky a `mw:Field` obalí vypisovanou hodnotu HTML tagem pro vytvoření odkazu. Naopak parametr `Parameters` slouží k definování, jaké URL parametry by se měly stránce předat. Na obrázku je tak předána hodnota z vlastnosti `ID`

pomocí parametru s názvem `ProductID`. Možné syntaxe pro definici jednoho parametru jsou následující:

```
@NAZEV_PARAMETRU=[HODNOTA]
@NAZEV_PARAMETRU=NAZEV_VLASTNOSTI
@NAZEV_PARAMETRU=NAZEV_VLASTNOSTI:FORMAT
```

`HODNOTA` je libovolný text, který má být použit jako hodnota parametru, `NAZEV_VLASTNOSTI` je libovolný název vlastnosti, který je možné vypsat z dané tabulky a konečně `FORMAT` je formátovací řetězec použitý pro zformátování hodnoty (lze využít libovolný formátovací řetězec, který lze použít v .NET metodě `String.Format`). Na obrázku byl využit jeden ještě nejmenovaný dynamický objekt – `mw:Static`. Jeho úloha spočívá v pouhém výpisu statického textu a případně jeho obalení HTML odkazem. `mw:Static` navíc umí podobně jako `mw:Template` zpracovat stejný obsah akorát s tím rozdílem, že neslouží k opakování.

```
<mw:DynamicList Source="Product">
  <table>
    <thead><tr>
      <th>Název hotelu</th>
      <th>Kategorie</th>
      <th>Destinace</th>
    </tr></thead>
    <tbody>
      <mw:Template>
        <tr>
          <td>
            <mw:Field Type="Name"
              NextPageID="101" Parameters="@ProductID=ID" />
          </td>
          <td><mw:Field Type="Category" /></td>
          <td><mw:Field Type="Destination" /></td>
          <td>
            <mw:Static Text="Detail"
              NextPageID="101" Parameters="@ProductID=ID" />
          </td>
        </tr>
      </mw:Template>
    </tbody>
  </table>
</mw:DynamicList>
```

Obr. 27 Šablona mw:DynamicList pro výpis hotelů

Poslední stránkou, která ještě chybí, je hlavní stránka se stromovitou strukturou destinací. Ještě než ale ukážeme kód pro její vytvoření, je nutné popsat, jak `mw:DynamicList` vlastně funguje. Pro správné vytváření strukturovaných výpisů je to totiž nesmírně důležité. Zdrojem dat pro `mw:DynamicList` je vždy výstup z nějakého SQL dotazu, který byl poskládán na základě všech vlastností, které jsou v jeho obsahu použity. To znamená, že tato struktura je vždy plochá – tedy pouhý seznam záznamů se všemi

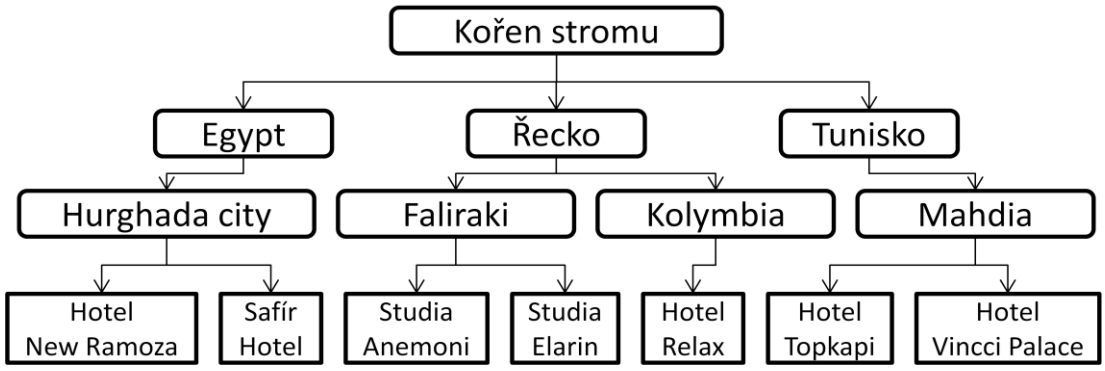
vlastnostmi. Plochá struktura se ale často absolutně nehodí pro strukturované výpisy, jakým může být například stromový výpis všech destinací. Z toho důvodu obsahuje `mw:DynamicList` podporu pro tzv. grupování záznamů. Grupování spočívá v převedení ploché struktury seznamu na stromovitou podle vlastností, které jsou předmětem nastavení. A protože grupování přímo souvisí i s řazením záznamů, nastavuje se obojí pomocí parametru `SortBy`. Tento parametr očekává seznam vlastností, podle kterých se má řadit, a u těch, podle kterých se má zároveň i vytvářet stromovitá struktura, stačí uvést za názvem `:G`. Například nastavení

```
SortBy="Destination.Lvl2.Name:G,Destination.Name:G,Name:G"
```

definuje seznam hotelů, který je setříděn primárně podle názvu země, poté podle názvu destinace a následně podlá názvu hotelu. Zároveň je nastaveno, že se podle všech těchto vlastností má i grupovat. Celkově je toto nastavení interpretováno tak, že se plochá struktura záznamů seřadí podle všech uvedených vlastností a následně se z této struktury vytvoří strom. Na první úrovni pod kořenem budou uzly, jež každý obsahuje jinou hodnotu vlastnosti `Destination.ID` a na další úrovni pak budou uzly obsahující každý jinou hodnotu vlastnosti `ID`. Znázornění tohoto postupu je na následujících dvou obrázcích:

Destination.Lvl2.Name	Destination.Name	Name
Egypt	Hurghada city	Hotel New Ramoza
Egypt	Hurghada city	Safír Hotel
Řecko	Faliraki	Studia Anemoni
Řecko	Faliraki	Studia Elarin
Řecko	Kolymbia	Hotel Relax
Tunisko	Mahdia	Hotel Topkapi
Tunisko	Mahdia	Hotel Vincci Nour Palace

Obr. 28 Původní plochá struktura vrácená databází



Obr. 29 Stromová struktura záznamů určená pro výpis

Ačkoliv se z obrázku může zdát, že vnitřní uzly obsahují jen vlastnost, dle které byly vytvořeny, opak je pravdou. Každý uzel (včetně kořene) je totiž

vždy klonem svého nejlevějšího potomka. `mw:DynamicList` s touto strukturou zachází tak, že vždy vezme kořen stromu a předá ho každému dynamickému objektu ve svém obsahu. `mw:Field` tak například vypisuje informaci právě z toho uzlu, který mu byl předán. `mw:Template` naopak funguje tak, že vezme aktuální uzel, a všem dynamickým objektům, které se v šabloně vyskytují, předá přímé potomky aktuálního uzlu.
Nyní je tedy možné přejít k tvorbě stromové struktury požadované v ukázkovém portálu:

```
<mw:DynamicList Source="Product"
  SortBy="
          Destination.Lvl1,Destination.Lvl1.ID:G,
          Destination.Lvl2,Destination.Lvl2.ID:G,
          Destination.Lvl3,Destination.Lvl3.ID:G
  ">
  <mw:Template>
    <div style='float:left;'>
      <h1>
        <mw:Field Type="Destination.Lvl1" NextPageID="100"
          Parameters="@DestinationRootID=Destination.Lvl1.ID" />
      </h1>
      <ul>
        <mw:Template>
          <li>
            <h2>
              <mw:Field Type="Destination.Lvl2" NextPageID="100"
                Parameters="@DestinationRootID=Destination.Lvl2.ID" />
            </h2>
            <ul>
              <mw:Template>
                <li>
                  <mw:Field Type="Destination.Lvl3" NextPageID="100"
                    Parameters="@DestinationRootID=Destination.Lvl3.ID" />
                </li>
              </mw:Template>
            </ul>
          </li>
        </mw:Template>
      </ul>
    </div>
  </mw:Template>
</mw:DynamicList>
```

Obr. 30 mw:DynamicList pro výpis strukturovaných dat

V nastavení stojí za povšimnutí způsob, kterým je nastaven parametr `SortBy` – vždy je napřed seřazeno podle názvu destinace a teprve poté grupováno podle jejího `ID`. Tímto způsobem bude zaručeno, že i v případě dvou destinací, které například nedopatřením budou mít stejný název, budou obě ve výpisu figurovat jako dva odlišné záznamy.
Poslední větší funkcí, kterou `mw:DynamicList` disponuje je možnost stránkovaných výpisů, které se nastavuje pomocí dvou parametrů: `PageSize`

a `PagingLevel`. První určuje počet záznamů na stránku, zatímco druhý určuje úroveň stromu, ve které se má stránkovat, tj. na které úrovni stromu se mají počítat záznamy pro stránkování. Například pro strom zobrazený na obr. 29 by v případě stránkování po dvou záznamech na druhé úrovni byly celkem stránky dvě (první stránka Hrughada a Faliraki, druhá Kolymbia a Mahdia). Naopak v případě stránkování na třetí úrovni by stránky byly 4 (po dvou hotelech na každé stránce).
Při nastaveném stránkování je nutné také uživateli nabídnout prostředek, jak toto stránkování řídit – tedy odkazy pro přechod na další či předchozí stránky. Pro tento účel existuje dynamický objekt `mw:Paging`, jehož možnosti nastavení spolu s dvěma příklady použití lze najít v systému nápovědy.

3.4.4 mw:SearchForm

Nejtypičtějším požadavkem uživatelů webových stránek a portálů, které obsahují nějaké výpisy záznamů, je umožnit jejich filtrování pomocí vyhledávacích formulářů. Redakční systém MagicContent proto nabízí jednoduchý prostředek, jak takovéto vyhledávací formuláře vytvořit bez nutnosti jejich programování. Stačí využít dynamického objektu `mw:SearchForm`.
Jeho funkcionalita a způsob nastavení bude opět popsán na příkladě použití v ukázkovém portálu na stránce s výpisem všech hotelů (obr. 31).

```
<mw:SearchForm>

  Název hotelu:
    <mw:Input Parameter="ProductName" />

  Typ zájezdu:
    <mw:Select Parameter="ProductCode">
      <mw:Option Value="" Text="Vše" />
      <mw:Option Value="svatba" Text="Svatby" />
    </mw:Select>

  Katalog:
    <mw:DynamicSelect Parameter="CatalogueID"
      Source="Catalogue" CatalogueName="léto"
      ValueFieldType="ID"
      TextFieldType="Name" />

  <mw:SubmitButton Text='Vyledat' />

</mw:SearchForm>
```

Obr. 31 Příklad nastavení vyhledávacího formuláře

Podobně, jako v klasickém HTML, je nutné všechny formulářové objekty uzavřít do jednoho formuláře – tedy do dynamického objektu `mw:SearchForm`. Tento objekt rozpoznává celkem čtyři podřízené objekty: `mw:Input`,

mw:Select, mw:DynamicSelect a mw:SubmitButton. První z nich slouží k vytvoření obyčejného pole (ekvivalent HTML tagu input s typem text), kterému stačí nastavit název parametru a mw:SearchForm se v případě odeslání formuláře postará o nastavení uživatelem zadané hodnoty do parametru s určeným názvem. Druhým objektem je mw:Select, který naopak slouží k vyhledávání podle předem definovaného statického seznamu hodnot. Nejzajímavějším objektem pro vyhledávání je ale nepochybně mw:DynamicSelect, který slouží pro vyhledávání dle dynamicky generovaného seznamu hodnot z databáze. Je založen na podobném principu jako mw:DynamicList – tedy stačí určit tabulku, ze které se mají hodnoty vybírat a názvy vlastností pro textovou a hodnotovou reprezentaci záznamů. Na obrázku s příkladem pro vyhledávání hotelů je definován mw:DynamicSelect, který zobrazuje seznam záznamů z tabulky Catalogue a jako vyhledávací hodnota je použita vlastnost ID, zatímco jako textová hodnota je použita vlastnost Name. Jelikož je mw:DynamicSelect založen na stejném základě jako mw:DynamicList, lze tak využít libovolného filtračního parametru pro danou zdrojovou tabulku tak, jak to umožňuje mw:DynamicList. Například pokud by bylo žádoucí omezit seznam katalogů jen na ty, jež obsahují v názvu slovo „léto“, stačilo by přidat do definice mw:DynamicSelect parametr CatalogName=“léto“.

3.5 Šablonovací systém pro formuláře

Jedním z cílů z úvodu byla možnost používat formuláře v jejich nejtypičtější podobě bez nutnosti definování, jak a co přesně mají obsahovat, aby bylo jejich použití co možná nejrychlejší a přímočaré. Zároveň jsme ale chtěli docílit možnosti, aby si uživatel mohl tento výchozí obsah přizpůsobit. Ukázková aplikace žádný takovýto formulář nepotřebuje, nicméně systém lze o formuláře s takovouto funkcionalitou libovolně rozšiřovat, a tak je nutné vysvětli princip, jakým je výše popsané funkcionality dosaženo.
Příkladem formulářů, které tuto funkcionalitu mají implementovánu, jsou editační formuláře použité pro vytvoření administračního rozhraní – tedy mw:ProjectForm, mw:SectionForm a mw:PageForm. Na obr. 39, je zobrazen obsah stránky s definicí editačního formuláře pro editaci sekce – tedy mw:SectionForm.

```
<div class='edit-panel'>
  <strong>Základní údaje</strong>
  <mw:SectionForm NextPageID='96' />
</div>
```

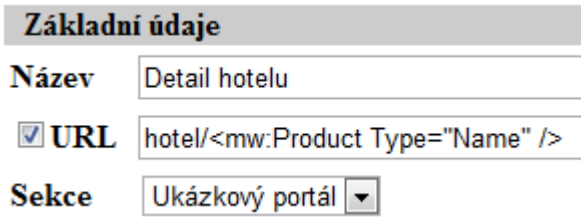
Obr. 32 Použití objektu mw:SectionForm s výchozím obsahem

Je vidět, že tento dynamický objekt nemá definován žádný obsah a přesto se formulář zobrazuje – to je umožněno systémem tzv. výchozího obsahu.

Každý objekt, který lze použít jako párovou XML značku - lze mu tedy nadefinovat nějaký obsah, může (ale také nemusí) definovat svůj výchozí obsah. Tento výchozí obsah se pak používá v případě, kdy se objekt do obsahu vloží jako nepárová značka – tedy bez obsahu, jako je tomu na obrázku. Tento výchozí obsah definoval autor (programátor) dynamického objektu a jeho přesnou podobu lze najít v nápovědě systému k danému objektu. Pokud si však uživatel přeje definovat jiné rozložení prvků formuláře, může tak učinit standardní cestou jakou se například definuje obsah formuláře `mw:SearchForm`, kterému se věnovala předchozí kapitola.

3.6 SEO-optimalizace URL adres

V kapitole 3.1.2 byla popsána správa URL adres všech stránek webového portálu. Zabývala se ale pouze statickými URL adresami – tedy takovými, které jsou jasně a předem definované. V kapitole 2.1.4 však byl zmíněn i způsob, jakým by se měly nastavovat URL adresy s částmi, které jsou řízeny nějakou hodnotu URL parametrů. K tomuto účelu slouží dynamické objekty, jež se umisťují do textu, který určuje URL části adres, jak je zobrazeno na následujícím obrázku, kde je použit dynamický objekt `mw:Product`[8] pro překlad URL parametru `ProductID` určující `ID` produktu do názvu tohoto produktu.



Obr. 33 URL s definovaným dynamickým překladem parametru

Každý dynamický objekt pro překlad hodnot URL parametrů do textové části URL adresy vždy očekává, že při sestavování URL adresy na danou stránku, mu je předána konkrétní hodnota parametru. Tuto hodnou pak přeloží do požadovaného textu, který má být součástí URL adresy. Pokud mu není explicitně hodnota předána, pokusí se ji získat z aktuální URL adresy. Tímto je zajištěno, že například při definování odkazu na stránku objednávky zájezdu z jeho detailu, není nutné odkazu definovat, že se má předat parametr `ProductID`. Naopak, pokud se má odkaz sestavit na stránce, která nemá v URL parametr `ProductID`, je nutné mu explicitně tento parametr dodat jak například zobrazeno na obr. 34.

[8] Tento dynamický objekt není standardní součástí systému. Byl doprogramován jako součást ukázkového napojení na databázi systému Magic2G, která obsahuje tabulku se seznamem produktů se sloupečky, které obsahují URL ekvivalent pro název produktu nebo jeho kód.

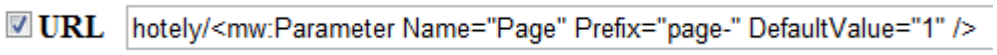
```
<mw:Static Text="Detail"
  NextPageID="101" Parameters="@ProductID=ID" />
```

Obr. 34 Vytvoření odkazu s definovanou hodnotou parametru ProductID

V opačném případě se vygeneruje URL adresa, která v místě pro překlad má prázdný text – tedy nekorektní adresa.

Standardní součástí systému jsou dynamické objekty `mw:Lang` a `mw:Parameter`. První z nich slouží pro překlad parametru `LangID`, který se používá pro podporu lokalizace obsahu, která je popsána v kapitole 3.6.1.3.

Druhý naopak sloučí pro překlad libovolné hodnoty parametru do URL adresy. Například URL adresu `http://www.nadrasky.cz/mc/hotely/?Page=5`, která obsahuje parametr `Page` pro určení čísla stránky, lze jednoduše převést například do podoby `http://www.nadrasky.cz/mc/hotely/page-5/`, která je mnohem čitelnější. Jak toho lze docílit je vidět na obr. 37, kde je použit právě objekt `mw:Parameter`.



Obr. 35 Příklad dynamického překladu URL parametru Page

3.6.1 Pokročilé vlastnosti

3.6.1.1 Podmínky

Všem dynamickým objektům, které jsou podporované redakčním systémem, lze přidat parametr `Condition`, jenž slouží k podmíněnému zobrazení objektu. Možné syntaxe podmínky jsou zobrazeny na následujících příkladech:

```
<mw:Static Text="☺" Condition="PageID==1" />
<mw:Static Text="☺" Condition="PageID>10" />
<mw:Static Text="☺" Condition="@MujParametr!=null" />
<mw:Static Text="☺" Condition="@MujParametr==@MujDruhyParametr" />
<mw:Static Text="☺" Condition="SectionID==10 & PageID<=15" />
<mw:Static Text="☺" Condition="SectionID!=10 | PageID>15" />
```

Obr. 36 Příklady podmínek

Přesná syntaxe by se dala pomocí jednoduché gramatiky popsat takto:

```
VYRAZ := PODMINKA [(&;|) VYRAZ]
PODMINKA := LEVY_OPERAND OPERATOR PRAVY_OPERAND
OPERATOR := (==;!=;<=;<;>=;>)
LEVY_OPERAND := (VLASTNOST;@NAZEV_PARAMETRU)
PRAVY_OPERAND := (null;HODNOTA;@NAZEV_PARAMETRU)
HODNOTA := libovolná textová hodnota
@NAZEV_PARAMETRU := název libovolného URL parametru
null := speciální označení hodnoty pro „nevyplněno“ – slouží pro porovnání, zdali hodnota vlastnosti/parametru existuje nebo je prázdná
```

Z definice syntaxe je patrné, že lze porovnávat libovolný URL parametr s jakoukoliv hodnotou či s hodnotou jiného URL parametru. Mimo to, je

možné porovnávat i hodnoty určitých vlastností, které je možné v daném kontextu získat. Kdekoliv je možné porovnávat vlastnost `PageID` a `SectionID`, které vždy obsahují `ID` aktuální stránky a `ID` aktuální sekce. Je tedy možné podmíněně zobrazovat určité objekty jen na některých stránkách či sekcích. V kontextu `mw:Template` v objektu `mw:DynamicList` lze zároveň i používat libovolnou vlastnost, kterou lze použít pro daný zdroj dat, ze kterého `mw:DynamicList` vypisuje hodnoty.
Jednotlivé podmínky lze skládat do podmíněných výrazů pomocí logických spojek & (`AND`) a | (`OR`), kdy při vyhodnocování má `AND` vyšší prioritu.

3.6.1.2 Způsob nastavování parametrů

Jak už bylo zmíněno dříve, parametry dynamických objektů se vždy dají nastavovat buď přímo do zdrojového kódu jako XML atribut u daného objektu, nebo v některých případech je umožněno přebírat hodnotu z URL parametru. To, jak přesně se parametry chovají, je možné najít v systému nápovědy, kde každý parametr má u sebe uvedeno, jakým způsobem systém postupuje při jeho nastavování. Existují celkem tři možnosti:

- Parametr lze nastavovat pouze z kódu – tedy jen pomocí XML atributu přímo u dynamického objektu.
- Jako hodnota parametru se primárně bere hodnota XML atributu z kódu a teprve poté se bere hodnota z URL parametru se stejným názvem.
- Opačný postup než předchozí varianta – tedy primárně z URL a teprve poté z XML atributu.

Toto výchozí chování lze jednoduše přepsat pomocí speciální definice hodnoty XML atributu tak, aby se vždy hodnota parametru objektu brala z konkrétního parametru v URL adrese. Tohoto může být například využito v případě, kdy dojde ke kolizi názvů filtračních parametrů a je tak nutné v jednom z objektů změnit chování tak, aby se hodnota kolizního parametru brala z jiného (příklad viz obr. 37).

```
<mw:DynamicList ... DestinationRootID="@ZemeID" ...
```

Obr. 37 Příklad přepsání výchozího chování parametru objektu

Nebo například v případě, kdy je nutné do parametru, který lze nastavovat pouze z kódu přiřadit hodnotu z URL, jak je ukázáno na obr. 38, kde je tímto způsobem docíleno vypsání hodnoty URL parametru s názvem `Typ`.

```
<mw:Static Text="@Typ" />
```

Obr. 38 Příklad výpisu hodnoty libovolného URL parametru

Další možností jak ovlivnit výchozí chování, je omezit přebírání hodnoty z URL. K tomuto účelu slouží speciální parametr každého dynamického

objektu s názvem `UrlIgnore`, do kterého lze vypsat všechny názvy URL parametrů (oddělené čárkou), které má daný objekt ignorovat. Nebo jednoduše lze zadat `UrlIgnore="All"` a tím tak budou ignorovány úplně všechny URL parametry. Logiku zadání lze taktéž otočit – tedy zadat seznam parametrů, které naopak jako jediné má dynamický objekt brát v potaz. Syntaxe tohoto způsobu vypadá například takto:
`UrlIgnore="ALL(DestinationID,ProductID)"` – slovně by se dalo přeložit „ignoruj vše kromě `DestinationID` a `ProductID`".

3.6.1.3 Lokalizační podpora

V systému existují prostředky, které jeho uživateli umožňují vytvořit webový portál, který podporuje vícejazyčné rozhraní. Předpokladem je, že v cílové databázi existuje tabulka, kde jsou uloženy všechny podporované jazyky. Tabulka musí obsahovat dva sloupce, které jsou dostupné v `mw:DynamicList` pod názvem `ID` a `Name`. Ve sloupci `Name` je nutné, aby byl uchováván název (zkratka) jazyka, který je rozpoznáván lokalizačním systémem frameworku .NET (tedy například pro češtinu to je `cs`[9] nebo specifičtěji `cs-CZ`[10]).
Aby systém poznal, který jazyk má být použit, je nutné, aby se v URL adrese vyskytoval parametr `LangID` s `ID` jazyka. Pro tento účel je nejlepší využít dynamického překladu parametrů do URL adresy, jenž byl popsán v předešlé kapitole. Dynamický objekt, který je určen právě pro překlad aktuálního jazyka je `mw:Lang`, kterému lze nastavit parametr `DefaultID`, který určuje výchozí jazyk. Tento překlad je nejlepší vložit přímo do URL adres webového projektu a tím je tak zajištěno, že se bude parametr `LangID` vyskytovat na každé stránce.
Lokalizační podpora je zabudovaná přímo do každého dynamického objektu vkládaného do obsahu a spočívá v možnosti nadefinovat, pro jaký jazyk je dynamický objekt určen, anebo možnost nadefinovat různé hodnoty parametrů pro různé jazyky. Obě tyto metody jsou ukázány na obr. 39:

```
<mw:Block Lang="en">
    <h1>Hotel list</h1>
</mw:Block>
<mw:Block Lang="cs">
    <h1>Seznam hotelů</h1>
</mw:Block>

<mw:Static Text="Výchozí text pro jakýkoliv jiný jazyk"
        cs:Text="Text pro českou jazykovou mutaci"
        en:Text="English text" />
```

Obr. 39 Ukázka lokalizačních metod v obsahu stránek

[9] Kód jazyka podle normy ISO 639-1 (6)
[10] Kód jazyka podle normy doplněný o kód regionu podle normy ISO 3166-1 (7)

První objekt s anglickým nadpisem se zpracuje pouze v jazykové mutaci stránek, které mají parametr `LangID` nastaven na jazyk s názvem `en` – tedy angličtinu. Druhý nadpis je naopak ručen pro češtinu. Třetí objekt využívá druhou metodu, kterou lze určit různé hodnoty parametrů pro různé jazyky. Každý XML atribut může být prefixován názvem jazyka. Při zpracování objektu pak systém ignoruje ty parametry, jejichž prefix se od aktuálního jazyka liší. Výchozí hodnota atributu, který nemá žádný prefix, se použije vždy v tom okamžiku, kdy žádný z prefixovaných atributů neodpovídá aktuálnímu jazyku určeného URL parametrem `LangID`.

3.7 Ladící nástroje

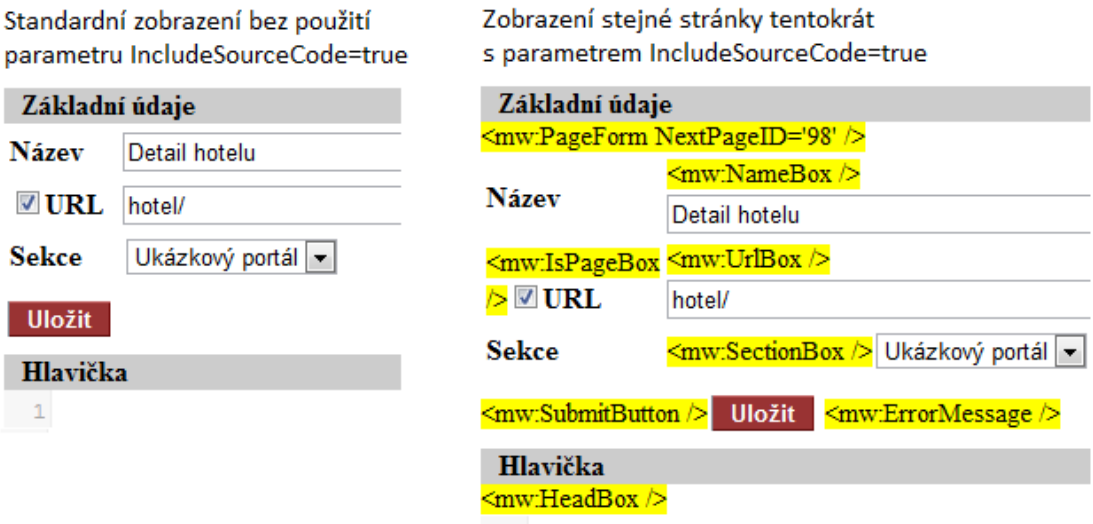
Součástí systému jsou různé prostředky, které pomáhají uživateli v případě, kdy se v systému vyskytuje nějaká chyba. Aby se systém a dynamické objekty chovaly korektně, je vždy nutné, aby byly všechny zadány syntakticky i sémanticky správně. V opačném případě systém nedokáže dynamický obsah interpretovat správně a musí o tom nějak informovat uživatele.
V systému existují tři typy chyb podle úrovně závažnosti:

- Uživatelské, plně zotavitelné
- Uživatelské, ze kterých se nadá zotavit
- Systémové, které většinou znamenají chybu v kódu dynamického objektu

Do první kategorie spadají veškeré syntaktické chyby typu neuzavřený tag či překlep v názvu objektu či atributu. Pokud systém na takovouto chybu narazí, je tento dynamický objekt včetně jeho dynamického obsahu do stránky vložen jako obyčejný text a v horní části stránky se vždy zobrazí žlutě podbarvená chyba informující o konkrétním typu chyby, jak je vidět na následujícím příkladě, kde je vidět dynamický objekt `mw:DynamicList` se zkomoleným názvem a způsob, jak si s tím systém poradil. Mezi tyto chyby patří například i některé sémantické – například použití `mw:Block` s parametrem `PageID`, jenž neobsahuje identifikátor žádné stránky. Prostě takové, které lze zkontrolovat už při inicializaci objektu.

Obr. 40 Příklad zotavitelné chyby

Do druhé kategorie spadají uživatelské chyby, které souvisejí s chováním objektů a jejich nastavení, ale které se těžko ověřují při prvotní inicializaci objektu. Pokud se takováto chyba v systému objeví, stránka bude obsahovat pouze žlutou chybovou hlášku.
Poslední kategorie je nejzávaznější a uživatel s ní zpravidla nic neudělá, protože se typicky jedná o chybu, kterou udělal programátor dynamického objektu. Tyto chyby končí HTTP stavem `500 – internal server error`.
Při vytváření webových projektů se uživatel často může dostat do situace, kdy potřebuje některé místo či objekt upravit, ale díky komplexní struktuře stránek a vnořených bloků například nemůže požadovaný objekt najít. Pro tyto účely systém obsahuje dva systémové URL parametry, které ovlivňují obsah stránky – jsou jimi `OnlySourceCode=true` a `IncludeSourceCode=true`. Oba slouží ke zjištění zdrojového kódu stránek (tedy toho, který je definován v administračním rozhraní a obsahuje dynamické objekty). První jmenovaný řekne systému, aby každý dynamický objekt vkládal do obsahu generované HTML stránky tak, jak ho najde ve zdrojovém kódu a interpretuje je jen do té míry, aby zjistil například jen jaký je jejich výchozí obsah – lze tak například jednoduše dohledat, v jakém bloku je přesně daný kus kódu zanořen. Druhý nijak neomezuje způsob interpretace dynamických objektů. Slouží k pouhému vložení zdrojového kódu každého dynamického objektu před HTML, které generuje, ve formě obyčejného textu, který je tak možné vyčíst přímo z vyrenderované stránky v prohlížeči. Příklad použití parametru `IncludeSourceCode=True` je vidět na následujícím obrázku:



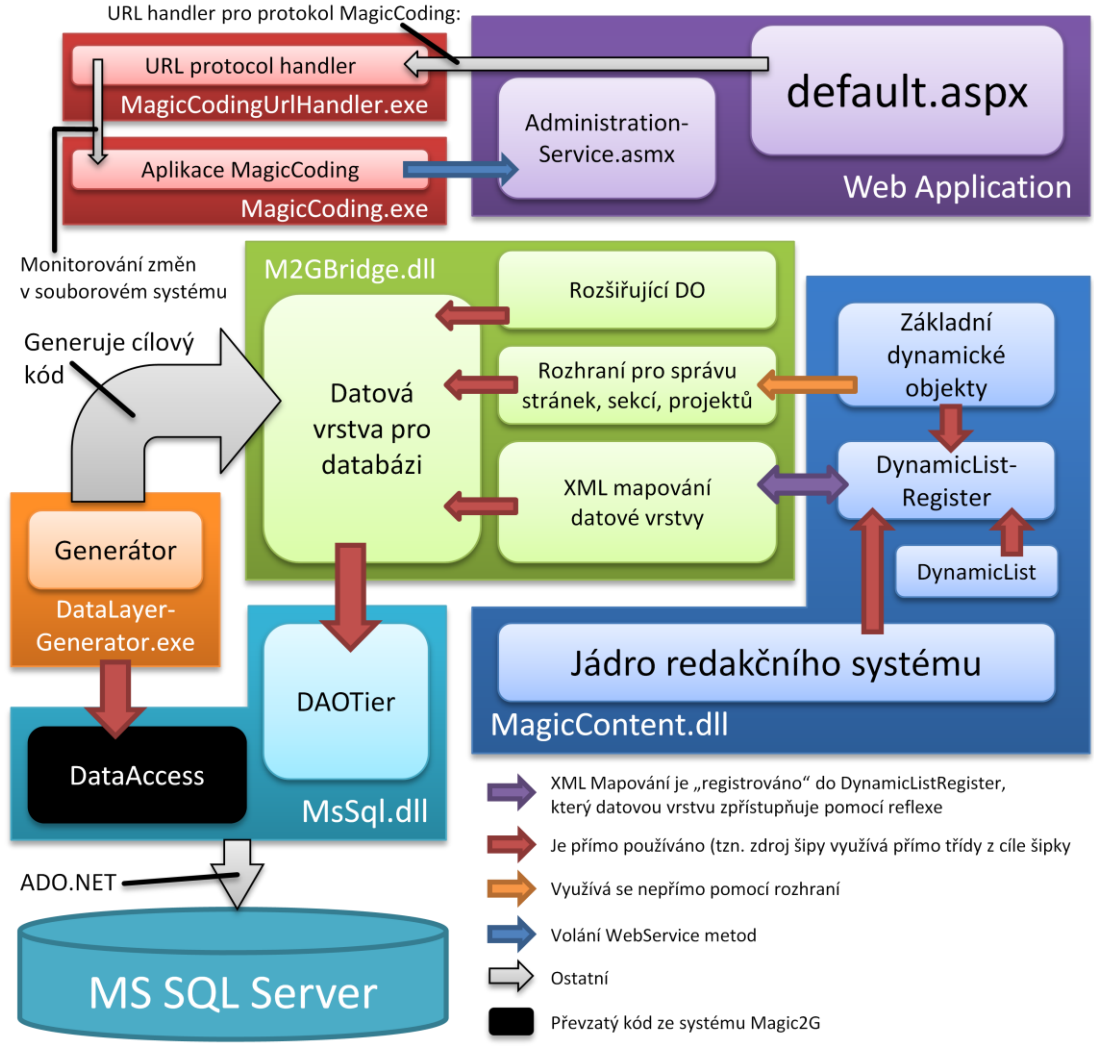
Obr. 41 Příklad použití parametru IncludeSourceCode=True

4 Programátorská dokumentace

Účelem celé této kapitoly je popsat hlavní architekturu celého systému MagicContent tak, aby čtenář byl schopen nejenom systém používat a rozšiřovat o nové dynamické objekty, ale také aby snadněji porozuměl celkovému konceptu, na kterém je postaveno jádro redakčního systému.
V první podkapitole bude popsáno, z jakých částí se systém skládá a jak jsou spolu propojeny. V podkapitolách, které budou následovat, pak budou jednotlivé části rozebrány podobněji. Inherentní částí práce je přiložená dokumentace vygenerovaná z XML komentářů v kódu, kde lze najít podrobnější popis jednotlivých popisovaných tříd a jejich metod.

4.1 Základní architektura

Systém MagicContent se skládá z celkem šesti hlavních komponent, jak je vidět na následujícím schématu celkové architektury:



Obr. 42 Celková architektura systému MagicContent

Základem celého systému je assembly MagicContent (tmavě modrá), která v sobě implementuje hlavní logiku celého redakčního systému. V jejím jádře jsou implementovány potomci základních tříd určených pro ASP.NET webové aplikace (konkrétně třídy `HttpApplication`, `HttpRequest` a `Page`), které slouží jako základní třídy pro webovou aplikaci (na obrázku fialová) reprezentující celý redakční systém. Součástí této assembly je také implementace všech základních objektů včetně `mw:DynamicList` a `mw:SearchForm`.

Aby systém vůbec mohl fungovat, je nutné k těmto dvěma základním komponentám doplnit část, která se stará o napojení na konkrétní databázi – jakýsi můstek mezi databází a redakčním systémem. Nejvhodnějším způsobem, jak vytvořit tuto část je pomocí další assembly, která se stará o implementaci tohoto napojení. Pro potřeby této práce byla tato důležitá část implementována jako ukázkové napojení na databázi systému Magic2G a assembly se tak jmenuje `M2GBridge` (na obrázku znázorněná zeleně).

Hlavní součástí můstku mezi databází a assembly MagicContent je nepochybně datová vrstva určená právě pro cílovou databázi. Tuto vrstvu lze kompletně vygenerovat pomocí jedné ze šesti částí celého projektu – generátorem datové vrstvy. Generátor je samostatná desktopová WPF aplikace celá implementována v projektu `DataLayerGenerator`.

Poslední velkou částí celé architektury je assembly `MsSql` (na obrázku světle modrou barvou). Skládá se ze dvou velkých částí – ze třídy `DataAccess` a ze tříd v namespace `DAOTier`. `DataAccess` zapouzdřuje přímý přístup k databázi pomocí ADO.NET tříd z namespace `System.Data.SqlClient`. V namespace `DAOTier` je naopak implementováno jádro datové vrstvy analyzované v kapitole 2.2.9.

Součástí projektu jsou ještě dvě desktopové aplikace, ve kterých je implementována klientská část funkcionality potřebná pro aplikaci `MagicCoding`.

4.2 Instalace a vystavení webové aplikace

Jako předpoklad úspěšného nasazení systému MagicContent spolu s ukázkovým napojením na databázi systému Magic2G se předpokládají následující prerekvizity:

- `Microsoft Windows 7 Professional 64 bit`[11]
- `Microsoft .NET Framework v4.0`
- `Microsoft SQL Server 2008 R2 v10.50`

[11] Webové servery jsou nejčastěji právě 64 bitové, a tak i konfigurační soubor web.config obsahuje cestu k 64 bitové variantě ISAPI modulu. Při použití na 32 bitovém operačním systému stačí změnit pouze tuto konfiguraci. Systém byl ale testován pouze v prostředí 64 bitového operačního systému, a tak byla tato skutečnost promítnuta i do minimální konfigurace.

- Microsoft SQL Server Management Studio
- Internet Information Services 7 (IIS7) pro ASP.NET s povoleným ISAPI modulem pro .NET v4.0[12]

Postup vystavení se skládá z následujících kroků:

1. Z instalačního CD z adresáře M2GSampleDatabase je nutné obnovit databázi například pomocí Management Studia na lokální SQL server do výchozí instance. Postup: spustit Management Studio; připojit se k výchozí instanci zadáním localost do pole Server name; LMB[13] na Databases; vybrat Restore Database…; vyplnit název databáze – např. M2GSample; zvolit zdrojový soubor M2GSample.bak; zaškrtnout u přidaného backup setu Restore; OK)
2. Zkopírovat obsah adresáře PrecompliedWeb do wwwroot adresáře s webovými aplikacemi pro IIS7 (typicky c:\inetpub\wwwroot\)
3. V případě, že databáze nebyla obnovena na výchozí instanci lokálního SQL Serveru, je nutné příslušným způsobem upravit connectionString na databázi v souboru …\wwroot\MagicContent\web.config
4. V administračním rozhraní IIS7[14] převést adresář MagicContent na webovou aplikaci.

4.3 Databázová vrstva

Základ pro datovou vrstvu představuje assembly MsSql, která se skládá ze dvou velkých částí: z namespace MsSql, které bylo převzato ze systému Magic2G a z namespace DAOTier, které v sobě implementuje veškerou logiku spojenou s návrhem datové vrstvy z kapitoly 2.2.9.

4.3.1 Namespace MsSql a třída DataAccess

Všechny třídy z namespace MsSql byly převzaty ze systému Magic2G, a tak se nejedná o dílo této práce. Nejdůležitější třídou je nepochybně třída DataAccess, která v sobě zapouzdřuje použití .NET třídy System.Data.SqlClient.SqlConnection. Implementuje množství metod sloužící ke spouštění dotazů nad databází. Tyto metody na vstupu očekávají buď SQL dotaz v textové podobě nebo přímo instanci třídy SqlCommand. O zpracování výsledku dotazu se již musí postarat volající kód pomocí zpřístupněného DataReaderu. Tyto metody jsou využívány hlavně v DataService třídách z namespace DAOTier a také v aplikaci DataLayerGenerator, která DataAccess využívá pro čtení struktury databáze.

[12] Podrobněji popsanou instalaci IIS7 spolu s konfigurací lze najít na přiloženém CD v souboru IIS7_instalace_a_konfigurace.pdf

[13] Left Mouse Button (levé tlačítko myši)

[14] Lze spustit z příkazové řádky pomocí „%windir%\system32\inetsrv\InetMgr.exe“

V hlavní části tohoto projektu – tedy v assembly `MagicContent` – je třída `DataAccess` pouze vytvářena a předávána do nižších vrstev. Metody pro přístup k databázi se tak v assembly žádným způsobem nevyužívají.
Instanci třídy `DataAccess` lze vytvořit buď přímým předáním `connectionStringu` s popisem přístupu k databázi (takto je vytvářena instance v projektu `DataLayerGenerator`) nebo pomocí statické vlastnosti `Default`. Druhá možnost je vhodná na použití ve webových aplikacích, protože vrací instanci unikátní pro aktuální webový request, jejíž `connectionString` se přebírá z nastavení z konfiguračního souboru `web.config`.

4.3.2 Namespace DAOTier a DataService třídy

Vrstva `DAOTier`[15] v sobě zahrnuje implementaci datové vrstvy analyzované v kapitole 2.2.9. Cílem následujícího textu je navázat na principy popsané v analýze a popsat přesný postup, jakým se přesně vytvářejí dotazy jen na základě znalosti vlastností. Dále je zde například zmíněn prostředek, který řeší problém s možnými konflikty v názvech (aliasech) jednotlivých vlastností (sloupců).
Pro pochopení fungování celého procesu tvorby dotazu, je nejprve nutné se seznámit se strukturou dat, které definují části dotazu – znovu tedy s třídami, které již byly přiblíženy v kapitole 2.2.9, ale tentokrát podobněji. Každý popis základní třídy z namespace `DAOTier` (na obrázku celkové architektury v modré části) bude následovat i ukázkou jejího použití při implementaci napojení na konkrétní databázi (na obrázku celkové architektury v zelené části). Tento ukázkový kód z napojení bude vždy příslušně označen, aby bylo jasné, že je celý kompletně vygenerovaný aplikací, která je součástí práce v assembly `DataLayerGenerator` a která je popsána v kapitole 4.3.6. Následující ukázky kódu byly co nejvíce zjednodušeny kvůli přehlednosti – byly tak odstraněny například klíčová slova `public` či jiná syntaktická specifika, která nejsou z pohledu architektury důležitá.

[15] DAO = Data Access Objects (objekty pro zpřístupnění dat)

```
class PropertyType
{
  string Name;
  string Expression;
}
class Property
{
  PropertyType PropertyType;
  string AliasPrefix;
  string Alias { get { return AliasPrefix + "_"  + PropertyType.Name; } }

  Func<DataService, TableDescriptor> CreateDescriptor;
}
```

Obr. 43 Definice tříd PropertyType a Property

Dříve bylo řečeno, že vlastnost (třída `Property`), obsahuje dvě základní vlastnosti: název a SQL výraz. Implementačně byly tyto dvě vlastnosti vyčleněny do samostatné třídy `PropertyType`. `Property` tak jen obsahuje referenci na objekt tohoto typu. Mimo to obsahuje ještě dvě důležité vlastnosti: `AliasPrefix` a delegát `CreateDescriptor`.

`AliasPrefix` slouží k jednoznačné identifikovatelnosti vlastnosti v nějakém dotazu – například v dotazu z obr. 10 na stránce 19 je nutné přidat ke sloupci `k.Nazev` nějaký alias, který musí být v rámci dotazu unikátní. Pro tento případ slouží právě `AliasPrefix`, který v případě vlastnosti na tomto obrázku bude obsahovat hodnotu „`Produkt_Katalog`“, takže `Alias` celé vlastnosti bude „`Produkt_Katalog_Nazev`“.

`CreateDescriptor` je delegát obsahující metodu, která slouží pro vytvoření příslušné instance třídy `TableDescriptor`, která popisuje, jak se má do dotazu připojit tabulka, ve které se daná vlastnost nachází (podrobnější popis bude poskytnut dále v textu, který se věnuje právě třídám `TableDescriptor`).

Protože SQL výraz a základní název sloupce v konkrétní tabulce jsou vždy stejné, ať je tabulka „přijoinovaná“ jakýmkoliv způsobem, je tohoto využito, a tak každá entita reprezentována nějakou tabulkou definuje statický seznam těchto podporovaných vlastností, jak je vidět na následujícím kódu:

```
class Katalog
{
  class PropertyTypes
  {
    public static PropertyType ID =
      new PropertyType() { Name = "ID", Expression = "k.ID" };

    public static PropertyType Nazev =
      new PropertyType() { Name = "Nazev", Expression = "k.Nazev" };
  }
}
```

Obr. 44 Ukázka konkrétní definice PropertyType objektů pro entitu Katalog (generovaný kód)

Další třídou, která je specifická pro každou entitu, je třída `TableProperties`. Ta má za úkol definovat, jaké vlastnosti lze z dané tabulky získat. Příklad pro entitu `Katalog` je následující:

```
class Katalog
{
  class TableProperties
  {
    Property ID = new Property(PropertyTypes.ID);
    Property Nazev = new Property(PropertyTypes.Nazev);

    public TableProperties(string aliasPrefix,
      Func<DataService, TableDescriptor> createDescriptor)
    {
      ID.AliasPrefix = aliasPrefix;
      Nazev.AliasPrefix = aliasPrefix;

      ID.CreateDescriptor =
         (dataService) => dataService.AddDescriptor(Nazev);

      Nazev.CreateDescriptor = createDescriptor;
    }
  }
}
```

Obr. 45 Definice třídy TableProeprties pro entitu Katalog (generovaný kód)

Z ukázky kódu je vidět, že tabulka pro entitu `Katalog` obsahuje dvě vlastnosti (`ID` a `Nazev`), které jsou založeny na typech sloupečků z předešlého obrázku (obr. 44). Konstruktor jako vstupní parametry očekává řetězec `aliasPrefix` určený všem vlastnostem a metodu `createDescriptor`. Tato metoda musí obsahovat kód, který vytváří potomka třídy `TableDescriptor`, který popisuje, jak se bude daná tabulka připojovat do konkrétního dotazu. V kapitole 2.2.9 byla místo metody uvažována přímo instance, nicméně z implementačního hlediska to musí být metoda. Instance potomka třídy `TableDescriptor` totiž má na starosti nejen popis JOINu, ale také shromažďuje všechny vlastnosti, které se mají z dané tabulky získat v jednom konkrétním dotazu. Zároveň se stará i o získání jejich hodnot z `SqlReader`u. Její instance se tak vytváří při každém dotazu. A právě proto, že se tato instance stará o všechny vlastnosti pro danou tabulku, je nutné, aby bylo zajištěno, aby se při přidávání dalších vlastností do dotazu používala již dříve vytvořená instance pro jinou vlastnost. Tohoto je dosaženo vhodným nastavením metod `CreateDescriptor` u každé z vlastností, kdy se jen poslední vlastnosti nastaví tato metoda na předaný parametr `createDescriptor`. Všem ostatním se nastaví na funkci, která jen vrací stejný deskriptor jako poslední vlastnost (na obr. 45 naznačeno zeleným písmem).

Nyní je vhodná doba na osvětlení provázání tříd `DataService` a `TableDescriptor`.

```
abstract class TableDescriptor
{
  List<Property> Properties;
  abstract string GetJoinFormula();
}

class DataService
{
  TableDescriptor AddDescriptor(Property prop)
  {
     … //pokus o nalezení již dříve vytvořené instance
    return prop.CreateDescriptor();
  }
  TableDescriptor AddDescriptorWithProperty(Property prop)
  {
     TableDescriptor descriptor = AddDescriptor(prop);
     descriptor.Properties.Add(property);
     return descriptor;
  }

  void CreateDescriptors()
  {
    foreach (Property property in DesiredProperties)
      AddDescriptorWithProperty(property);
  }
}
```

Obr. 46 Důležité metody tříd DataService a TableDescriptor

Již dříve bylo napsáno, že třída `TableDesciptor` má na starosti dvě důležité věci: popsat, jak se konkrétní tabulka připojí do dotazu pomocí SQL JOINu a zároveň i shromažďovat všechny vlastnosti, které se mají získat z dotazu. První funkcionalita je zajištěna metodou `GetJoinFormula`, kterou musí přepsat její potomek a slouží k vrácení SQL výrazu, který obsahuje příslušný JOIN. Pro shromažďování vlastností se naopak využívá kolekce `Properties`. Třída `DataService` při sestavování dotazu volá svoji metodu `CreateDescriptors`, která slouží k vytvoření všech deskriptorů pro všechny tabulky, jež jsou potřeba pro všechny vlastnosti, které se mají v dotazu vrátit. Pro vytvoření deskriptoru vlastnosti implementuje dvě metody: `AddDescriptor` a `AddDescriptorWithProperty`. Jejich chování se liší jen nepatrně. `AddDescriptor` má za úkol najít již dříve vytvořenou instanci deskriptoru pro danou vlastnost (kouká se do interního slovníku) a pokud takovou nenajde, zavolá metodu `CreateDescriptor` přímo na vlastnosti (a tuto si poté uloží do interního slovníku). `AddDescriptorWithProperty` dělá úplně do samé akorát s tím rozdílem, že po přidání deskriptoru se daná vlastnost nastaví do kolekce `Properties` daného deskriptoru. Při volání `AddDescriptorWithProperty` je tak zajištěno, že daná vlastnost bude součástí výsledku dotazu (zjednodušeně „bude v SELECTu“). Zatímco

voláním `AddDescriptor` bude jen zajištěno, že se do dotazu připojí jen tabulka pro danou vlastnost.
Všechny výše popsané metody a vlastnosti jsou využity při definici vlastnosti reprezentující celou tabulku, jak je vidět na následující ukázce:

```
class Produkt
{
  class TableProperties
  {
    Property KatalogID = new Property(PropertyTypes.KatalogID);

    Katalog.TableProperties Katalog =
        new Katalog.TableProperties("Katalog", (dataService) =>
        {
            dataService.AddDescriptorWithProperty(KatalogID));
            return new Katalog.KatalogTableDescriptor(KatalogID);
        });
  }
}
class Katalog
{
  class KatalogTableDesciptor : TableDescriptor
  {
    public KatalogTableDescriptor(Property requiredProperty)
    { … }

    public override string GetJoinFormula()
    {
      return "LEFT JOIN Katalog as /k/ " +
             "ON k.ID = {RequiredProperty}";
    }
  }
}
```


Obr. 47 Ukázka implementace JOINované vlastnosti (generovaný kód)

Na obrázku je ve třídě `Produkt.TableProperties` definována vlastnost `KatalogID`, která je využita při tvorbě JOINu Katalog, který je instancí třídy `Katalog.TableProperties`. Konstruktoru této třídy je předán `aliasPrefix` „`Katalog`“ a jako parametr `createDescriptor` je předána lambda-metoda, vytvářející deskriptor popisující připojení katalogu. V těle metody se napřed do dotazu přidá vlastnost `KatalogID`, která je potřebná pro JOIN tabulky `Katalog`, a tato se pak předává jako vlastnost, podle které se má provést JOIN, do konstruktoru třídy `Katalog.KatalogTableDescriptor`, která popisuje nejtypičtější připojení katalogu – tedy podle jeho primárního klíče `ID`.
Řetězec, který vrací metoda `GetJoinFormula`, musí vždy obsahovat SQL výraz pro připojení tabulky k dotazu. Ačkoliv se jedná o SQL syntaxi, jsou zde dvě drobné odlišnosti. První z nich je víceméně kosmetická – je nutné uzavřít všechny použité aliasy joinovaných tabulek mezi dvě lomítka. To proto, aby bylo možné při sestavování dotazu tento alias nahradit a nedocházelo tak ke konfliktům v názvech – více o tomto problému se rozepisuje kapitola 4.3.3. Druhá už upravuje sémantiku výrazu. JOIN je totiž

vždy závislý na nějaké vlastnosti (nebo vlastnostech), podle kterých se má provést. Místo takového výrazu se do textu JOINu vkládá značka `{RequiredProperty}`, která je pak při samotném sestavování dotazu nahrazena příslušnou vlastností, která je předána v konstruktoru deskriptoru. Takto je možné využívat jednu definici JOINu kdekoliv v systému a je tak možné připojit například tabulku katalog kdekoliv, kde je nějaký cizí klíč odkazující na primární klíč tabulky `Katalog`. Při vytváření datové vrstvy pro konkrétní databázi se tak zbytečně neduplikuje již jednou napsané SQL. Toto je velká výhoda v případech, kdy JOIN není zcela typickým a obsahuje například i kontrolu platnosti záznamu či je veden přes více tabulek – programátor tak při použití již na toto nemusí myslet a je tak zároveň i zmenšen prostor pro vytvoření možné chyby.

Vedle způsobu, jakým jsou vytvářeny třídy `TableProperties` existuje ještě jeden, který je použitý pro definici statické vlastnosti `Properties`. Tato vlastnost vyžaduje, aby deskriptor, který je použit pro připojení tabulky k dotazu, ve skutečnosti nedělal vůbec nic, protože daná tabulka již v dotazu existuje za `FROM`. K tomuto účelu slouží nejjednodušší forma `TableDescriptoru`, která se jmenuje `SimpleTableDescriptor`. Způsob, jakým je vyvářena vlastnost například pro tabulku produkt je vidět na obr. 48, kde je například i vidět, že jako `aliasPrefix` se objektu třídy `TableProperties` předává prázdný řetězec.

```
class Produkt
{
  static Produkt.TableProperties Properties =
    new Produkt.TableProperties("", (dataService) =>
    {
        return new SimpleTableDescriptor();
    });
}
```

Obr. 48 Vytvoření hodnoty statické vlastnosti Properties (generovaný kód)

4.3.3 Metody pro čtení dat a tvorba dotazu

Jak už bylo zmíněno, sestavování kompletního dotazu řídí třída `DataService`. Ve skutečnosti je veškerá logika umístěna do abstraktní třídy `BaseDataService`. `DataService` je pouze jejím silně typovým rozšířením[16], které díky typovosti umožňuje naplno využívat našeptávání v prostředí Visual Studia. Jedná se ale opět o abstraktní třídu, a tak je vždy nutné tvořit potomka pro každou tabulku, pro kterou je `DataService` třída určena.

[16] Třídě BaseDataService lze předávat například libovolnou vlastnost Property. Silně typový potomek toto omezuje a dovoluje přidávat pouze ty vlastnosti, které jsou určeny pro daný typ servisu pro danou tabulku. Jedná se o běžně používaný koncept například v třídách aplikačního rozhraní Windows Communication Foundation.

Potomci jsou však velmi jednoduché třídy, které obsahují vždy jen logiku specifickou pro danou tabulku. Tyto třídy lze opět generovat pomocí aplikace `DataLayerGenerator`. Příklad třídy `DataService` pro tabulku `Produkt` je na následující ukázce kódu:

```
public class ProduktFilter : DataService<
    ProduktItem,
    Produkt.Property,
    ProduktFilter.FilterParametersDefinition,
    ProduktFilter>
{
  public ProduktFilter()
    : base("Produkt", "p", Produkt.Properties.ID)
  { }

  public class FilterParametersDefinition : Filter
  {
    …
  }

  protected override void AppendWhere(
     Joiner whereJoiner, SqlParameterCollection sqlParams)
  {
    …
  }
}
```

Obr. 49 Definice potomka třídy DataService pro tabulku Produkt (generovaný kód)

Na povšimnutí stojí hlavička definice třídy – silně typový předek `DataService` totiž vyžaduje čtyři typové parametry:

- První parametr (`ProduktItem`) určuje typ datové položky, kam se budou ukládat hodnoty vrácené databází – může se jednat o libovolného potomka třídy `DataItem`[17], tak klidně přímo o třídu `DataItem`.
- Druhý určuje typ vlastností (`Produkt.Property`), které se mohou do dotazu přidávat. Podobně jako v případě prvního parametru to může být libovolný potomek třídy `Property` anebo sama třída `Property`.
- Třetí určuje typ třídy, kde jsou definovány všechny filtrační parametry pro danou entitu (více o filtračních parametrech viz následující kapitola).
- Posledním parametrem je typ samotného potomka. Tento je předáván čistě kvůli implementaci statických metod (v předkovi) pro zjednodušené volání, kdy není nutné přímo vytvářet instanci potomka (více viz níže).

Od potomka je také explicitně vyžadováno nadefinování konstruktoru, protože předek `BaseDataService` vyžaduje celkem tři parametry:

- Název tabulky, pro kterou je potomek určen („`Produkt`“).

[17] Třída `DataItem` byla popsána už v kapitole 2.2.9. Jedná se o třídu zapouzdřující slovník, kde klíčem je objekt třídy `Property` a hodnota je libovolný `object`. Slouží tak pro uchování hodnot získaných z databáze pro jeden záznam.

- Alias tabulky `Produkt` („p“), který je používaný v SQL výrazech vlastností ve třídě `Produkt.TableProperties` nebo také v SQL výrazech určených za `WHERE` klauzuli dotazu.
- A nakonec vlastnost reprezentující primární klíč tabulky (`Produkt.Properties.ID`), který je využíván v metodách pro zjišťování `ID` záznamu. Pokud tabulka nemá primární klíč nebo je primární klíč určen více sloupci, je možné zvolit libovolnou vlastnost (v tom případě budou metody, jako je například `GetID`, nepoužitelné).

Poslední částí kódu, který je nutné v potomkovi vytvořit je přetížení abstraktní metody `AppendWhere`, která slouží pro sestavení `WHERE` podmínky dotazu. O způsobu sestavování `WHERE` podmínek se lze více dočíst v následující kapitole.

Nyní je možné popsat způsoby, jakým se dají servisní třídy používat. V předcích jsou implementované nejpoužívanější metody pro přístup k datům. Nejjednodušší jsou například `Exists` a `Count`. Jejich účelem je zjištění, zdali v tabulce existují nějaké záznamy vyhovující nadefinovaným podmínkám (resp. kolik jich je). Příklad použití `Exists` je vidět na následující ukázce:

```
ProduktFilter dataService = new ProduktFilter();
dataService.FilterParameters.KatalogID = 123;
if (dataService.Exists(DataAccess.Default))
  Console.WriteLine("V databázi existuje produkt z katalogu s ID 123!");
```

Obr. 50 Příklad použití metody Exists

Každá z metod vždy jako parametr vyžaduje instanci třídy `DataAccess`, která je použitá pro přístup k databázi. Většina metod také existuje ve statické variantě, která ještě více zjednodušuje kód (viz obr. 51, kde se filtrační parametry nastavují lamda-metodou, které je předán objekt (pojmenovaný `f`) z vlastnosti `FilterParameters` z předchozího obrázku). Kompletní seznam metod včetně všech jejich přetížení lze najít v přiložené dokumentaci.

```
if (ProduktFilter.Exists(DataAccess.Default, f => f.KatalogID = 123))
  Console.WriteLine("V databázi existuje produkt z katalogu s ID 123!");
```

Obr. 51 Příklad použití statické varianty metody Exists

Nejpoužívanější metody jsou však různá přetížení metody `GetList`. Příklad jejího použití lze najít v kapitole 2.2.9 na obr. 13 na straně 22. Postup je vždy takový, že se určí, které vlastnosti mají být z dané tabulky získány. Dále se pak nastaví případné filtrační parametry a následně se zavolá metoda pro získání jednotlivých záznamů v podobě seznam objektů třídy `DataItem` (či jejího potomka).

Celý postup jakým je poskládán dotaz a jak se dosáhne výsledku v podobě seznamu objektů `DataItem` je velmi zjednodušeně znázorněn na následující ukázce kódu na obr. 52 (pro demonstrační účely je kód velmi zjednodušen a

nejedná se o přesnou kopii zdrojového kódu). Kvůli přehlednosti je popis kódu obsažen přímo v ukázce ve formě komentářů.

```
public List<DataItem> GetList(DataAccess dataAccess)
{
  // Zde se vyvoří všechny deskriptory pro každou vlastnost
  // a každému z nich se nastaví jeho pořadí v rámci dotazu.
  foreach (Property prop in DesiredProperties)
    AddDescriptorWithProperty(prop);

  SqlCommand command = new SqlCommand();
  StringBuilder selectBuilder, joinsBuilder, whereBuilder, orderByBuilder;

  // Každý deskriptor pak vytvoří všechny JOINy a naplní část dotazu za SELECT.
  // Deskriptor při tomto procesu z JOINů zároveň zjišťuje všechny použité
  // aliasy tabulek, které pak ve všech částech SQL nahradí tak, že za ně
  // přípojí znak _ a číslo se svým pořadím. Je tak zajištěno unikátní oznacční
  // všech tabulek v dotazu.
  foreach (TableDescriptor descriptor in AllDescriptors)
    descriptor.PrepareQuery(selectBuilder, joinsBuilder, orderByBuilder);

  // Poté se vytvoří WHERE podmínka.
  AppendWhere(whereBuilder, command.Parameters);

  // Nakonec se ze všech částí se poskládá celý dotaz, který za FROM uvede
  // hodnotu z konstruktoru (tedy název tabulky a její alias).
  command.CommandText =
    "SELECT " + selectBuilder +
    "FROM " + BaseTableName + " " + BaseTableAlias +
    joinsBuilder +
    "WHERE " + whereBuilder +
    "ORDER BY " + orderByBuilder;

  // Dotaz se následně odešle do databáze ke zpracování.
  dataAccess.DoCommandOperation(command);
  SqlDataReader reader = command.ExecuteReader();

  // Postupně se čtou databází vrácené záznamy.
  List<DataItem> resultList;
  while (reader.Read())
  {
    // Pro každý záznam se vytvoří položka ve výsledeném seznamu,
    DataItem item = new DataItem();
    // do které nastaví všechny dekriptory hodnoty z readeru.
    foreach (TableDescriptor descriptor in AllDescriptors)
      descriptor.SetValue(reader, item);
    resultList.Add(item);
  }
  // A je hotovo :)
  return resultList;
}
```

Obr. 52 Algoritmus sestavení dotazu a získání jeho výsledku z databáze

Poslední metodou je metoda `GetTree`, která je využívána dynamickým objektem `mw:DynamicList`. Jediným místem, kde se tato metoda od metody `GetList` liší, je způsob, jakým jsou čtena data za `readeru` ve `while` cyklu. Zde se totiž napřed přečtou hodnoty vlastností, které tvoří stromovitou strukturu a teprve po vytvoření příslušných uzlů stromu se projdou všechny deskriptory, které nastaví zbylé vlastnosti. Pokud se má navíc stránkovat, je

kód, který prochází všechny deskriptory a získává všechny hodnoty vlastností z `readeru`, volán jen pro ty uzly, které se vyskytují na požadované stránce. Zpracování dotazu je tak výrazně urychleno. Navíc pokud není vyžadováno znát celkový počet stránek, je možné po přečtení záznamu, který se už nachází za požadovanou stránkou, přerušit zpracování celého dotazu. A protože nejčastěji zobrazovanou stránkou je ta první, jedná se opět o velmi zajímavou optimalizaci.

4.3.4 LazyJoin operace

Datová vrstva má ještě jednu zajímavou funkci, která má za cíl urychlit stránkované výpisy. Jedná se o tzv. `LazyJoin` operace, které ale nejsou v redakčním systému nijak využity[18]. Jejich princip spočívá v tom, že v dotazu, který se posílá ke zpracování, jsou jen vlastnosti z hlavní tabulky a je využito některé metody používané pro stránkování v prostředí T-SQL[19] (tedy buď pomocí klauzule `TOP` nebo pomocí využití funkce `ROW_NUMBER`). Tento dotaz je tak velice rychle zpracován, jelikož SQL server nemusí provádět nákladné JOIN operace na všech datech. Teprve po získání vlastností všech záznamů z hlavní tabulky se prochází postupně všechny deskriptory a místo metody pro získání hodnot z `readeru` se volá metoda, jenž má za úkol tyto hodnoty zjistit pomocí samostatného dotazu. Ačkoliv se při tomto přístupu posílá do databáze velké množství samostatných dotazů, je zpracování často mnohem rychlejší, než standardní přístup jednoho dotazu, ve kterém je vše připojeno pomocí mnoho JOIN operací. Zrychlení je nejpatrnější v okamžiku, kdy je v tabulce velké množství záznamů (desítky tisíc a více) a stránkování je omezeno na „rozumné“ množství záznamů (např. do sta záznamů na stránku). Zpracování všech JOIN operací v jednom velkém dotazu je totiž nákladnější, než zpracování desítek či stovek podstatně jednodušších dotazů. Tyto dotazy jsou totiž velice rychlé, jelikož často využívají databázových indexů či přímo indexů nad primárními klíči.

4.3.5 Filtrační parametry

Princip pro sestavování `WHERE` podmínek byl navržen tak, aby jejich použití bylo co možná nejjednodušší, ale zároveň dovolilo definovat libovolnou formu podmínky. Pro tento účel byly vytvořeny jednoduché třídy, jejichž instance slouží jako tzv. filtrační parametry. V systému jich existuje několik typů rozlišených podle typu sestavované podmínky a databázového typu

[18] Stránkování v redakčním systému využívá pouze dynamický objekt mw:DynamicList. Ten ale vyžaduje speciální typ stránkování, který není založen na celkovém počtu všech záznamů vrácených z databáze a nelze tak využívat LazyJoin operace.

[19] Označení varianty jazyka SQL používaného v prostředí MS SQL Serveru, který obsahuje různá rozšíření oproti SQL standardu.

sloupečku či hodnoty, pro kterou jsou určeny. Hlavní rozdělení podle typu sestavované podmínky vypadá následovně:

- Vyhledávání podle jedné nebo více definovaných hodnot pomocí operátoru porovnání (= a <>) či množinového `IN` a `NOT IN`
  - Sem spadají třídy `IntSearch`, `DateSearch`, … (další viz dokumentace)
- Vyhledávání hodnoty, která spadá do zadaného intervalu
  - Sem spadají třídy `IntRangeSearch`, `DateRangeSearch`, …
- Vyhledávání textové hodnoty
  - Sem spadají třídy `TextSearch` a `FullTextSearch`

Každá z těchto tříd umožňuje nadefinovat libovolný tvar požadované podmínky. Například `TextSearch` umožňuje určit, zdali se má vyhledávat kdekoliv v textu nebo jen na začátku nebo zdali se má rozlišovat diakritika či velikost písmen a nebo se například místo `LIKE` operátoru má použít fulltextová funkce `CONTAINS`. `IntRangeSearch` a jí podobné lze naopak určit, zdali se má vyhledávat v otevřeném či uzavřeném intervalu. Nebo dokonce v případě hledání podle intervalu, který se má porovnávat v databázi se dvěma sloupci, které také určují interval, lze například určit, zdali se má být interval v databázi obsažen celý ve vyhledávaném či zdali stačí, jen aby jakkoliv zasahoval do vyhledávaného. Všechny možnosti lze najít v dokumentaci k příslušné třídě.

Implementace konkrétních parametrů spočívá v jeho nadefinování ve třídě `FilterParameters` (viz definice potomka třídy `DataService` v kapitole 4.3.2) a pomocí některé z metod, které nabízí předek `BaseDataService`, ho zakomponovat do procesu sestavování `WHERE` podmínek v překryté metodě `AppendWhere`. Příklad definice konkrétních filtrů lze najít na obr. 53, kde jsou ve vnořené třídě `FilterParametersDefinition` nadefinovány filtrační parametry `ID`, `SekeceID` a `WebovyProjektID` a v překryté metodě `AppendWhere` je pak implementováno jejich připojení k dotazu. Parametry `ID` a `SekceID` lze jednoduše generovat pomocí aplikace `DataLayerGenerator`. Složitější filtrační parametr `WebovyProjektID` je však už nutné implementovat ručně v `partial` souboru pro danou třídu.

```
partial class StrankaBlokFilter
{
  partial class FilterParametersDefinition
  {
    public IntSearch ID;
    public IntSearch SekceID;
    public IntSearch WebovyProjektID;
  }
  protected override void AppendWhere(Joiner whereJoiner, SqlParameterCollection sqlParams)
  {
    SetMultiValueCondition(
      whereJoiner, sqlParams, FilterParameters.ID, "sb.[ID]", "@ID");

    SetMultiValueCondition(
      whereJoiner, sqlParams, FilterParameters.SekceID, "sb.[SekceID]", "@SekceID");

    SetExistsMultiValueCondition(whereJoiner, sqlParams, FilterParameters.WebovyProjektID,
          "EXISTS(SELECT * FROM Sekce sek WHERE sb.SekceID = sek.ID AND {0})",
          "sek.WebovyProjektID", "@WebovyProjektID");
  }
}
```

Obr. 53 Příklad definice filtračních parametrů pro tabulku StrankaBlok

Všechny filtrační parametry jsou standardně spojovány pomocí logického operátoru `AND`. Třída `DataService` navíc ale umožňuje vytvořit i komplexnější filtrační parametry skládané pomocí jiných logických operátorů tak, jak je zobrazeno na obr. 54.

```
var filter1 = new StrankaBlokFilter.FilterParametersDefinition();
filter1.ID = 456;

var filter2 = new StrankaBlokFilter.FilterParametersDefinition();
filter2.SekceID = 81;

var filter3 = new StrankaBlokFilter.FilterParametersDefinition();
filter3.WebovyProjektID = 23;

StrankaBlokFilter dataService = new StrankaBlokFilter();
dataService.ComplexFilter = filter1 | (filter2 & !filter3);
```

Obr. 54 Ukázka skládaných filtrů pomocí logických operátorů

Jednoduché použití pomocí přetížení logických operátorů umožňuje implementace třídy `FilterComposition`, která má za úkol shromažďovat jednotlivé filtrační třídy a spojovat je do dotazu příslušnými operátory. Má přetíženy binární operátory `&` a `|` a unární `!` pro všechny kombinace dvojic objektů typu `FilterCompodition` a třídy `Filter`, od které všechny třídy s filtračními parametry dědí.

4.3.6 Generátor DataService tříd

Aplikace pro generování datové vrstvy `DataLayerGenerator` je součástí projektu ve stejně pojmenované assembly a je tak možné ji spouště buď přímo z prostředí Visual Studia nebo jako samostatnou desktopovou aplikaci, jejíž sestavenou verzi lze najít na přiloženém CD v adresáři `DataLayerGenerator`.

Uživatelské rozhraní je velice jednoduché a intuitivní. Při prvním použití stačí zadat `connectionString` na databázi a stisknout tlačítko „Načíst strukturu“. Aplikace pak na základě systémového katalogu příslušné databáze zjistí všechny tabulky včetně jejich sloupců a na základě definovaných cizích klíčů i jejich provázání.
Detail vlastností každé tabulky lze zobrazit jejím vybráním z levého seznamu. Na základních údajích lze najít základní název třídy, který bude součástí všech vygenerovaných tříd, alias používaný u tabulky a také kód pro vygenerovaný `TableDescriptor` (jak SQL pro JOIN, tak `LazyJoin`). Pro vygenerování `DataService` třídy pro danou tabulku je nutné vybrat zaškrtávátko. Na další záložce Sloupce je uveden seznam všech sloupců tabulky. U každého z nich lze navolit, zdali má být pro něj vygenerována vlastnost (`Property`) a také i možnost nadefinovat filtrační parametr a jeho typ. Třetí záložka umožňuje nadefinovat libovolné JOINy z dané tabulky spolu i s výběrem vlastnosti, která je k provedení JOIN operace vyžadována. Poslední záložka se věnuje možnostem vygenerování XML mapování pro `mw:DynamicList`, kterému se věnuje samostatná kapitola 4.3.7.
Po nastavení požadované podoby datové vrstvy je možné si veškeré nastavení uložit do souboru pro potřeby pozdější modifikace nebo přegenerování datové vrstvy.
Před samotným generováním je nutné ještě vyplnit název namespace, do kterého se veškeré třídy budou generovat a zároveň i adresáře, kam se mají výsledné soubory se zdrojovým kódem uložit. Pro samotné vygenerování pak stačí stisknout jedno tlačítko.
Výsledné soubory odpovídají podobě, jaká byla prezentována v předcházejících kapitolách s tím rozdílem, že všechny generované třídy mají ve své hlavičce uvedeno klíčové slovo `partial`, aby bylo možné třídy jakkoliv rozšiřovat a přesto neztratit možnost je kdykoliv znovu přegenerovat.

4.3.7 Mapování vlastností pro mw:DynamicList

V kapitole 2.2.10 byl popsán důvod, proč je třeba vytvořit mapování mezi názvy vlastností datové vrstvy a zároveň i nastíněno řešení tohoto mapování. Mapování je možné opět generovat aplikací `DataLayerGenerator`, jehož výsledkem je jeden XML soubor. Tento soubor je pak vždy nutné při startu aplikace zaregistrovat a tak umožnit redakčnímu systému využívat datovou vrstvu prostřednictvím daného mapování. Třída, která se o toto mapování stará nese název `DynamicListRegister` a veškeré mapování ukládá do interních statických slovníků. Mapování se dá vytvořit dvěma způsoby: buď pomocí přímého volání statických metod pro zaregistrování konkrétní položky (například metoda `RegisterSource` umožňuje zaregistrovat pod libovolným názvem jeden konkrétní `DataService`) nebo hromadně pomocí jednoho XML souboru. Možná struktura XML může vypadat jako na následující ukázce,

která je převzata z vygenerovaného souboru pro ukázkovou databázi `M2GSample`.

```
<type name="Produkt" alias="Product" defaultPropertyAlias="Name">
  <properties>
    <property name="ID" alias="ID" />
    <property name="Kod" alias="Code" />
    <property name="Nazev" alias="Name" />
  </properties>
  <joins>
    <join name="Poradatel" alias="Organizer" type="Firma" />
    <join name="Katalog" alias="Catalogue" type="Katalog" />
  </joins>
  <filters>
    <filter field="ID" alias="ProductID" setter="SourceCodeThenUrl" />
    <filter field="Kod" alias="ProductCode" setter="SourceCodeThenUrl" />
    <filter field="Nazev" alias="ProductName" setter="SourceCodeThenUrl" />
    <filter field="PoradatelID" alias="OrgID" setter="SourceCodeThenUrl" />
    <filter field="KatalogID" alias="CatalogueID" setter="SourceCodeThenUrl" />
  </filters>
</type>
```

Obr. 55 Ukázka XML mapování datové vrstvy pro tabulku Produkt

Pro zpracování XML stačí zavolat statickou metodu `RegisterFromXML`. Třída `DynamicListRegister` XML mapování zpracovává tak, že si jen jednou projde XML a postupně volá statické metody pro registraci. Tímto je umožněno vytváření tzv. „partial XML“, které již nemusí obsahovat kompletní mapování a mohou tak být použita například k „doregistrování“ některých filtračních parametrů či vlastností. Tato vlastnost registrace je velmi důležitá pro generátor datové vrstvy, jenž umožňuje vygenerovat XML mapování pro generované vlastnosti a filtrační parametry. To se v generátoru provádí tak, že se u příslušných sloupců na záložce „DynamicList“ jen vyplní požadované aliasy (tedy názvy určené pro `mw:DynamicList`) u těch položek, které mají být přístupné pomocí objektu `mw:DynamicList` či `mw:SearchForm`.

Povinnou součástí mapování musejí být objekty využívané jádrem redakčního systému (projekt, URL projektu, stránka, sekce a také podporovaný jazyk). Jsou to tabulky pro stránky, sekce, projekty, URL projektů a tabulku se seznamem podporovaných jazyků. Aliasy pro tyto objekty musejí odpovídat těm, které jsou využívány jádrem redakčního systému. Co všechno má být součástí registrace a s jakými aliasy, je možné vyčíst z ukázkového XML, pro databázi `M2GSample`, které lze nalézt v adresáři `Generated` v assembly `M2GBridge`.

4.4 Jádro redakčního systému

Jádrem redakčního systému MagicContent je několik navzájem provázaných tříd. Následující seznam stručně popisuje jejich zaměření:

- `MagicContentApplication`

Rozšiřuje základní ASP.NET třídu `HttpApplication` a upravuje její chování v metodě `BeginRequest` tak, že jen volá statickou metodu `CreateRequest` třídy `DynamicRequest`.

- `DynamicRequest`
  Rozpoznává URL adresu a v případě nalezení požadované stránky přepíše zpracování požadavku na stránku `Default.aspx`., která má za úkol zpracovávat všechny URL adresy definované v redakčním systému. Webová aplikace tak může obsahovat pouze tuto jednu jedinou stránku.
- `DynamicContentPage`
  Slouží jako předek stránky `Default.aspx`, kde je definována struktura HTML stránky. V události `OnInit` vytváří objekt třídy `DynamicContentParser`, kterému předává dynamický obsah aktuální stránky. Jeho výsledkem je seznam ASP.NET ovládacích prvků, které mají být umístěny do obsahu stránky.
- `DynamicContentParser`
  Parsuje dynamický obsah a stará se o vytváření správných dynamických objektů. Výsledkem zpracování obsahu, je seznam ASP.NET ovládacích prvků, které jsou buď statickým HTML textem, nebo jsou to prvky vrácené dynamickým objektem.
- `UrlManager`
  Slouží k vyhledávání a sestavování URL adres celého webového projektu.

4.4.1 Zpracování requestu a třída UrlManager

Zpracování requestu řídí třída `DynamicRequest` a celý průběh zpracování se dělí na celkem dvě fáze.

První je nalezení URL a tím identifikace aktuální stránky a sekce. Potřebnou funkcionalitu zajišťuje třída `UrlManager`, jenž má na starosti nejen veškerou logiku spojenou s rozpoznáváním URL adres, ale také s jejich sestavováním. Interní datová struktura třídy `UrlManager` je strom, jehož uzly jsou instancemi třídy `UrlNode`. Každý uzel stromu reprezentuje jednu část URL adresy, jak bylo popsáno v kapitole 2.2.3, kde byla také znázorněna jeho podoba na obr. 8. Hledání ve stromě funguje tak, že se vytvoří seznam částí aktuální URL adresy. Tento seznam je reprezentován třídou `UrlPoiner`, která v sobě implementuje zároveň i logiku umožňující snadný pohyb v tomto seznamu. Pokud se celý URL strom skládá pouze se statických částí, je vyhledávání celkem přímočaré a funguje tak, že se s každou úrovní posune ukazatel ve třídě `UrlPointer` směrem dopředu. Po vyčerpání všech částí by měl aktuální uzel stromu obsahovat ID stránky. Pokud tomu tak není, hledaná URL adresa v systému neexistuje. Složitější situace nastává v okamžiku, kdy některé části URL adres jsou dynamické.

Dynamické části URL adres jsou při sestavování stromu převedeny na tzv. dynamické objekty pro přepis URL parametrů. Každý z těchto objektů musí být potomkem abstraktní třídy `DynamicUrlObject` a implementovat tak dvě abstraktní metody – jednu pro překlad URL parametru na textovou hodnotu a jednu pro překlad textové hodnoty zpět na hodnotu URL parametru. Při vyhledávání URL adresy se tak dynamickému objektu předá aktuální kolekce URL parametrů a část právě zpracovávané URL adresy. Dynamický objekt pak buď do kolekce URL parametrů přidá přeloženou hodnotu anebo vrátí informaci, že se předaný text nepodařilo přeložit.

Při vyhledávání se systém v každém uzlu nejprve vždy dívá do slovníku, kde jsou uchovávány odkazy na potomky pod statickým klíčem (tzn. slovník je typu `Dictionary<string, UrlNode>`, kde klíčem je statická část URL adresy). Pokud v tomto slovníku neexistuje žádný klíč odpovídající aktuální hledané část URL adresy, je nutné postupně projít seznam všech částí, které jsou reprezentovány dynamickým objektem. V tomto kroku systém počítá s tím, že ačkoliv lze aktuální část URL adresy přeložit na nějaký URL parametr pomocí jednoho konkrétního dynamického objektu, nelze předpokládat, že bude hledání v této větvi stromu úspěšné. Je nutné tak aplikovat hledání formou jednoduchého back-trackingu, kdy se při neúspěchu znovu zkouší rozpoznat aktuální část URL adresy v dalším z dynamických objektů aktuálního uzlu.

Při sestavování URL adresy na nějakou konkrétní stránku je situace o poznání jednodušší. Stačí díky internímu slovníku vzít přímo konkrétní uzel stromu pro požadovanou stránku a projít strom po úrovních směrem ke kořeni. V každém uzlu se tak získá část URL adresy, která v kořeni dává výslednou URL adresu stránky.

Druhou fází zpracování requestu je zpracování nalezené stránky. Toto je řešeno pomocí přepisu aktuální cesty na stránku `default.aspx` (zavoláním ASP.NET metody `HttpContext.Current.RewritePath`). Této fázi se blíže věnuje následující kapitola.

4.4.2 Zpracování dynamického obsahu

Každou stránku v redakčním systému zpracovává výchozí `aspx` stránka (`default.aspx`). Její implementace je velice jednoduchá. Nemusí mít totiž ani `code-behind` část a stačí, že rozšiřuje základní třídu `DynamicContentPage` z assembly `MagicContent`. Obsahem stránky je základní struktura HTML, ve které je definováno několik `asp:PlaceHolder` objektů, do kterých se má vkládat skutečný obsah stránek.

Implementace třídy `DynamicContentPage` při zpracování stránky v události `OnInit` zjistí z databáze obsah sekce, pod kterou spadá aktuální stránka, a tento obsah předá nově vytvořené instanci třídy `DynamicContentParser` ke zpracování. Jeho výsledkem je seznam objektů typu `asp:Control` (ASP.NET

třída sloužící k vyrenderování HTML na výstup stránky) a případný seznam uživatelských chyb, které se mají umístit na vrchol stránky. Tento seznam `asp:Control` objektů je po zpracování celého obsahu vložen do jednoho z `asp:PlaceHolder` objektů, který je určen pro vložení skutečného obsahu stránky.
Parser postupně zpracovává předaný obsah tak, že jednoduše hledá začátky dynamických objektů – tedy řetězec `<mw:`. Po nalezení této posloupnosti vždy předá její pozici spolu s obsahem třídě `TagParser`, která v sobě implementuje stavový automat, který zpracuje text XML tagu do objektové podoby reprezentované třídou `ParsedTag`.
Syntakticky správně rozpoznaný tag je poté nutné převést na správný dynamický objekt. K tomuto účelu slouží tzv. resolvery. Tyto objekty implementují rozhraní `IDynamicObjectResolver`, které deklaruje následující metody:

```
IResolvedObject ResolveObject(ParsedTag tag);

Control ResolveLiteral(string literalContent);

bool AllowResolvingPropagation { get; }
```

Obr. 56 Rozhraní IDynamicObjectResolver

První metoda `ResolveObject` je klíčem k celému rozpoznávání objektů. Je jí vždy předán celý tag, který byl v obsahu nalezen, a jejím výsledkem je rozpoznaný dynamický objekt. Pokud resolver nedokáže rozpoznat předaný objekt, jednoduše vrací hodnotu `null`.
Parser při zpracování obsahu udržuje interní zásobník resolverů, který na začátku zpracování stránky vždy obsahuje resolver pro základní objekty implementovaný přímo v assembly `MagicContent`. Pro rozpoznávání tagu se vždy nejprve použije resolver na vrcholu zásobníku a pokud ten objekt nerozpozná, vezme se resolver pod ním. Takto se postupuje až na dno zásobníku a pokud ani tam není tag rozpoznán, systém toto vyhodnotí jako chybně zadaný objekt a tuto informaci uloží do seznamu chyb, které jsou určeny k zobrazení uživateli. Resolvery slouží také ke zpracování statického textu, který parser nachází mezi jednotlivými dynamickými objekty. Zpracování tohoto textu vždy musí obstarat resolver na vrcholu zásobníku metodou `ResolveLiteral`. Tato metoda může například jednoduše vrátit instanci ASP.NET třídy `LiteralControl` a nebo zpracovat statický obsah jiným způsobem a vrátit tak hodnotu `null`.
Každý rozpoznaný objekt musí implementovat rozhraní `IResolvedObject` složeného z následujících metod:

```
Control LoadControl();

bool IsSupportingContent { get; }

string GetContent();

IDynamicObjectResolver GetResolver();
```

Obr. 57 Rozhraní IResolvedObject

Nejdůležitější metodou je zde `LoadControl`, která slouží pro vrácení `asp:Control` objektu, který bude sloužit jako prostředek pro vyrenderování HTML, pro který je daný dynamický objekt určen. Některé dynamické objekty mohou být párové a umožnit tak definovat v nich nějaký obsah. Parser se o této skutečnosti může dozvědět díky vlastnosti `IsSupportingContent`. Metoda `GetContent` pak slouží pro případy, kdy uživatel použil objekt, který umí zpracovávat obsah (tedy `IsSupportingContent` vrací `true`), ale použil ho jako nepárovou XML značku – tedy bez obsahu. Parser v tomto případě volá metodu `GetContent` pro získání výchozího obsahu, který následně zpracuje stejným způsobem, jako by byl uveden v obsahu stránky. Poslední metodou je `GetResolver`, která je využívána dynamickými objekty, které rozpoznávají nějaké vnořené objekty. Výsledkem metody může být libovolný resolver, který parser vloží na vrchol zásobníku s resolvery a vyjme ho až při zpracování ukončovacího tagu objektu s obsahem. Některé objekty, jako je například `mw:Template` z objektu `mw:DynamicList` neumožňují vkládat libovolný dynamický objekt do jejich obsahu – jednoduše proto, že obsah šablony se musí opakovat pro každý záznam. Resolver objektu `mw:Template` tak vrací ve vlastnosti `AllowResolvingPropagation` hodnotu `false`, kterou parser interpretuje tak, jako by byl resolver umístěn na dně zásobníku – tedy v případě, kdy místo objektu vrátí metoda `ResolveObjekt` hodnotu `null`, vyhodnotí toto parser jako chybu.

4.5 Dynamické objekty

V předchozí kapitole bylo popsáno, že dynamické objekty jsou libovolné objekty implementující rozhraní `IResolvedObject` a princip, jakým způsobem systém s těmito objekty pracuje. Tato kapitola se věnuje možnostem, jak lze systém MagicContent o tyto dynamické objekty rozšiřovat a jaké prostředky k tomu systém nabízí.

4.5.1 Parametry a třída ObjectParameters

Téměř každý dynamický objekt vyžaduje ke svému fungování různé parametry, jejichž hodnoty jsou zadávány v textové podobě buď přímo jako XML atribut nebo přebírány jako hodnota z URL parametru. O tuto funkcionalitu se stará třída `ObjectParameters`. Při tvoření nového dynamického objektu stačí nadefinovat potomka a všechny požadované

parametry vytvořit jako klasické C# vlastnosti s požadovaným typem, ke kterým je nutné přidat C# atribut s názvem `Param`. Název parametru je stejný jako název vlastnosti a zbylé chování je možné definovat v připojeném atributu. Příklad definice parametrů je vidět na následující ukázce.

```
private class ParametersClass : ObjectParameters
{
  [Param(SetterType.OnlySourceCode, IsReplacedByContent = true,
    Summary = "Obsah tohoto objektu.")]
  public string Text { get; set; }

  [Param(SetterType.OnlySourceCode, IsReplacedByContent = true,
    Summary = "ID bloku, který se má vložit jako obsah.")]
  public int? PageID { get; set; }

  [Param(SetterType.OnlySourceCode,
    Summary = "Pokud je tento parametr nastaven, bude obsah " +
    "tohoto objektu obalen odkazem na danou stránku.")]
  public int? NextPageID { get; set; }
}
```

Obr. 58 Příklad definice parametrů dynamického objektu

Nastavování parametrů na konkrétní hodnoty se provádí pomocí metody `SetValues`, která jako vstup očekává objekt `ParsedTag`. Metoda se postará o převod textových hodnot na požadované typy a zároveň zkontroluje i případné nastavení povinných parametrů. V případě chyby v nastavení vyvolává výjimku `UserException`, kterou vždy zachytí parser a zobrazí tak chybovou hlášku uživateli. Třída `ObjectParameters` slouží zároveň i jako prostředek pro přístup k hodnotám URL parametrů pomocí vlastnosti `UrlParameters`. Ačkoliv se lze k hodnotám URL parametrů dostat i přímo přes statickou vlastnost `Current` třídy `DynamicRequest`, je lepší používat vlastnost poskytovanou třídou `ObjectParameters`, protože je v ní zakomponována logika parametru `UrlIgnore`, kterým může uživatel objektu zakázat čtení nějakého URL parametru.

4.5.2 Základní prostředky pro tvorbu dynamických objektů

Při tvorbě nových dynamických objektů je možné využít některé z připravených komponent, které slouží k jejich zjednodušení.

Nejjednodušší formou dynamického objektu je vytvoření klasického ASP.NET uživatelského ovládacího prvku – tzv. ascx user control. Aby systém dokázal tento dynamický objekt rozpoznávat, je nutné např. v souboru `global.asax` při startu webové aplikace zaregistrovat novou instanci třídy `AspControlsResolver` pomocí volání statické metody `RegisterResolver` na třídě `DynamicContentParser`. `AspControlsResolver` implementuje rozhraní `IDynamicObjectResolver` a slouží pro rozpoznávání objektů ze souborů s příponou „`.ascx`“. V jejich konstruktoru je očekávána cesta k adresáři webové aplikace, kde jsou definovány všechny objekty,

které má resolver rozpoznávat. Názvy dynamických objektů jsou tak určeny názvem souboru (bez přípony). `AspControlsResolver` při obdržení XML tagu pro rozpoznání tak jen vytvoří cestu k `ascx` souboru a pomocí metody `LoadControl` aktuální ASP.NET stránky vytvoří instanci ovládacího prvku. Pokud mají tyto objekty obsahovat i nějaké parametry, je možné tohoto docílit implementací jednoduchého rozhraní `IParametrizedControl`, které obsahuje jedinou metodu `GetParameters`, jejímž úkolem je vytvořit instanci třídy s parametry, která je popsána v předešlé kapitole.

Druhou možností je vytvářet dynamické objekty přímou implementací rozhraní `IDynamicObject`, které je více popsané v kapitole 4.4.2. Pokud má objekt rozpoznávat i nějaký obsah, je doporučeno pro resolver využít jako základní třídu abstraktní `BaseResolver`. V této třídě je totiž implementováno zpracování parametru `Condition`, které slouží pro podmíněné zpracování dynamického objektu. Důležitou vlastností každého resolveru by měla být existence bezparametrického konstruktoru (klidně i privátního), aby bylo možné tento objekt zkonstruovat pro potřeby automatické nápovědy.

K rozpoznávání objektů je možné využít prostředky třídy `ObjectRegister`, která při registraci metody pro vytvoření objektu pod zvoleným názvem zároveň umožňuje jednoduše nadefinovat i nápovědu pro tento objekt. Výhodou použití tohoto objektu je i snadná implementace metody `GetSupportedObjects`, která je jednou z metod rozhraní `IDynamicObjectResolver` a kterou je také nutné překrýt v každém vytvářeném resolveru. Metoda slouží pro podporu automatické nápovědy, kdy každý resolver musí umět vrátit seznam všech objektů, které podporuje. Tento požadovaný seznam je možné získat pomocí stejně pojmenované metody třídy `ObjectRegister`.

Další třídou, která usnadňuje rozpoznávání obsahu, je `AspRepeaterObject`. Třída je potomkem `BaseResolver` a zároveň i implementuje rozhraní `IResolvedObject`. Jedná se tak o resolver, který je zároveň i dynamickým objektem. Pomocí tohoto objektu je možné definovat obsah, který má být šablonou v objektu ASP.NET třídy `Repeater`, a tak docílit opakování libovolného obsahu, který je napojen na serverovou logiku pomocí ASP.NET ovládacích prvků umístěných v šabloně `Repeateru`. Příklad jejího použití je možné najít v implementaci objektu `mw:ProjectForm`, kde byl tento objekt použit pro implementaci objektu `mw:UrlList`, který slouží k definici seznamu URL adres projektu.

Poslední důležitým krokem při vytváření dynamických objektů je přidání atributu `DynamicObject`, ve kterém se jednoduše definuje vše potřebné pro generování automatické nápovědy. Například se zde nastavuje typ třídy s parametry nebo typ resolveru pro rozpoznávání dynamických objektů v obsahu. Pro podporu nápovědy existují v systému ještě další atributy, jejichž

použití je možné najít v jejich dokumentaci. Jedná se například o `AlternativeContent`, `Snippet` nebo také `ParamGroup`.

4.5.3 mw:DynamicList

Dynamický objekt `mw:DynamicList`, jehož použití je popsáno v kapitole 3.4.3, zaujímá v assembly `MagicContent` nemalou část implementace, a tak pro něj bylo vyhrazeno celé namespace `MagicContent.DynamicList`. Základní třídou je nepochybně `DynamicListObject`, která implementuje rozhraní `IResolvedObject` a reprezentuje tak samotný objekt `mw:DynamicList`. Samotný objekt `mw:DynamicList` toho však moc neimplementuje – pouze shromažďuje informace potřebné pro sestavení dotazu z podřízených objektů a v události `OnPreRender` pak zavolá příslušnou metodu pro získání dat z databáze, jejíž výsledek je předán všem podřízeným objektům. Celou logiku z předešlé věty implementuje objekt třídy `DataLayerProvider`, jejíž instanci drží `mw:DynamicList` ve vlastnosti `DataLayer`, kterou využívají všechny podřízené objekty k registraci všech vlastností, které musejí být získány z databáze. Toto je možné provádět pomocí metody `RegisterProperty`, které stačí předat název požadované vlastnosti a tato se za pomoci třídy `DynamicListRegister` přidá do interního `DataServisu` pro danou tabulku určenou parametrem `Source` u `mw:DynamicList`.

O vytváření podřízených objektů se stará třída `TemplateResolver`, jejíž název napovídá, že implementuje rozhraní `IDynamicObjectResolver` a slouží tak k rozpoznávání objektů které se používají právě uvnitř šablony (tedy obsahu) dynamického objektu `mw:DynamicList`.

Každý z podřízených objektů musí být potomkem abstraktní třídy `BaseBlock`, která implementuje základní logiku potřebnou při vytváření objektů, které mají renderovat obsah na základě předaného objektu `DataItem` reprezentující jeden záznam vrácený databází. Všichni rozpoznávaní potomci musejí být zaregistrování ve třídě `TemplateResolver` pomocí statické metody `RegisterBlock`, které stačí předat název objektu a metodu pro jeho zkonstruování (nejčastěji lambda-metoda volající konstruktor). Touto statickou metodou je tak umožněno libovolné rozšiřování `mw:DynamicList` o potřebné další objekty. Třída `TemplateResolver` vždy po rozpoznání objektu a jeho vytvoření volá metodu `Inicialize` implementovanou v základní třídě `BaseBlock`. Této metodě je vždy předávána reference na základní objekt `mw:DynamicList`, který je nastaven do vlastnosti `MainObject`. Každý z podřízených objektů tak má tak přímý přístup například k vlastnosti `DataLayer`. Tímto je umožněno, aby každý z podřízených objektů mohl upravovat podobu dotazu přidáváním požadovaných vlastností.

Třída `BaseBlock` definuje dvě abstraktní metody: `CreateParameters` a metodu `RenderData`. První slouží k vytvoření instance třídy s definicí parametrů objektu, na kterém při inicializaci volá metodu `SetValues`. Potomci se tak

nemusí starat o toto volání a mohou rovnou přistupovat ke svým parametrům pomocí vlastnosti `Parameters`. Druhá metoda je klíčová pro celý proces renderování výstupu a její hlavička vypadá následovně:

```
public abstract void RenderData(HtmlTextWriter writer, DAOTier.DataItemTree dataSource);
```

Obr. 59 Hlavička metody pro renderování obsahu v podřízených objektech mw:DynamicList

Jako první parametr jí je vždy předán `HtmlTextWriter`, který slouží jako prostředek pro renderování výstupu. Druhý parametr pak reprezentuje záznam, jehož data by měl daný objekt použít při renderování obsahu. Jedná se o objekt třídy `DataItemTree`, která jen rozšiřuje třídu `DataItem` o seznam podřízených záznamů a jedná se tak vlastně o uzel stromu.
Zvláštní roli v procesu renderování hraje podřízený objekt `mw:Template` (třída `TemplateBlock`), který stejně tak jako všechny ostatní dědí od `BaseBlock`. Jeho implementace metody `RenderData` je velice jednoduchá a spočívá v iteraci všech podřízených záznamů parametru `dataSource` a pro každý záznam volá na všech dynamických objektech ve svém obsahu metodu `RenderData`, které předává právě iterovaný záznam. Metoda `RenderData` objektu `mw:Template` je tak centrem celého renderovacího procesu objektu `mw:DynamicList`.

4.5.4 Dynamické objekty pro přepis URL parametrů

Zvláštním typem dynamických objektů jsou dynamické objekty pro přepis URL parametrů. Jejich úkolem je implementovat logiku, která je schopna převést nějakou textovou hodnotu získanou z URL adresy do podoby hodnoty pro URL parametr a opačně z hodnoty URL parametru vytvořit textovou hodnotu určenou jako část URL adresy. Základní třídou, od které musejí všechny dynamické objekty pro přepis URL parametrů dědit, je abstraktní `DynamicUrlObject`. Protože při zpracování URL se nepoužívají žádné resolvery, je nutné každý dynamický objekt, který má systém rozpoznávat, zaregistrovat pomocí statické metody `RegisterObject` třídy `DynamicUrlObject`. Je nutné jí vždy předat název objektu a metodu pro jeho vytvoření.
Implementace objektu je pak velice jednoduchá a stačí přepsat následující tři abstraktní metody:

```
DynamicUrlObjectParameters CreateParameters();

List<string> GetUrlPartsOverride(DataAccess dataAccess, UrlParameters parameters);

bool ProcessUrl(DataAccess dataAccess, TaggedUrlParameters parameters, UrlPointer urlPointer);
```

Obr. 60 Metody určené pro přepsání v dynamických objektech pro přepis URL parametrů

První metoda slouží pro vytvoření třídy s definicí parametrů. Jak hlavička napovídá, je nutné, aby každá z nich dědila od třídy `DynamicUrlObjectParameters`, která definuje dva základní parametry Prefix a

Postfix, které jsou určeny jako text, který se bude automaticky přidávat před/za text každé části URL.
Druhá metoda slouží pro vytvoření seznamu částí určených do URL adresy. Metoda k tomu dostává veškeré potřebné prostředky – tedy objekt pro přístup k databázi a seznam hodnot všech parametrů.
Poslední metoda je určená pro opačný postup – tedy z URL adresy získat hodnoty URL parametru/ů. Stejně jako předešlé metodě je předáván objekt pro přístup k databázi jako první parametr. Parametr metody s názvem `parameters` naopak slouží pro případné uložení hodnoty URL parametru, která odpovídá URL části adresy. Je to objekt třídy `TaggeedUrlParameters`, která interně udržuje zásobník nalezených hodnot, aby bylo možné provádět případný back-tracking při vyhledávání ve stromě (viz kapitola 4.4.1). Posledním parametrem je objekt třídy `UrlPointer`, který umožňuje získat aktuálně zpracovávanou část URL adresy a případně umožnit zpracovat i více než jednu část.

4.6 Implementace aplikace MagicCoding

Aplikace `MagicCoding` se skládá celkem ze dvou assembly, které jsou obě sestavovány do spustitelné podoby `exe` souborů. Jedná se o assembly `MagicCoding` a `MagicCodingUrlHandler`.
V první je implementována hlavní logika celé aplikace v souboru `MainWindow.xaml.cs`. Implementace je založena na kontrolování jakékoliv změny ve file-systému, kdy obsah stránek či sekcí je ukládán do souborů, při jejichž změně se vyvolá událost, která změnu uloží na server. O kontrolu změn ve file-systému se stará objekt .NET třídy `FileSystemWatcher`, která umožňuje navázat události při přidání či změně souboru. Spojení se severem je zajištěno pomocí `WebService` metod, které jednoduše umožňují předávání informací pomocí HTTP protokolu, čímž je zajištěna funkčnost i v případě, kdy je například klientská část aplikace schována v síti za firewallem. `WebService` metody jsou implementovány ve webové aplikaci v souboru `AdministrationService.cs` a jsou zpřístupněny pomocí `AdministrationService.asmx`.
Webové prohlížeče z důvodů bezpečnosti neumožňují spouštět jakýkoliv spustitelný kód přímo a nelze tak jednoduše vyvolat jakoukoliv akci, která by umožnila například předat informace o tom, která stránka či sekce se má editovat či stáhnout. Jediná možnost jak spustit aplikaci z prostředí prohlížeče je využití tzv. obsluhy URL protokolu (URL Protocol Handler). V systému Windows lze zaregistrovat libovolnou aplikaci, která se má spustit po kliknutí na odkaz, který vyžaduje zpracování pomocí jiného protokolu (než je například HTTP či FTP). V aplikaci `MagicCoding` tak existuje tlačítko, které vytvoří "`.reg`" soubor, ve kterém jsou definovány klíče pro zaregistrování protokolu s názvem `MagicCoding`. Jako obsluhu tohoto protokolu je vždy

nastavena cesta k aplikaci `MagicCodingUrlHander`. Po úspěšné registraci "`.reg`" souboru do registrů systému Windows je pak této aplikaci vždy předán celý odkaz, na který bylo v prohlížeči kliknuto, v podobě parametrů z příkazové řádky. Tlačítka ve webovém administračním rozhraní jsou tak ve skutečnosti odkazy obsahující informace o tom, jaký objekt se má editovat, spolu se základní URL adresou webové aplikace. Příklad takového odkazu je na následujícím obrázku:

```
<a href='MagicCoding:P93_Prehled_projektu%7cwww_nadrasky_cz%7chttp%3a%2f%2fwww.nadrasky.cz%2fmc'>
```

Obr. 61 URL adresa využívající protokol MagicCoding

Aplikace `MagicCodingUrlHander` vstupní parametry vždy zpracuje tak, že vytvoří v adresáři, který je monitorován aplikací `MagicCoding`, nový soubor obsahující identifikaci objektu, který se má stáhnout a zároveň i textový soubor, ve kterém je uložena URL adresa webové aplikace. Tímto způsobem je tak jednoduše zajištěna komunikace mezi webovým prohlížečem a aplikací `MagicCoding`.

5 Závěr

Hlavním cílem práce bylo vytvořit redakční systém, který by umožňoval napojení na kteroukoliv databázi a jednoduše poskytoval prostředky k vypisování libovolných dat z této databáze. Stávající řešení, která byla popsána v úvodu, postrádají právě možnost napojit se na již existující databázi. A pokud toto umožňují, je vždy třeba implementovat kód datové vrstvy ručně. Při vývoji a návrhu redakčního systému MagicContent jsme se proto primárně zaměřovali na takové funkce, které existující redakční systémy buď úplně postrádají anebo je implementují jen z malé části. Výsledkem je velmi jednoduše použitelný a hlavně libovolně rozšiřitelný systém, který dominuje v situacích, na které se jiné systémy příliš nehodí. Jsou to právě situace popsané v úvodu – tedy například tehdy, kdy máme nějaký robustní informační systém, který je nasazován u různých zákazníků, kteří zároveň požadují vytvořit webové prezentace, které čerpají data z tohoto systému. Díky dynamickým objektům (tzn. předem připraveným komponentám), je možné velmi rychle postavit webový portál obsahující i složitější aplikační logiku bez nutnosti jí znovu programovat a je tak znovu využit aplikační kód, který byl napsán již v minulosti pro jiného zákazníka. Jsou tím tak ušetřeny nemalé náklady při prvotním nasazování systému. Zároveň je umožněno celý webový portál vytvářet i bez znalosti programování, a tak je možné přesunout veškerou práci na HTML kodéry – tedy zaměstnance, kteří nejsou programátory, ale umějí dobře značkovací jazyk HTML a CSS. To je další velká úspora, jelikož jsou tito zaměstnanci často levnější než programátoři vyvíjející aplikační logiku.

Důležitou roli v celkové flexibilitě systému hraje tzv. šablonovací systém, který řeší situace, kdy je nutné pro každého zákazníka vytvářet podobě fungující webové portály, ale s úplně odlišným vzhledem. Šablonovací systém v sobě skrývá možnost jakéhosi „prototypování“ vzhledu i chování, ale zároveň umožňuje i tento standardní vzhled i chování upravovat definováním šablony pro konkrétní komponentu – tedy dynamický objekt. Pokud programátor komponenty při jejím vývoji nadefinuje například formulář umístěný v tabulce s přesně rozloženou strukturou uživatelských ovládacích prvků, HTML kodér, který tuto komponentu bude chtít použít, buď může rovnou využít standardní vzhled a rozložení anebo si může libovolně tento vzhled a rozložení ovládacích prvků přizpůsobit.

Nejdůležitější komponentou systému je ale bezesporu dynamický objekt `mw:DynamicList`, s jehož pomocí lze na webových stránkách vypisovat libovolná data z databáze bez nutnosti cokoliv programovat. Díky této komponentě, která jen na základě šablony umí na pozadí sestavit SQL dotaz, je možné reagovat na změnové požadavky od zákazníka týkající se obsahu webu téměř okamžitě, protože není nutné zasahovat do zdrojových

kódů aplikace a procházet tak celým vývojovým procesem, který končí vystavením nové verze aplikace. V systému MagicContent lze tyto změnové požadavky řešit rovnou.

Všechny výše popsané výhody jsou již ověřené v praxi. Redakční systém MagicContent byl totiž zakomponován do informačního systému Magic2G (pod obchodním názvem is>content) a s jeho pomocí byl již postaven nemalý počet webových portálů pro tuzemské cestovní kanceláře. Jako příklad jeho reálného nasazení může například být web http://www.ckalex.cz nebo také http://www.kovotour.cz.

Součástí systému MagicContent je také zajímavá aplikace MagicCoding. Ač její implementace nebyla nikterak složitá, je velmi užitečnou pomůckou při tvorbě obsahu, kdy si uživatel může vychutnávat všech pokročilých funkcí libovolného desktopového editoru i přesto, že editovaný obsah je ve skutečnosti ukládán na server.

Datová vrstva, jež byla implementována jako součást práce a měla sloužit jen jako pozadí pro dynamický objekt `mw:DynamicList`, byla nakonec navržena tak, aby mohla být použitelná i jako samostatný celek. Jednoduchost jejího použití zároveň s možností vygenerovat zdrojový kód popisující strukturu konkrétní databáze z ní dělá zajímavou alternativu k existujícím technologiím jako je například Hibernate nebo LINQ to SQL oproti nimž má tři velké výhody: První je, že programátor má opravdovou kontrolu nad podobou SQL dotazů, které jsou posílány do databáze. Druhou je to, že při každém přístupu k datům z databáze si programátor definuje seznam vlastností (sloupečků), které bude vyžadovat, a tak zpracování dotazů, které obsahují jen tyto sloupce, je často mnohem rychlejší než při ekvivalentním použití technologie Hibernate. Poslední výhodou je velká univerzálnost při definici filtračních parametrů, které umožňují rychle a jednoduše omezovat záznamy pomocí skládaných výrazů ve `WHERE` části SQL dotazu.

Implementovaná datová vrstva je stejně jako redakční systém MagicContent využívána v informačním systému Magic2G, kde postupně nahrazuje původní datovou vrstvu založenou na tvorbě uložených procedur a jejich volání pomocí ADO.NET technologie. Její nasazení zjednodušilo a urychlilo podstatnou část vývoje nových komponent díky možnostem jejího generování a celkové flexibilitě tvorby filtračních parametrů.

Díky domu, že při vývoji redakčního systému MagicContent jsme se zaměřovali jen na ty části, které jiným systémům chybí, nezbyl bohužel čas na některé části, jenž často brání reálnému nasazení. Touto částí je například absence modulu pro přihlašování uživatelů a s tím souvisejícího zabezpečení webového administračního rozhraní, které je tak přístupné komukoliv, kdo zná jeho URL adresu. Ačkoliv se jedná o velmi důležitou část

každého redakčního systému, cílem bylo reálné nasazení v systému Magic2G, který ale tento modul již obsahuje.

Další oblastí, na kterou nebyl při vývoji kladen dostatečný důraz, je podpora pro tzv. kešování. Díky tomu, že veškerý obsah zpracovává jedna stránka `default.aspx`, není možné využít kešovacího mechanizmu poskytovaného technologií ASP.NET. Zároveň ani datová vrstva neposkytuje žádnou možnost, jak umožnit uchovávat výsledek často používaných dotazů v paměti serveru. Škálovatelnost při nasazení je tak tímto značně omezena. Implementace této funkcionality ale již byla nad rámec obsahu této práce.

Všechny cíle, které měl výsledný systém splňovat a které byly blíže rozepsány v úvodu, systém MagicContent splňuje. Jeho nesporné výhody tak spočívají ve znuvupoužitelnosti jednou napsaného kódu, v jednoduchosti vytváření dynamického obsahu, v oddělení aplikační logiky od výsledného vzhledu HTML stránek díky šablonovacímu systému, v rychlém nasazení nad libovolnou databází díky implementovanému generátoru datové vrstvy a v neposlední řadě také v tom, že vytváření i složitých webových portálů obsahující dynamický obsah vypisovaný z databáze zvládne i uživatel, který umí pouze HTML a CSS bez nutnosti cokoliv programovat.

6 Reference

1. .NET Web Content Management System | Kentico CMS for ASP.NET. [Online] Kentico Software. http://www.kentico.com/.

2. **Nathan, Adam.** *Windows Presentation Foundation Unleashed.* místo neznámé : Sams Publishing, 2006. 0-672-32891-7.

3. **Haverbeke, Marijn.** *CodeMirror.* [Online] http://codemirror.net/.

4. **Walter, Aarron.** *Building Findable Websites: Web Standards, SEO, and Beyond.* místo neznámé : New Riders Press, 2008. 0321526287.

5. Search Engine Optimization. *Wikipedie.* [Online] http://cs.wikipedia.org/wiki/Search_Engine_Optimization.

6. Wikipedia. *List of ISO 639-1 codes.* [Online] http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes.

7. Wikipedia. *ISO 3166-1.* [Online] http://en.wikipedia.org/wiki/ISO_3166-1.

7 Přílohy

7.1 Popis obsahu přiloženého CD

- `Obsah.txt` – popis obsahu CD
- `IIS7_instalace_a_konfigurace.pdf` – popis instalace Internet Information Services 7 na PC s OS Windows 7 Professional 64 bit
- `Diplomova_prace.pdf` – tento dokument
- `[DataLayerGenerator]` – obsahuje soubor `DataLayerGenerator.zip` se zkompilovanou aplikací DataLayerGenerator
- `[Documentation]` – adresář s vygenerovanou dokumentací
- `[M2GSampleDatabase]` – záloha ukázkové databáze
- `[MagicCoding]` – obsahuje soubor `MagicCoding.zip` se zkompilovanou aplikací MagicCoding
- `[PrecompiledWeb]` – zkompilovaná webová aplikace připravená pro vystavení na webový server
- `[Source]` – adresář se zdrojovými kódy

7.2 Dokumentace vygenerovaná z XML komentářů v kódu

Součástí práce je i vygenerovaná dokumentace z XML komentářů v kódu. Lze ji najít v adresáři `Documentation` v souboru `index.html`.