Wstęp

* Dlaczego IDE jest ważne.
* Dlaczego język jest ważny
* Gdy w pracy napisane jest „ogólnym rozwiązaniem” lub „w większości frameworków” oznacza to żę zostały sprawdzone frameworki : Zend 1,2 ; ASP.NET MVC, Grails, Symfony 1, 2
* To do ASP.NET 4 MVC VS ASP.NET (aka MVC vc MVP/MVVM itp.)
* ViewModel w MVC a ViewModel w MVVM
* Adnotacje to atrybuty w c# (chyba najlepiej jak najwcześniej –uzte są w walidacji i w routingu)

Todo: zamienić wszystkie asp.net 4 na 5

Todo: routing, w asp.net 5 dodali routowanie przez adnotacje atrybuty  
Todo: Formularze przenieść z kontrolera za widok

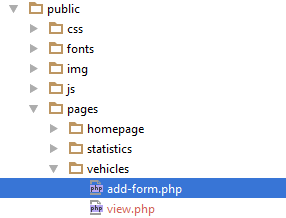
Trasowanie

Todo: przejrzeć wszystko i sprawdzić czy adres zawarty w roucie nazywany jest szablonem trasy

Trasowanie (z ang. Routing) jest mechanizmem mapującym adresy URL do odpowiadającym im akcjom kontrolera. Kiedy użytkownik klika w link, przechodzi do strony poprzez bezpośrednie wpisanie adresu lub następuje automatyczne przekierowanie rozpoczyna się nowe połączenie z serwerem. W założeniach protokołu http adres URL zawierał w sobie ścieżkę reprezentującą fizyczne położenie pliku na dysku.

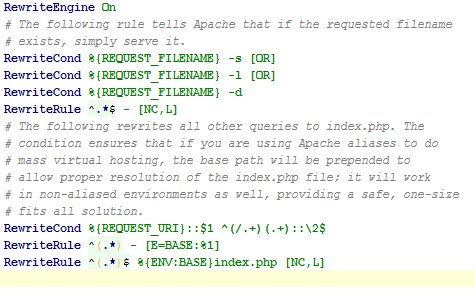
Takie rozwiązanie, choć w pełni dające możliwość zbudowania kompletnej aplikacji, posiada szereg wad, m.in.:

* Eksponowanie klientowi struktury i hierarchii plików oraz katalogów na serwerze. O ile prawdą jest, że dla poprawnie zabezpieczonej aplikacji powinno być obojętne czy struktura plików jest jawna czy nie, to jednak jej znajomość przez hakera jeszcze przed atakiem pozwala na jego sprawniejsze i szybsze wykonanie.
* Trudność zaimplementowania dispachowania {TODO: język} akcji kontrolera.
* Bezpośredni dostęp poprzez ścieżkę narzuca problem ograniczania dostępu do konkretnych plików, które nie powinny być dostępne dla wszystkich użytkowników, bądź tylko dla wybranej grupy.
* Nieatrakcyjny i nieprzyjazny wygląd adresów URL dla użytkownika.



*Rys. 1 Aby uruchomić przedstawiony plik add-form.php należałoby przejść do adresu /public/pages/vehicle/add-form.php*

Mechanizm trasowania umożliwia obejście tego problemu. Żądanie do serwera jest przechwytywane, a następnie kierowanie do warstwy trasowania. Tam tzw. Trasownik (z ang. Router) jest odpowiedzialny za przeanalizowanie żądania, nie tylko pod kątem podanego adresu, ale również użytej metody http. Sposób w jaki jest to dokonywane zależy od serwera. Przykładowo Serwer Apache, który używany jest przy tworzeniu stron PHP, używa pliku .htaccess, który służy do konfigurowania katalogu, w którym się zawiera.



*Rys. 2 Domyślny plik .htaccess używany przez Zend Framework 2 do przechwytywania wszystkich żądań.*

Czym jest trasa?

Trasa (ang. Route) (z ang. „szlak” lub „trakt”) definiuje jeden lub grupę adresów URL. Określa również liczne parametry jakie charakteryzują dany adres. Mechanizm trasowania po otrzymaniu do przetworzenia wysłanego żądania do serwera przegląda kolejno skonfigurowane trasy. Poszukuje tak długo, aż uda mu się dopasować parametry żądania do któregoś ze skonfigurowanych tras. Jeśli nie uda się dopasować żadnej trasy wtedy zwracany jest kod błędu odpowiedzi 404.

Dopasowywanie tras

Podstawowa trasa zazwyczaj składa się z:

* nazwy – każda trasa posiada unikatową nazwę identyfikującą ją
* adresu – ciąg znaków reprezentujący adres URL kierujący do trasy, może być parametryzowany
* ograniczenia parametrów – narzucają wymaganie względem parametrów adresu
* ograniczenia metody żądania – określa, jakie metody żądania muszą zostać użyte, aby użyć tej trasy
* kontroler i akcja – mówi dokąd ma kierować dana trasa jeśli jest spełniona.

TODO: Przykładowa trasa

Ograniczenia tras

TODO: constrainty

Ograniczenia (ang. constraints) są to dodatkowe reguły jakie możemy narzucić na trasę, aby była ona spełniona.

Są dwa rodzaje ograniczeń:

* Ograniczenia na parametry trasy
* Ograniczenia na metodę żądania

Ograniczenia parametrów

Adres trasy może składać się z wielu członów, niektóre z nich mogą być parametrami, tzn. mogą przyjmować różne wartości. Czasami możemy dopuścić, aby wartość danego parametru była dowolna. Jednak czasami chcemy, aby przyjęła jedną z kilku możliwych wartości, była określonego typu np. liczbą całkowitą, albo była opcjonalna tj. podanie jej w adresie jest niewymagane.

Dobrym przykładem jest adres, który zawiera w sobie rok, np.:

TODO: adres z rokiem

/utwory/2010

Powyższy adres kieruje do listy utworów w roku 2010. Ostatni człon jest parametrem, który może przybierać różne wartości, jednak wszystkie one będą liczną całkowitą – nie będą zawierały Rok jest czterocyfrową liczbą całkowitą. Zakładając, że nazwaliśmy owy parametr „year” ograniczenie na niego w APS.NET MVC 4 wyglądałoby następująco:

year = @"\d{4}"

Gdzie wartość w cudzysłowie jest wyrażeniem regularnym jakie musi zostać spełnione. Litera „d” reprezentuje liczbą całkowitą, a liczba 4 dozwoloną długość wartości. Umiejętność stosowania tych ograniczeń sprowadza się w głównej mierze do znajomości wyrażeń regularnych. Jeśli żądanie będzie pasowało to adresu trasy, ale ograniczenia parametrów nie będą spełnione, wtedy zostanie zwrócony kod 404 z adekwatną informacją. We badanych frameworkach stosowanie ograniczeń parametrów wygląda tak samo, i sprowadza się do przyporządkowania wyrażenia do parametru.

Ograniczenia metod żądania

Dwiema głównymi metodami żądania używanymi podczas tworzenia serwisów WWW są GET i POST. W typowej aplikacji GET będzie używane jako standardowa metoda do poruszania się po serwisie, a POST użyte zostanie do wysyłania formularzy. Jeśli posiadamy akcję, która wyłącznie będzie obsługiwać formularz wtedy możemy tak skonfigurować trasę, aby jedynie zezwalała na metodę POST. Inne metody są rzadko używane w aplikacjach MVC. Przypadkiem, w którym pozostałe metody często są stosowane jest sytuacja, gdy nasza aplikacja pełni rolę serwisu REST.

Przykład ograniczenia trasy do metody GET w frameworku ASP.NET MVC 4:

new { httpMethod = new HttpMethodConstraint("Get") }

Definiowanie tras

Wszystkie adresy URL, których dostarcza do użytku aplikacja MVC powinny być zdefiniowane. Istnieje wiele sposobów na definiowanie tras. Niektóre frameworki umożliwiają tworzenie zasad trasowania na kilka sposobów, inne tylko jeden.

Do najpowszechniejszych sposobów definiowania tras należą:

* Definiowanie poprzez plik konfiguracyjny
* Definiowanie bezpośrednio poprzez kod programu
* Definiowanie poprzez adnotacje

Każdy sposób ma swoje wady i zalety, jednak przed porównaniem należy się głębiej z nimi zapoznać. Do porównania użyjemy przykładowej trasy, który zostanie zdefiniowana różnymi sposobami.

Cechy trasy:

* będzie odpowiedzialny do kierowania na stronę z listą samochodów
* samochody będą podzielone na kategorie
* zawiera paginację

Szablon adresu, którego chcemy użyć:

TODO: samochody/{category}/{page}

Powyższy szablon składa się z trzech sekcji:

* samochody – wymagamy, aby ta część adresu była niezmienna
* category – ta sekcja jest parametrem - dowolnym ciągiem znaków reprezentującym kategorię
* page – ta sekcja również jest parametrem - numerem strony, jest liczbą całkowitą, parametrem opcjonalnym (zakładamy, że jeśli nie zostanie podana to kierujemy na pierwszą stronę)

Przykładowe adresy spełniające powyższe wymogi:

samochody/terenowe/5

samochody/osobowe

samochody/ciężarowe/17

Chcemy, aby ta trasa kierowała do określonej akcji kontrolera:

* kontroler: Cars
* akcja: List

Posiadając powyższe wymagania jakie ma spełnić trasa możemy przejść do porównania różnych sposobów jej definicji.

Definiowanie poprzez plik konfiguracyjny

Ten sposób zakłada umieszczenie definicji tras w odrębnym pliku. W zależności od możliwości frameworku może być więcej plików niż jeden. Jego zawartość reprezentowana jest w określonym formacie np. XML, yaml.

TODO: przykład pliku konfiguracyjnego

Definiowanie bezpośrednio poprzez kod programu

W tym rozwiązaniu wykorzystuje się dostarczone klasy i ich metody do konfigurowania tras. Kod lub odwołanie do niego zazwyczaj należy umieścić w odpowiednim miejscu aplikacji (np. w bootstrapie).

TODO: przykład definiowania przez kod

Definiowanie poprzez adnotacje

Adnotacje (ang. annotations) są to metadane dołączone do kodu programu. Są atrybutami nadającymi specjalne cechy klasom, ich metodom i polom. Niektóre frameworki umożliwiają ich użycie do definiowania tras. Specjalna adnotacja wraz ze wszystkimi parametrami opisującymi trasę dołączana jest wprost do akcji kontrolera.

Porównanie popularnych rozwiązań

Każdy z tych sposobów ma swoje wady i zalety. Najlepszym sposobem ich porównania, będzie wybranie obszarów procesu tworzenia programu, na które mają wpływ i rozpatrzenie jak się w nich sprawują.

1. Przeglądanie i wyszukiwanie tras

Podczas tworzenia aplikacji może dojść do sytuacji, w której programista przejdzie do jakiejś podstrony serwisu. Jeśli ma za zadanie zmodyfikowanie zachowania tejże strony zapewne będzie chciał przejść akcji kontrolera, która została wywołana, aby wyświetlić stronę. Jeśli projekt jest mały, to szukanie może nie być problematyczne, ale w sytuacji gdy w dużym serwisie istnieją setki tras to odnalezienie odpowiedniej może być kłopotliwe. W przypadku dostępnych narzędzi do debugowania problem może być łatwo rozwiązany. Przykładem są dodatki, które dołączają do zawartości strony pasek, który zawiera pomocne informacje przy tworzeniu strony (patrz rys. TODO\_RYS\_001).



Rys. 1 Dodatek ZFDebug do Zend Framework 2 dodaje pasek na dole strony, na którym widać jaka akcja jakiego kontrolera jest aktualnie wykorzystywana.

Niestety czasami może nastać sytuacja, w której taki dodatek nie będzie pokazany. Przykładem może być zapytanie AJAX do serwera, które zwraca jedynie informacje w formacie JSON zamiast kodu HTML, który jest wymagany do działania narzędzia. Inna sytuacja występuje, gdy strona nie działa, a błąd powoduje niezaładowanie widoku. W takich przypadkach jeśli programista nie pamięta lub nie wie, do której akcji doprowadza adres URL, wtedy musi odnaleźć ją własnoręcznie. Jeśli zna strukturę projektu wtedy może się domyślić gdzie leży kontroler, jeśli nie, wtedy musi ręcznie przejrzeć projekt. Jeśli adres jest stały tzn. nie posiada żadnych parametrów wtedy wystarczy zwykłe wyszukiwanie. Jednak gdy adres posiada parametry wtedy odnalezienie odpowiedniej trasy może trudne.

* Definiowanie poprzez plik konfiguracyjny

Zaletą tego podejścia jest to, że dokładnie wiadomo, od czego rozpocząć poszukiwanie. Jeśli plik jest jeden to od razu wiemy, że w nim znajduje się odpowiedź. Jednak, niektóre umożliwiają i zachęcają do dzielenia aplikacji na moduły. Przykładem jest Zend Framework 2, w którym zaleca się dużą granulację modułów. Biorąc pod uwagę fakt, że w dużym serwisie takich modułów mogą być dziesiątki, a każdy moduł może posiadać konfigurację własnych tras to przeszukiwanie może potrwać długo. Jedyną pomocą jest znajomość wyrażeń regularnych i możliwość ich użycia do przeszukiwania projektu.

* Definiowanie bezpośrednio poprzez kod programu

Definiowanie poprzez kod programu pozwala na dowolność organizowania definicji tras w programie. W zależności od skomplikowania przyjętych metod odnalezienie odpowiedniej trasy może być łatwe jak w sposobie definiowania poprzez plik. Jednak, zastosowane konstrukcje programistyczne, choć pomocne w innych sytuacjach, mogą znacząco utrudnić poszukiwania.

* Definiowanie poprzez adnotacje

Todo: poprawić wcięcie powyżej

Choć adnotacje wydają się eleganckim rozwiązaniem, które umieszcza definicje tras przy kontrolerach, których dotyczą to powodują jeden problem. Wyszukiwanie trasy pasującej do danego adresu URL jest uciążliwe ponieważ należy przeszukać wszystkie kontrolery, których zapewne będzie jeszcze więcej niż modułów aplikacji (zazwyczaj moduł posiada więcej niż jeden kontroler). Dodatkowym utrudnieniem jest to, że adnotacje w kontrolerze przeplatają się z metodami i z innymi adnotacjami. Powoduje to wyszukiwanie konkretnej treści w szumie informacji, co jest bardzo uciążliwe.

Podsumowując, rozwiązania poprzez plik konfiguracyjny i kod programu w sytuacji, gdy nie zastosowano skomplikowanych konstrukcji, są wygodnymi i łatwymi sposobami, pozwalającymi w miarę sprawnie przeszukiwać trasy. Adnotacje z powodu dowiązania do akcji są porozrzucane po kontrolerach i ich odnajdowanie może być kłopotliwe.

TODO: definiowanie poprzez adnotacje

1. Sprawdzenie użyć akcji

TODO

Definiowanie tras w wybranych frameworkach

ASP.NET MVC 4.0

[todo – przejrzeć dokument i sprawdzić czy nigdzie napisane nie ma, że ASP.NET MVC nie ma trasowania przez adnotacje (atrybuty), w MVC 5 dodali to już]

Jedynym gotowym rozwiązaniem do definiowania tras w tym frameworku jest ich tworzenie poprzez kod. Wraz z utworzeniem nowej aplikacji zostaje stworzona metoda statyczna „RegisterRoutes”.

TODO: Przykład trasy w MVC 4.0

Do zmiennej „routes” możemy dodawać kolejno zdefiniowane trasy. Najprostszym sposobem jest definiowanie wszystkie jawnie w tej metodzie. W takim wypadku plik z tą klasę będzie pełnił tę samą rolę. Zaletą tego podejścia będzie przejrzystość tras – wszystko znajduje się w jednym miejscu. Problem może powstać, gdy plik zacznie się rozrastać, wtedy prawdopodobnie dobrym rozwiązaniem podzielenie tras na mniejsze grupy i umieszczenie ich w oddzielnych klasach. ASP.NET MVC 4.0 nie dostarcza gotowego rozwiązania konfigurowania tras w oddzielnym pliku. Było to możliwe w wersji 2.0 frameworka, gdzie trasy umieszczano w pliku „web.config”. Plik ten jednak jest głównym plikiem konfiguracyjnym, a umieszczanie definicji tras obok pozostałej konfiguracji utrudnia odczytywanie zawartości. Odseparowanie tras było, więc dobrym pomysłem pod względem zwiększenia czytelności konfiguracji.

Jako, że rozwiązanie bazuje na czystym kodzie programu istnieje pełna możliwość stworzenia własnych komponentów, które będą tworzyć trasy w dowolny sposób. Nic nie stoi na przeszkodzie, aby stworzyć silnik odczytujący konfiguracje tras z pliku lub poprzez adnotacje tak jak w innych frameworkach (drugie rozwiązanie wymaga bardzo dobrej znajomości języka C# i platformy .NET). Od twórcy zależy jak bardzo przejrzysta będzie jego implementacj. Wszystkie definicje ładowane są do pamięci podczas uruchamiania aplikacji podczas startu serwera także wszelkie czasochłonne operacje wykonywane są tylko raz . W wypadku aplikacji, której wymagania sprawiają, że gotowe mechanizmy trasowania są niewystarczające programista może napisać własny silnik dopasowywania tras. W tym celu wystarczy stworzyć implementację dwóch interfejsów IHttphandler oraz IRouteHandler.

Todo: biblioteka z routowanie atrybutami

<http://blogs.msdn.com/b/webdev/archive/2013/10/17/attribute-routing-in-asp-net-mvc-5.aspx>

TODO: własna implementracja:  
https://www.simple-talk.com/dotnet/.net-framework/asp.net-mvc-routing-extensibility/

ZEND FRAMEWORK 2

Zend Framework 2 używa plików konfiguracyjnych do definiowania . Jest to framework, który stara się silnie wspierać modułowość. Z tego powodu może istnieć wiele plików konfigurujących trasy – po jednym pliku na każdy moduł, który korzysta z warstwy kontrolera. Każdy moduł powinien zawierać plik o nazwie „module.config.php”. Plik obok definicji tras zawiera inne dane konfiguracji modułu takie jak fabryki, sterowniki ORM, menadżer serwisów, widoku. Teoretycznie taki plik może się rozrosnąć, jednak kiedy stosuje się zalecaną konwencję dzielenia aplikacji na moduły rozmiar pliku nie powinien nigdy być nadmierny. Należy nadmienić, że mimo, iż rozwiązanie stosuje konfigurowanie poprzez plik, to nie on zapisany w żadnym uniwersalnym formacie typu XML lub YAML. Zawartością pliku jest tablica asocjacyjna zapisana w języku PHP, która zawiera kolejna zagnieżdżone tablice posiadające odpowiednie . Można zatem uznać, że rozwiązanie zaproponowane przez Zend Framework 2 jest hybrydą łączącą definiowanie poprzez plik i definiowanie poprzez kod.

Tablica asocjacyjna powinna zawierać klucz „router”/”routes”, pod którego wartością znajdzie się lista tras w danym module.

TODO: odwrotna koleność wyszukiwania tras

TODO: Routy w zf2

Typy tras

Zend Framework 2 dzieli trasy na typy. Typ trasy określa w jaki sposób będzie ona przetwarzana.

1. Segment

Jest to typowy rodzaj trasy. Jego działanie odpowiada zachowaniu tras w innych frameworkach MVC. Stanowi on, że trasa jest to zwykłą trasą z parametrami.

1. Literal

Dopasowanie dosłowne. Trasa zostanie wybrana wtedy i tylko wtedy, kiedy adres całkowicie w pełni będzie jej odpowiadał. Ten typ działa tak samo jak Segment bez parametrów. Został stworzony dla poprawy wydajności, gdyż do dopasowania wystarczy wykorzystać proste porównanie łańcucha znaków, bez poszukiwania i porównywania parametrów, czy walidowania ograniczeń. Jeśli dany adres jest zawsze w pełni stały, wtedy prawdopodobnie należy użyć tego typu.

1. Method

Ten typ działa jak Segment dodaje możliwość określenia metody http jaka może zostać użyta do dopasowania. Jeśli połączenie wykorzystuje metodę GET, a definicja zezwala jedynie na typ POST, wtedy trasownik uzna, że warunki nie zostały spełnione i proces poszukiwania trasy będzie trwać dalej. Ta metoda może być bardzo przydatna kiedy chcemy, aby nasza aplikacja MVC była serwisem restowym. Serwis REST używa różnych metod http do wykonywania różnych operacji. Można tak zdefiniować trasy typu Method, aby każda prowadziła do innej akcji kontrolera w zależności od metody.

1. Regex

Typ, który używa wyrażenia regularnego. Jest to dość potężne narzędzie pozwalające na szereg różnych zastosowań. Należy używać go rozsądnie, gdyż wyrażenia regularne mogą być bardzo obciążające wydajnościowo.

1. Part

Jedną z ciekawych możliwości, które daje framework jest organizowanie tras w strukturę drzewa. Jeśli trasa jest korzeniem drzewa i zostanie dopasowana wtedy trasownik przechodzi do przeszukiwania jej dzieci. Natomiast jeśli nie uda się przyporządkować trasy do adresu, wtedy cała trasa wraz z dziećmi jest pomijana. Umiejętne zorganizowanie tras na drzewa i poddrzewa daję zysk wydajnościowy oraz zwiększa czytelność kodu. Part sam w sobie nie jest rodzajem trasy, a rodzajem trasownika. Jest to spowodowane tym, że w rzeczywistości, gdy adres zostanie dopasowany to nastąpi przekierowanie dopasowania do dzieci korzenia. korzeń sam w sobie nigdy nie może być zwrócony jako wynik dopasowania.

TODO: zendr outy zorganizowane w drzewa (TreeRoute)

TODO: PODSUMOWAĆ PORÓWNANIE ROZWIĄZAŃ (ZEND MA FAJNE TYPY ROUTÓW)

Używanie tras w warstwie widoku

Warsta widoku zawiera treść kodu html, którego nieodłączną częścią są hiperłącza zwane linkami, po których kliknięciu następuje przekierowanie użytkownika do innej strony WWW. W kodzie html link jest elementem „a” z atrybutem „href” oraz z zawartości, która jest treścią wyświetlaną w przeglądarce.

TODO: Example link

Wartość atrybutu „href” jest adresem URL. Jeśli kieruje on do strony wewnętrznej, wtedy prawdopodobnie po kliknięciu połączenie zostanie przechwycone przez trasownik.

Wartość atrybuty może być zapisana wprost w postaci zwykłego łańcucha znaków [todo: dodać „jak na rysunku xxx”]. Niestety to rozwiązanie posiada oczywistą wadę. Jeśli zdecydujemy się zmienić adres na inny, będziemy musieli odnaleźć wszystkie jego wystąpienia w plikach widoku.

Funkcje generujące adres na podstawie trasy

Najpopularniejszym rozwiązaniem jest budowanie linków na podstawie trasy. Większość frameworków dostarcza funkcje pomocnicze, które na wejściu przyjmują nazwę trasy, a jako wyjście zwracają adres URL, który może być użyty bezpośrednio jako wartość atrybutu „href”.

TODO: przykład linku z urlhelperem

Dzięki takiemu rozwiązaniu widok zostaje odseparowany od fizycznych adresów. Programista musi jedynie przejmować się jaka trasa obsługuje docelową akcję, nie musi wiedzieć pod jakim adresem się ona znajduje. Można dowolnie zmieniać URL, a każdy link będzie prowadzić do poprawnej akcji kontrolera. Jeśli trasa jest sparametryzowana, wtedy funkcja dostarcza możliwość przekazania parametrów.

TODO: przykład linku z urlhelperem i parametrami

Funkcja ta potrafi walidować trasę oraz jej parametry. Jeśli trasa o podanej nazwie nie istnieje, zostanie wtedy wyrzucony wyjątek.

Screen z wyjątekime

W podobny sposób funkcja pozwala sprawdzić poprawność przekazanych parametrów. Jeśli jakiś obowiązkowy parametr został pominięty lub nie spełnia narzuconych ograniczeń wtedy również zostanie wyrzucony wyjątek z komunikatem o błędzie.f

TODO: Screen z wyjątekime

Zaletą takiej praktyki jest szybsze znajdywanie błędów w adresach URL stosowanych w widoku. Programista wie o błędzie już w chwili wejścia na stronę, zamiast dopiero po kliknięciu w wybrany link znajdujący się na niej.

Funkcje generujące pełny link na podstawie trasy

Niektóre frameworki dostarczają dodatkowe funkcje, które są w stanie zbudować cały link, nie tylko wartość atrybutu „href”. Taka funkcja otrzymuje dodatkowe parametry: obowiązkowo treść elementu „a” oraz opcjonalnie inne atrybuty wraz z wartościami.

TODO: Screen z metodą HTML.Link

Taka funkcja jest wygodna, gdy aplikacja nie stosuje skomplikowanych linków tzn. nie potrzeba przekazywać wiele argumentów elementu „a”. Jeśli natomiast zachodzi potrzeba dodawania np. kodu JavaScript w atrybutach takich jak: „onlick”, onmouseover”; wtedy może zajść potrzeba zrezygnowania z tej funkcji, aby nie zaciemniać kodu.

Funkcje generujące adres na podstawie akcji kontrolera

Taka funkcja przyjmuje nazwę akcji, do której ma docelowo kierować link. Proces polega na wyszukaniu trasy, która prowadzi do podanej akcji, a następnie wygenerowania adresu, którego używa.

TODO: Screen z urlem na podstawie akcji

Zaletą tego podejścia możliwość tworzenie linków w aplikacji zanim jeszcze ustalona zostanie struktura adresów URL. Możemy rozpocząć pracę bez zdefiniowanych tras, a wszystkie żądania będą przechwytywane przez trasy generyczne. W momencie gdy kolejne ścieżki są dodawane, zawarte w nich adresy zaczną pojawiać się w warstwie widoku. Problem pojawi się kiedy chcemy, aby dwa różne adresy kierowały do tej samej akcji kontrolera. Funkcja mając do dyspozycji akcję, która jest obsługiwana przez dwie trasy, wybierze automatycznie pierwszą z nich i wstawi do widoku URL przez nią określony. Jest to jednak nieczęsty przypadek, kiedy pod dwoma adresami pojawić się ma dokładnie ta sama strona WWW. Innym problemem może być lekki spadek wydajności. Trasy są zorganizowane wg ich nazw, a nie akcji, których dotyczą lub mogą dotyczyć. Przeszukiwanie tras w poszukiwaniu takiej, która może prowadzić do danej akcji jest trochę wolniejsze niż przejście do trasy prosto po nazwie.

Funkcje generujące pełny link na podstawie akcji kontrolera

W tym wypadku zachowanie jest połączeniem funkcji generujących adres na podstawie akcji kontrolera i generujących pełen link. Funkcja również jako parametry przyjmuje akcje, ale tworzy pełen link zamiast samego adresu URL.

Funkcje trasowania używane w widoku wybranych frameworkach

ASP.NET MVC 4.0

Framework domyślnie dostarcza obiekty pomocnicze, wśród nich są m.in. „HtmlHelper” oraz „UrlHelper”. Posiadają on metod przydatnych w warstwie widoku. Wśród nich są cztery, które odpowiadają funkcjom generującym adresy wymienionym wcześniej.

Są to:

1. UrlHelper.RouteUrl - generuje adres na podstawie trasy
2. HtmlHelper.RouteLink - generuje link na podstawie trasy
3. UrlHelper.Action - generuje adres na podstawie akcji
4. HtmlHelper.ActionLink - generuje link na podstawie akcji

Jak widać framework dostarcza wszystkie możliwe funkcje. Dziwi jednak fakt braku konsystencji nazewnictwa gdzie metoda UrlHelper.Action nie posiada sufiksu „Url”, który by sprawiał, że byłaby spójna z innymi. Stanowi to jednak mało ważny problem.

Zend Framework 2

Framework dostarcza warstwie widoku szereg obiektów pomocniczych. Wśród nich jest

„Url”. Dostarcza on jedynie możliwość tworzenie adresu na podstawie trasy. Możliwość budowania całych linków nie została dostarczona. Nie jest to jednak skomplikowana funkcjonalność, więc można ją samemu zaimplementować jeśli zajdzie taka potrzeba. Zend Framework 2 wspiera łatwe dodawanie własnych obiektów pomocniczych. Możliwość tworzenia adresów na podstawie akcji nie została uwzględniona w ogóle. Powodem może być nieefektywność takiego rozwiązania. Domyślny trasownik nie daje możliwości przeglądania skonfigurowanych tras, więc własna implementacja jest znacznie utrudniona.

Symfony 1 i Symfony 2

Oba frameworki dostarczają podstawową funkcjonalność tworzenia adresu na podstawie nazwy klasy. Symfony 1 we wszystkich wersjach dostarcza dodatkowo funkcję „link\_to”, która stworzy pełen link. Powodem usunięcia tej funkcjonalności w wersji drugiej mogła być niechęć do używania funkcji i obiektów, które tworzą kod html. Obecnie wiele IDE potrafi parsować i analizować treść html, wyświetlając przy tym napotkane błędy składni. Jeśli natomiast atrybuty elementu „a”, który jest linkiem, zostaną przekazane do funkcji frameworka, wtedy IDE nie będzie wstanie sprawdzić ich poprawności.

TODO:

Przykład w symfony 1 i symfony 2

IDE z podpowiadaniem składni html i nie znanym atrybutem

W prostych przypadkach funkcja generująca pełne linki mogłaby być przydatna , zależy to jednak od preferencji programisty.

TODO:

Porównanie długość funkcji link\_to z url\_to.

Trasowanie i dobre praktyki

Zarządzanie trasami

Używanie trasowania umożliwia łatwiejsze zarządzanie adresami w aplikacji WWW. Stosowanie generatorów adresów sprawia, że nie przejmujemy się jak wygląda sam URL. Za każdym razem kiedy chcemy go utworzyć wystarczy odwołać się do trasy po jej, która to definiuje fizyczny odnośnik. O ile niemal wszystkie frameworki zezwalają na wygodne zarządzanie adresami, to niestety pozostaje jeszcze kwestia zarządzania trasami. Nie trzeba martwić się wystąpieniami adresu, gdy go zmieniamy, ponieważ odwołujemy się wszędzie do trasy. Pozostaje pytanie co należy zrobić kiedy chcemy zmienić trasę, a przede wszystkim jej nazwę, wg której jest ona używana w widoku.

TODO:

Obrazek z trasą w widoku z podpisem:  
„Proszę zauważyć, że do wyszukania trasy używamy łańcucha znaków”

Co zrobić kiedy chcemy zmienić nazwę trasy, np. gdy ma ona zbyt ogólną nazwę?  
Przykładem może być sytuacja, gdy nasza aplikacja wyświetla profil danej osoby. Do tego celu możemy użyć trasy o nazwie „profile”. Jednak wraz z rozwojem aplikacji chcemy wyświetlać inny rodzaj profilu, który dotyczy czegoś innego np. profil firmy. Dla przejrzystości źródła, chcemy zmienić nazwę obecnej trasy z „profile”, na „user/profile”. W jaki sposób znaleźć wszystkie miejsca w kodzie, gdzie trasa byłą używana? Wyszukanie po nazwie „profile” (także wraz cudzysłowem) może zwrócić bardzo wiele wyników, z którym większa część nie ma nic wspólnego z trasą.

Innym pytanie może brzmieć: Ile razy trasa jest używana i przez jaki plik?  
Gdy wiemy, że trasa jest niepotrzebna, ponieważ strona do której prowadziła został usunięta, musimy pozbyć się linków do niej kierujących lub przekierować je na inne trasy. Ponawia się problem odnalezienia wszystkich przypadków użycia.

Najprostszym rozwiązaniem jest nieużywanie nazw wprost, ale poprzez stałe. Można stworzyć klasę lub klasy, które będę posiadały stałe zawierające nazwy. Aby zmienić samą nazwę trasy, wystarczy zmienić wartość pod stałą. Natomiast jeśli chcemy zmieniać sama stałą, wtedy narzędzia do refaktoryzacji pomogą w łatwym dokonaniu zamiany.

TODO:

Przykład pliku ze stałymi, z nazwami tras.

Oraz wykorzystanie urla z nazwą trasy w widoku.

Wymuszenie podania obowiązkowych parametrów

Trasy mogą posiadać parametry. Np. trasa „profile” musi w adresie zawierać id osoby, której profil chcemy wyświetlić.

TODO:

Trasa „profile” z wymagany id

Nie podanie tego parametru, na skutek pomyłki lub zmiany definicji trasy, będzie skutkować wyrzuceniem wyjątku podczas wejścia na daną podstronę.

Ogólnym przyjętym rozwiązaniem we frameworkach jest przekazywanie kolekcji parametrów trasy jako parametru funkcji/metody.

TODO:

Wywołanie profilu z parametrami w PHP

Podpis: W jeżyku PHP do przekazywania parametrów używana jest tablica asocjacyjna.

TODO:

Profil z parametrami parametrami w ASP.NET MVC

Podpis: C# ma Dictionary lub anonimowy obiekt

Todo :Nic nie zabezpiecza przed literówką.

Kontroler

Warstwa kontrolera (ang. Controller) jest odpowiedzialna za przetwarzanie odebranych od użytkownika poleceń. Warstwa zazwyczaj składa się wielu kontrolerów, z których każdy ma zadanie obsługiwać inne polecenia. W technologiach WWW związanych z MVC jest to zazwyczaj przychodzące żądanie wysłane przez przeglądarkę, gdy użytkownik wykona akcję typu kliknięcie w link lub wysłanie formularza. Kontroler decyduje jakie czynności należy wykonać i deleguje je do warstwy modelu, która zajmie się ich przeprowadzeniem. Następnie przekaże wybrane dane do warstwy widoku, z którym jest związany.

Zakres odpowiedzialności kontrolera w aplikacjach WWW

Kontroler jest świadomy środowiska, w ramach którego działa. Wie i może korzystać z różnych czynności i informacji związanych z naturą architektury serwisów WWW, których inne warstwy powinny być nieświadome.

Do zadań kontrolera należy zarządzanie sesją użytkownika tj.:

* rozpoczynanie i kończenie sesji
* uwierzytelnianie
* autoryzacja

Kontroler może decydować o kodzie odpowiedzi http.

Najczęściej używanymi są:

* 200 - zakończenie sukcesem
* 404 - nie znaleziono
* 403 - brak uprawnień
* 500 – wewnętrzny błąd systemu

Należy zwrócić uwagę, że wszystkie wymienione kody są automatycznie obsługiwane przez frameworki. Jest to rozwiązanie wygodne dla programisty, który nie musi się przejmować architekturą WWW. Kod 200 będzie domyślnie zwrócony, gdy nie wystąpi żaden błąd. 404 jest podawany, gdy żądany adres nie istnieje w skonfigurowanych trasach, jeszcze zanim zostanie uruchomiony kontroler. 403 wystąpi, gdy użytkownik nie jest zalogowany lub nie ma uprawnień do danej strony, lub zasobu. Kod 500 zostanie przekazany, gdy zostanie wyrzucony i nieobsłużony wyjątek.

Framework nie zawsze automatycznie może rozpoznać jaki kod powinien być użyty. Przykładem może być nie istniejący zasób danych w bazie danych. O ile adres URL jest poprawny to brak zasobu, który jest wymagany do obsłużenia adresu może być wystarczającym powodem, aby zwrócić kod 404. Musi to być niestety manualnie wykonane przez programistę, jednak frameworki starają się to ułatwić.

TODO: przykład wywołania 404

Do innych obowiązków kontrolera należy sterowanie przekierowaniami. Przykładem może być zalogowanie, po którym chcemy wyświetlić inną stronę, w zależności od wykonywanej roli.

TODO:

Przykład przekierowania

TODO: może napisać coś jak kontrolerowi wybiera się widok i jak sam może decydować który widok wyświetlić

Implementacja warstwy kontrolera

Typowa implementacja warstwy we frameworkach MVC zakłada istnienie wydzielonych klas, których metody będą przetwarzać poszczególne żądania użytkownika. Klasy te nazywa się po prostu kontrolerami. Niektóre frameworki (ASP.NET MVC, Zend Framework 2, Symfony 2) stosują konwencję nazewnictwa, wg której nazwę klasy kontrolera kończy się słowem „Controller” np. „ProfileController”. Celem tego jest jasne zaznaczenie przeznaczenia klasy. Metody kontrolerów nazywane są akcjami. Tutaj różne frameworki mają inne wymogi nazywania.

Stosowane nazewnictwa:

* Symfony 1 – metody kończą się prefiksem „execute”
* Zend Framework 2, Symfony 2 – metody kończą się prefiksem „Action”
* ASP.NET MVC – dowolna nazwa

Raz jeszcze poprzez konwencję twórcy starają się wyróżniać elementy o specjalnym przeznaczeniu. W tym wypadku oddzielają metody wyznaczone do przetwarzania akcji użytkownika od zwykłych metod.

TODO: Przykład kontrolera

Jeśli jakieś dane pochodzące z modelu powinny być wyświetlone użytkownikowi, wtedy obowiązkiem kontrolera jest ich przekazanie do warstwy widoku.

Todo: o przekazywaniu do widoku

Porównanie rozwiązań

Obsługa parametrów przychodzących

Akcja kontrolera wykonuje polecenie użytkownika. Przykładowo metoda „deleteAction” kontrolera „profile” ma za zadanie usunięcie profilu. Ten sam kontroler może posiadać inne akcje takie jak „editAction” czy „addAction”. Samo wywołanie akcji świadczy o intencjach użytkownika, ale na ogół potrzeba jeszcze dodatkowych informacji. W tym celu istnieje możliwość przesyłania dodatkowych parametrów z przeglądarki. Istnieje kilka sposobów na przesyłanie takich danych. W technologii stron WWW do podstawowych należą:

* GET – dane przekazywane w adresie
* POST –dane pochodzące z formularza

Przykładem może edycja profilu. Dane z formularza są przekazywane jako POST. W adresie znajduje się parametr „profile\_id” wraz z wartością – jest parametry typu GET.

TODO: przykładowy adres: (wyrzucić opis z ramki?)

/profile/edit?profile\_id=7

Pod takim adresem wysyłany jest formularz z danymi do edycji. Wartość 7 parametru „profile\_id” stanowi, który profil edytujemy. Będzie ona dostępna jako parametr GET.

Frameworki wzbogacają dodatkowo te parametry o ROUTE. Są dane zawarte w samej ścieżce adresu, a nie jako dodatkowe atrybuty na jego końcu.

TODO: przykładowy adres z parametrem w Roucie

/profile/7/edit

Taki adres wygląda bardziej estetycznie. Prowadzi do takiej samej akcji jak adres powyżej, jednak tym razem w parametrach GET nie będzie się nic znajdować.

Zend Framework 2 oferuje kontrolery, którym wstrzyknięto tablice zawierające wszystkie parametry.

Ich zawartość dostęp na jest poprzez następujące metody:

|  |  |
| --- | --- |
| $this->params()->fromPost('paramname'); | Pobiera dane z POST |
| $this->params()->fromQuery('paramname'); | Pobiera dane z GET |
| $this->params()->fromRoute('paramname'); | Pobiera parametry z trasy |

Zaletą używania tych metod jest odseparowanie się od leżącej pod spodem architektury. Pomaga to w łatwiejszym testowaniu jednostkowym kontrolerów.

ASP.NET MVC oferuje podobny dostęp do danych w kontrolerze:

|  |  |
| --- | --- |
| Request.Form | Pobiera dane z POST |
| Request.QueryString | Pobiera dane z GET |
| RouteData.Values | Pobiera parametry z trasy |

Nie jest to jedyny sposób w jaki framework oferuje dostęp do parametrów.

Rozpatrzmy przykładowy szablon trasy:

TODO:

samochody/{category}/{page}

Widoczne są dwa parametry reprezentujące kategorią i numer wyświetlonej podstrony. Adres pasujący do szablony to np.:

TODO:

samochody/terenowe/2

Akcja pobierająca wartości parametrów może wyglądać następująco:

TODO:

public ActionResult Index()

{

string category = RouteData.Values["category"];

int page = RouteData.Values["page"];

return View();

}

Używając tego podejścia musimy w każdej akcji dodawać linijki kodu pobierające parametry z kontenera RouteData.Values. Nie jest to szczególny problem, ale powiększa to rozmiar kodu, który nie wykonuje żadnych istotnych operacji. Pobieranie parametrów z trasy jest dość częstym zabiegiem, dlatego framework dostarcza inny sposób. Parametry trasy mogę być przekazywane jako parametry metody. Oto przykładowa akcja, która wykonuje to samo, ale drugim sposobem:

TODO:

public ActionResult Index(string category, int page)

{

return View();

}

Zaletą tego podejścia jest nie zaśmiecanie ciała metody pobieraniem parametrów z trasy. Również same parametry w definicji metody pokazują co jest oczekiwane od trasy.

System dopasowujący parametry opiera się na mechanizmie refleksji dostarczanym przez język C#. Dzięki czemu framework jest w stanie odczytać nazwy parametrów i jako ich wartości przekazać dane pochodzące z trasy. Dzięki temu możliwa jest również zamiana kolejności parametrów. Jeśli jakiś parametr jest opcjonalny, wtedy framework po raz kolejny wykorzystuje możliwości języka C# i pozwala na użycie typów dopuszczających wartość zerową (ang. Nullable types). Zakładając, że wartość „page” może być pominięta w adresie akcja wyglądałaby następująco:

TODO:

public ActionResult Index(string category, int? page)

{

return View();

}

Proszę zauważyć znak zapytania przy typie parametru page. Mówi on, że zmienna może być pusta i nie znajduje się pod nią żadna wartość.

PHP domyślnie nie dostarcza funkcjonalności typów dopuszczających wartość zerową, jednak nie są one trudne w implementacji. Od wersji 5 język dostarcza również mechanizmy refleksji, które pozwalają pobierać nazwy parametrów metody. Jak widać PHP spełnia wymagania potrzebne, aby napisany w nim framework mógł zastosować przekazywanie parametrów trasy wprost do metody. W Zend Framework 2 niestety taka możliwość nie została zaimplementowana.

Odwrócenie sterowania

Todo: opisać oddzielnie odwrócenie sterowania i wstrzykiwanie zależności

Odwrócenie sterowania jest rozwiązaniem projektowym, które pozwala na oddzielenie poszczególnych elementów kodu w taki sposób, aby obiekty nadrzędne nie zajmowały się same tworzeniem elementów podrzędnych, których używają. Przykładem może być klasa będąca serwisem (element nadrzędny), który używa repozytorium (element podrzędny).

Zend Framework 2

Framework posiada wbudowany mechanizm odwróconego sterowania oparty o wzorzec lokalizatora serwisu (ang. Service Locator). Każdy moduł aplikacji posiada plik „Module.php”. W nim m.in. znajduje się konfiguracja wszystkich serwisów, których chcemy używać. Definiowanie polega na utworzeniu tablicy asocjacyjnej, której kluczami są nazwy serwisów, a wartościami domknięcia, które zwracają serwis. Domknięcie zajmuje się tworzeniem instancji serwisu. lokalizatora serwisu. Przykładem może być sytuacja, gdy jeden serwis korzysta z drugiego i domknięcie musi je wstrzykiwać. Istnieje pełna możliwość zarządzania instancjonowaniem serwisów. W zależności od wybranej konfiguracji lokalizator będąc poproszony o serwis może zawsze zwracać jedną, tą samą Może do tego celu używać instancję lub za każdym razem tworzyć nową. Oprócz udzielaniem dostępu do serwisów napisanych przez twórców aplikacji lokalizator wyszukuje również gotowe serwisy wbudowane we framework.

Podczas tworzenia aplikacji lokalizator serwisów jest dostępny domyślnie na poziomie warstwy kontrolera. Każdy kontroler z Zend Framework 2 posiada metodę udzielającej dostępu do lokalizatora, który umożliwia używanie serwisów.

Todo:

Przykład Module.php z dwoma serwisami, jeden wstrzykuje sobie drugi, pobierając go z $sm.

Todo:

Przykład wyciągnięcia serwisu poprzez serwis locatora w kontrolerze.

Lokalizator serwisu pomaga w poruszaniu się w modułowej strukturze projektów tworzonych w Zend Framework 2. Język PHP od wersji 5.3.0 posiada wsparcie dla przestrzeni nazw. Niestety ten mechanizm wciąż posiada wiele wad. Przenoszenie modułów w ramach projektu może łatwo spowodować błąd aplikacji spowodowany niemożliwością odnalezienia klas, które dotychczas były używane bez przeszkód. Lokalizator serwisu potrafi samodzielnie odnajdywać wymagane serwisy. Dzięki temu sprawia, że kod źródłowy staje się niezależny od fizycznej struktury modułów i katalogów w projekcie.

Istnienie wbudowanego mechanizmu może wydawać się narzuceniem rozwiązania. Jednak, jeśli programista jest w stanie sobie poradzić z ładowaniem klas, nie musi w cale z tego rozwiązania korzystać. Realizacja wzorca jest dość elastyczna i powinna spełniać wymagania nawet dla dużych projektów. Istnienie lokalizatora serwisu jest zaletą frameworka, szczególnie dla zespołów, które poszukują gotowej implementacji dla zastosowania odwrócenia zależności.

[Todo: stosowanie fabryk controlerów – ważne jako kontrast jak to się robi w ASP.NET MVC (tu są fabryki kontrolerów, tam trzeba pisać własne)]

TODO kod z implemetnacją fabryki w postaci klasy i postaci domknięcia

Todo: dobre praktyki, wydzielenie nazw serwisów do constów.

- Tworzenie metod hintujących jaki typ serwisu jest zwracany.

- Stworzenie silnie typowanego wrapera na serwis lokatora

- nie wstrzykiwanie serwis lokatora do obiektów, używanie SL tylko w kontrolerach

ASP.NET MVC 4

Framework nie dostarcza żadnego gotowego rozwiązania pozwalającego do zastosowania odwrócenia sterowania. Istnieje oczywiście możliwość wprowadzenia gotowej implementacji jednak nie będzie ona domyślnie zintegrowana z frameworkiem.

W mniejszych projektach nie jest potrzebny pełen system zarządzanie zależnościami typu kontenery Spring czy Unity. Jednak ze względów przejrzystości kodu mechanizmy takie jak wstrzykiwanie zależności są zawsze zalecane, szczególnie jeśli ilość kodu może się potencjalnie zwiększyć. Tu dodatkowy problem jaki stwarza ASP.NET MVC 4 to również brak wsparcia przy tworzeniu kontrolerów. Aby użyć wstrzykiwania przez konstruktor programista musi dostęp nad tworzeniem obiektu. Zend Framework umożliwiał tworzenie własnych fabryk w postaci domknięcia lub oddzielnej klasy. Fabryki są organizowane wg nazw kontrolera. Trasa wybrana przez trasownik posiada zapisaną nazwę kontrolera, która później jest użyta do wyboru fabryki. W ASP.NET MVC 4 nie istnieje gotowe rozwiązanie tego typu, ale istnieje możliwość stworzenia własnej fabryki kontrolerów. Aby stworzyć fabrykę w postaci klasy należy zaimplementować interfejs IControllerFactory. Interfejs posiada trzy metody, ale tylko jedna zajmuje się tworzeniem kontrolerów, dlatego łatwiejszym sposobem jest stworzeniem klasy poprzez dziedziczenie po klasie DefaultControllerFactory i nadpisaniu metody CreateController. Metoda jako parametr dostaje nazwę kontrolera dopasowanego podczas trasowania oraz szczegóły wykonanego żądania w postaci klasy RequestContext. Takie informacje są wystarczające, aby móc tworzyć własne kontrolery, jednak to programiście pozostaje zorganizowanie budowy kontrolerów i ich konfigurowania.

<http://www.dotnetcurry.com/showarticle.aspx?ID=878>

Todo: kod z implementwacją fabryki kontrolera

Formularze

Formularze umożliwiają użytkownikom na wprowadzenie danych i wysyłanie ich do serwera, gdzie zostają przetworzone. Są tworzone na wzór prawdziwych formularzy, zawierają pola, w które można wprowadzać tekst, zaznaczać i wybierać różne opcje. Są nieodłączną częścią stron internetowych i nie istnieje wiele witryn, które by z nich nie korzystały. Typowa interakcja z formularzem wymaga operacji przechodzących przez wszystkie warstwy aplikacji. Przykładem może być formularz edycji profilu. Najpierw bieżące dane muszą zostać pobrane z bazy danych, następnie na ich podstawie tworzony jest model (często przy użyciu ORM), który kolejno zostaje przekazany przez kontroler do widoku gdzie na jego podstawie zostają uzupełnione pola formularza. W ten sposób użytkownikowi zostaje przedstawiony formularz, który jest wstępnie wypełniony starymi danymi. Kiedy użytkownik poprawi informację i wyśle formularz, wtedy wszystkie warstwy zostaną odwiedzone ponownie. Dodatkowo wykonana zostanie walidacja danych, która może być traktowana jako oddzielna. W przypadku, gdy dane są poprawne nowy stan zostanie zapisany, jeśli natomiast walidacja zwróci błąd, wtedy użytkownik zostanie o tym poinformowany i poproszony o poprawę wpisanych informacji. Sama walidacja może zostać wykonana kilkukrotnie w oddzielnych warstwach. Na przykład walidacja w warstwie widoku umożliwia wykrywanie błędów bez wykonywania połączenia z serwerem, co zmniejsza obciążenia serwera i jest przyjaźniejsze użytkownikowi, gdyż ten otrzymuje informację o błędach natychmiast.

Często podawanie informacji po raz pierwszy wymaga bardzo podobnego formularza jak ich późniejsza edycja. Jako, że walidacja po stronie klienta nie jest rzetelna naturalnym jest obowiązkowa walidacja po stronie serwera. Jak widać obsługa nawet prostych formularzy może być pracochłonna ponieważ jest wiele czynników, które należy wziąć pod uwagę. Strony internetowe posiadają na ogół wiele formularzy, więc

nie dziwi fakt, że frameworki MVC próbują dostarczyć rozwiązań, które przyspieszyłyby proces ich tworzenia.

Zend Framework 2

Formularz w tym frameworku jest traktowany jako obiekt klasy Zend\_Form. Do niego możemy dodawać kolejne pola formularza. Pola są reprezentowane poprzez obiekty klasy Zend\_Form\_Element. Jest kilka klas, które po niej dziedziczą i reprezentują inne rodzaje pól:

|  |  |
| --- | --- |
| Typ pola | Opis |
| text | Standardowe pole tekstowe. |
| textarea | Jak pole tekstowe, ale przeznaczone na większą ilość tekstu. |
| password | Jak pole tekstowe, ale wpisywany tekst nie jest widoczny. |
| radio | Pole jednokrotnego wyboru. |
| checkbox | Pole wyboru tak/nie. |
| select | Rozwijana lista, z której można wybrać jedną pozycję. |
| multiselect | Rozwinięta lista, z której można wybrać wiele pozycji. |
| file | Umożliwia wysłanie pliku. |

Każde z powyższych pól posiada własną klasę, która je reprezentuje i może zostać dodane to formularza.

Todo: kod dodawanie pola do formularza

[http://framework.zend.com/manual/2.0/en/user-guide/forms-and-actions.html#](http://framework.zend.com/manual/2.0/en/user-guide/forms-and-actions.html)

Oprócz pól pobierających dane od użytkownika formularz może posiadać inne elementy takie jak przycisk wysyłania formularza czy ukryte pola, w których programista może zapisywać dodatkowe informacje. One również są reprezentowane poprzez klasy dziedziczące po Zend\_Form\_Element.

Zazwyczaj każde pole posiada opis będący informacją jakie dane są oczekiwane. Jest nazywany etykietą (z ang. label). Przykładem może być „Imię”, „Numer telefonu”. Zend Framework umożliwia dodawanie ich do elementów.

Todo: dodawanie labela do pola

Jeśli pole wymaga wpisania ciągu znaków to może wymagać sprawdzenia poprawności. Przykładem może być numer telefonu, który powinien składać się z cyfr lub adres e-mail wymagający określonego formatu. Tu framework również pozwala na dodawanie tzw. walidatorów (z ang. validator). Istnieje kilka standardowych walidatorów umożliwiających nałożenie pewnych restrykcji na wpisywane wartości. Przykładem może być walidator Between sprawdzający czy wpisana wartość jest liczbą znajdującą się w zadanym przedziale.

Todo: walidator between

<http://framework.zend.com/manual/2.0/en/modules/zend.validator.set.html>

Dodanie walidator

Oczywiście istnieje możliwość stworzenia własnego walidatora jeśli zajdzie taka potrzeba.

Todo: customowy walidator

<http://framework.zend.com/manual/2.0/en/modules/zend.validator.set.html>

Pole formularza nie zawsze musi być wymagane. Opcjonalne pola oznacza się przy użyciu metody setRequired.

Todo: przykład z setRequired(false)

<http://framework.zend.com/manual/2.0/en/modules/zend.validator.set.html>

Czasami warunek walidacji wymaga sprawdzenia kilku pól. Na przykład potwierdzenie adresu e-mail składa się z dwóch pól i potrzebne jest sprawdzenie obu z nich. W tym celu można nadpisać metodą isValid formularza, gdzie można dodać takie sprawdzenie.

Todo: nadpisanie isValid w formualrzu.

<http://framework.zend.com/manual/2.0/en/modules/zend.validator.set.html>

Bardziej rozbudowany formularz, może wymagać walidacji warunkowej – walidacja danego pola/pól może być pominięta w określonej sytuacji. Przykładowo formularz może prosić o podanie numeru dowodu osobistego. Jeśli jednak może być on wypełniony przez osoby niepełnoletnie wtedy dodatkowo udostępnione zostanie pole typu „checkbox” gdzie osoba ten fakt może zaznaczyć. W takiej sytuacji sprawdzenie poprawności numeru dowodu nie odbędzie. Aby dokonać walidacji warunkowej, jak i w poprzednim wypadku, należy posłużyć się metodą isValid. W niej należy sprawdzić dane warunki, a następnie w zależności od ich spełnienia, dodać „w locie” walidatory.

Todo: walidacja warunkowa z walidatorem w locie.

<http://stackoverflow.com/questions/1431924/zend-form-element-should-only-be-required-if-a-checkbox-is-checked>

Todo: napisać więcej o formularzach w Zend2 (m.in. w widoku)

ASP.NET MVC 5

Obsługa formularzy w ASP.NET MVC 5 używa odmiennej mechaniki niż inne framework. Nie tworzy się tu oddzielnej klasy, która bezpośrednio przedstawia formularz. Zamiast tego formularz jest częścią modelu widoku. Model widoku jest obiektem klasy i zawiera informacje przekazywane pomiędzy kontrolerem, a widokiem. Jeśli pole formularza jest napisem, to pole w modelu będzie łańcuchem znaków, jeśli jest liczbą to analogicznie pole widoku będzie również liczbą.

Todo:

Przykład modelu widoku.

Jeśli dana strona posiada formularz to model widoku będzie zawierał jego własności. Model będzie przekazywany z kontrolera do widoku i przetłumaczony na formularz. Następnie użytkownik uzupełnieni pola formularze i po zakończeniu pracy wyśle go do serwera Żądanie trafia do kontrolera, gdzie dane formularza zostają użyte do zbudowania ponownie modelu widoku, tym razem jednak z danymi wpisanymi przez użytkownika. Całość została zaprojektowana w taki sposób, aby programista jak najmniej przejmował się obsługą żądania – nie musi w ogóle przejmować pobieraniem danych przekazanych metodą POST, gdyż framework przepisze je automatycznie do modelu. Twórcy postarali się również, aby walidacja odbywała się automatycznie, odciążając kontroler i sprawiając, że będzie on bardziej przejrzysty.

W przeciwieństwie do Zend Framework 2 rozwiązanie Microsoftu pozwala użyć dwóch akcji kontrolera do obsługi jednego formularza. Pierwsza używa metody GET i zajmuje się wyświetlaniem formularza, druga stosuje metodę POST i jest odpowiedzialna za odebranie i przetworzenie przesłanych przez użytkownika informacji. Jest to możliwe dzięki mechanizmowi przeciążania metod w języku C#, który pozwala na stworzenie dwóch metod o tej samej nazwie (ale innej sygnaturze).

Todo: dwie kacje kontrolera dla jednego formularza

Akcja używająca metody GET jest używana jako pierwsza. Tworzy ona model widoku, który będzie wypełniony przez użytkownika poprzez formularz. Model może być pusty tzn. nie zawierać będzie żadnych początkowych wartości, przez formularz również będzie pusty. Dzieje się tak zazwyczaj kiedy użytkownik ma stworzyć nowy obiekt.

Todo: tworzenie pustego formularza

Podpis: stworzenie pustego modelu widoku jest po prostu storzeniem nowego obiketu

Możliwe jest również ustawienie wartości początkowych dla formularza, poprzez przypisanie wartości do odpowiednich pól modelu.

Todo: tworzenie NIEpustego formularza

Podpis: ustawienie wartości domyślnych wymaga jedynie prostego

Przygotowany model zostaje przekazany do widoku, gdzie będzie wyświetlony. Podobnie jak w Zend Framework 2 dokonuje się tego przez zwracanie jako wynik metody.

Todo: tworzenie NIEpustego formularza

Return View(viewModel);

Widok następnie używa modelu do tworzenia formularza w kodzie HTML.

Todo: tworzenie formularza w widoku

opis kod markup(tłumaczeni markup znaleźć)

Todo: screenshot ze stworzonym formularzem

Kiedy użytkownik wypełni formularz i go wyśle, uruchomiana zostanie tym razem metoda kontrolera używająca metody POST. Metoda ta przyjmuje ten sam model widoku jako parametr i może użyć jego własności i go przetworzyć, np. zapisując wprowadzone dane do bazy danych.

Todo: OPISAĆ że metoda post możę odbierać różne parametry

FormCollection collection

Każde pole to inny parametr

Po uzyskaniu danych z formularza należy sprawdzić ich poprawność. Jeśli informacje są zgodne z restrykcjami na nie nałożonymi wtedy można je przetworzyć. W przeciwnym wypadku należy ponownie wyświetlić formularz, zwracając błędy w nim zawarte, tak by użytkownik mógł je poprawić. Do sprawdzenia zgodności framework używa obiektu ModelState, który posiada własność IsValid. Zawiera ona wartość prawda/fałsz mówiącą o poprawności danych.

Todo: ModelState.IsValid

ASP.NET MVC 5 posiada unikalny mechanizm walidacji oparty o atrybuty (adnotacje). Korzystając z możliwości języka C# twórcy starali się odciążyć logikę kontrolerów poprzez umieszczenie reguł walidacji bezpośrednio w modelu widoku. Każde pole formularza jest reprezentowane jako pole modelu widoku. Do tego pola możemy dodawać atrybuty, które będą nakładały restrykcje nałożone na to pole. Taki atrybut musi dziedziczyć po klasie ValidationAttribute.

Todo: Przykład viewmodelu z atrybutami

Framework dostarcza gotowe walidatory. Przykłady najpopularniejszych:

* RequiredAttribute – stanowi, że dane pole jest wymagane
* MaxLengthAttribute – przyjmuje jako parametr liczbę całkowitą, która mówi jak wiele znaków może posiadać dane pole
* MinLengthAttribute – działa analogicznie jak powyższy atrybut, ale dotyczy minimalnej liczby znaków
* EmailAddressAttribute – sprawdza czy podana wartość jest adresem email
* PhoneAttribute – sprawdza czy podana wartość jest poprawnie sformatowanym numerem telefonu

Istnieje oczywiście możliwość stworzenia własnych walidatorów.

Todo: Przykład viewmodelu z atrybutami

Walidotor jest odpowiedzialny za ustalenie treści błędu. Domyślnie wszystkie wiadomości są w języku angielskim, ale konstruktor może przyjąć jako parametr treść błędu. Jest to zachowanie typowe dla wszystkich dostarczonych walidatorów.

Todo: Przykład walidatora z podaną treścią błedu

Warto zaznaczyć, że nie istnieją walidatory sprawdzające czy podana wartość jest odpowiedniego typu. Na przykład nie istnieje walidator sprawdzający czy podany ciąg znaków jest liczbą całkowitą. Nie jest to wymagane ponieważ model widoku zna typ pola i może sam wywnioskować czy podana wartość jest poprawnego typu.

Mankamentem niestety jest brak możliwości kontrolowania takiej walidacji. Prowadzi to do braku opcji wyboru błędu, który ma się wyświetlić użytkownikowi. Jednym obejściem tego problemu jest wykorzystanie innego walidatora RangeValidator. Umożliwia on wybranie zakresu wartości liczby całkowitej. Jako parametry przyjmuje możliwą najmniejszą i największą wartość danego pola. Jeśli pole nie będzie liczbą całkowitą walidator zwraca błąd. Jednocześnie ustawienie parametrów na odpowiednio maksymalnie niską i maksymalnie dużą wartość, jaką można przypisać liczbie całkowitej, sprawi że walidator będzie zwracał błąd tylko kiedy dana nie będzie liczbą całkowitą. Należy jedynie zmienić domyślą treść błędu z sugerującej, że liczba nie zawiera się w zakresie, na mówiącą, że dana wartość nie jest w ogóle liczbą. Inny, bardziej poprawny sposób, aczkolwiek bardziej pracochłonny, wymaga stworzenia własnego walidatora. Sprawdzałby on po prostu czy podana wartość składa się tylko i wyłącznie z cyfr oraz ewentualnie znaku ujemnego (dodatkowo może wystąpić przecinek, jeśli chcemy dopuścić liczby zmiennoprzecinkowe) . Istnieją gotowe biblioteki stworzone przez osoby trzecie typu „Data Annotations Extensions”, dostarczają one m.in. taki walidator.

Innym brakującym walidatorem jest brak maksymalnej wartości liczby całkowitej oraz minimalnej. Istnieje wspomniany wcześniej RangeValidator, ale zwraca on zawsze tą samą treść błędu. Programista często chciałby podać dokładnie czy dana wartość była za duża, lub czy była za mała. Niestety jeśli zajdzie taka potrzeba musi on się uciec do stworzenia własnego walidatora lub skorzystania z gotowych rozwiązań z zewnątrz.

Walidacja poprzez adnotacje pozwala na łatwe sprawdzanie pojedynczych pól – do każdego pola doczepiamy jeden lub więcej walidatorów. Każdy walidator sprawdza wyłącznie swoje pole. Problem zaczyna się, gdy reguła walidacji wymaga sprawdzenia więcej niż jednego pola. Przykładem może być sytuacja kiedy dla pewności prosimy użytkownika, aby dwukrotnie podał adres email. Musimy wtedy sprawdzić czy obie podane wartości są identyczne. Inny przykład to formularz, który umożliwia wpisanie liczb całkowitych, których suma nie powinna przekroczyć określonej wartości. W takim wypadku walidator musi przetworzyć jeszcze większą ilość pól.

Todo: zapychacz – obrazek z widokiem z formularza z ilomaś tam inputami na wartości, które w sumie nie mogą przekroczyć jakijs tam sumy

Todo: zapychacz – obrazek z widokiem formularza z mailami

Walidacja globalna (taka, która wymaga sprawdzenie większej ilości pól) w ASP.NET MVC 5 może być nadal dokonana poprzez adnotacje. Każdy walidator jako parametr otrzymuje „kontekst walidacji”(obiekt klasy ValidationContext). Zawiera on informacje o m.in. wszystkich wartościach jakie były przekazane do formularza, w tym z pozostałych pól. W ten sposób walidator uzyskuje dostęp do większej ilości informacji, które są mu potrzebne.

Todo: walidator wykorzystujący kontekst do walidacji.

Ważnym aspektem tego podejścia jest to, że walidacja uzna wystąpienie błędu jako błąd należący do pola, do którego został przypisany walidator. Skutkuje to wyświetleniem komunikatu zaraz obok pola, przez co użytkownik powiąże błąd z tą właśnie wartością. Do programisty należy obowiązek zadbania, aby komunikat błędu występującego na podstawie wartości z kilku pól, a wyświetlany tylko przy jednym z nich, był intuicyjny dla odbiorcy.

Todo: błąd występujący gdy email się nie zgadzają

Podpis: W przypadku, gdy dwie wartości adresu email są różne możemy przyjąć, że to druga wartość jest błędna ponieważ jest ona potwierdzeniem pierwszej, dlatego komunikat zostanie wyświetlony przy niej.

ASP.NET MVC 5 posiada jeszcze inny sposób walidowania danych. Jeśli model widoku będzie implementował interfejs IValidatableObject to będzie mógł on sam przeprowadzić kompletną walidację. Interfejs wymaga implementacji metody Validate, w której przeprowadza się sprawdzenie poprawności. Metoda jako parametr przyjmuje kontekst walidacji(ten sam co w przypadku walidacji przy użyciu adnotacji), a jako wynik zwraca listę znalezionych błędów. Każdy błąd jest obiektem klasy ValidationResult. Zawiera on treść błędu oraz listę pól których dotyczy(ustala to programista). Jeśli lista pól, których dotyczy błąd, jest pusta, wtedy jest on uznany za błąd globalny t.j. obejmuje cały formularz, a nie konkretne pole.

W zależności od potrzeb formularza można połączyć oba podejścia – jednocześnie zastosować walidację przez adnotację i w tym samym czasie zastosować walidację całości poprzez implementację interfejsu IValidatableObject. Podejście to powoduje jednak, że walidacja znajduje się w dwóch miejscach naraz co może powodować następujące problemu:

* Framework najpierw sprawdzi reguły ustalone przez adnotację, jeśli jakieś zostaną naruszone to znalezione błędy zostaną zwrócone natychmiast, a walidacji całościowa(globalna) się nie odbędzie. Powoduje, że użytkownikowi może wyświetlić się błąd dotyczący jakiegoś pola, po jego poprawieniu wyświetlić się inne błędy, które mogły być podane wcześniej. Użytkownik jest zmuszony dwukrotnie poprawiać formularz. Istnieje możliwość manualnego uruchomienia walidacji całościowej w takiej sytuacji, ale wymaga to dodania kodu, który zaciemnia kod aplikacji.

Zapobiec temu można poprzez użycie tylko walidacji z IValidatableObject, ale może to wymagać manualnego walidowania pól, co będzie niewygodne dla większych formularzy.  
<http://www.haneycodes.net/trigger-ivalidatableobject-validate-when-modelstate-isvalid-is-false/> todo: wkleić kod z wpowyższego linku i dodac do bibliografii

* Programista, który nie tworzył walidacji formularza, może mieć problem ze znalezieniem miejsca, które spowodowało zwrócenie błędu. Jeśli zauważy, że model widoku używa adnotacji, to będzie właśnie tam będzie poszukiwał reguły, która może znajdować się zupełnie gdzie indziej.

Todo: ??? maybe not do - Javascript i dataannotations

Todo: Kontroler przekazuje ViewModel.

Kontroler odbiera ViewModel w metodzie POST.

[http://www.asp.net/mvc/overview/older-ve](http://www.asp.net/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-5)

[rsions/mvc-music-store/mvc-music-store-part-5](http://www.asp.net/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-5)

http://www.asp.net/mvc/overview/older-versions/working-with-the-dropdownlist-box-and-jquery/using-the-dropdownlist-helper-with-aspnet-mvc

Todo: przykład z definiowaniem typu pola formularza w kontrolerze  
i przykład z typem wybranym w widoku.  
  
<http://msdn.microsoft.com/en-us/library/dd410596%28v=vs.100%29.aspx>

W Zend 2 obiekt formularza wiedział jakiego typu pola są używane w warstwie widoku. Jeśli miała być widoczna lista rozwijana, to było to deklarowane podczas konfiguracji w kontrolorze. W ASP.NET MVC 5 jest to opcjonalne – to czy dana kolekcja elementów będzie wyświetlona jako lista rozwijana czy pole typu „radiobox” może zostać określone w widoku, bez wiedzy kontrolera. Podobnie jest z innymi typami pól. Istnieje również możliwość sprecyzowania typu jeszcze przed wyświetleniem widoku.

<http://stackoverflow.com/questions/16100300/asp-net-mvc-custom-validation-by-dataannotation>

<http://stackoverflow.com/questions/2417113/asp-net-mvc-conditional-validation>

http://symfony.com/doc/current/book/validation.html

Symfony 2

Todo: Formularze symfony 2 podobne są do zend2, ale mają integrację z encjami doctrine2 (choć wymaga to seterów których nie lubimy)

Sym

Model

Widok

Todo: pojechać po używaniu bezpośrednio zmiennych, bo kłócą się helperami

Todo: nie można odpalić akcji z poziomu widoku w ZF2 (action partiale)

ViewModel

PostDispatch, Predispatch

TODO: opisać jak model w ujęciu ORM bywa mylony z modelem MVC

<http://michalorman.pl/blog/2010/03/model-widok-kontroler/>

Widok

Słownik:

Hacker

MVC

Model

Warstwa Widok anzywa w skrócie po prostu widokiem

Kontroler

Framework

Żądanie – Request

Paginacja

Tablica asocjacyjna

Metoda HTTP

Tag html

IDE

Adres URL

refaktoryzacji

Service Locator/ lokalizator serwisów

Domknięcie

Walidator

przestrzeni nazw

Adnotacja

Sygnatura metody

Bilbiografia

<http://www.asp.net/mvc/tutorials/controllers-and-routing/creating-a-route-constraint-cs>

<http://www.altcontroldelete.pl/artykuly/tworzenie-regul-routingu-w-asp-net-mvc-3/>

<http://dataannotationsextensions.org/>