TWÓRZ ROZBUDOWANE APLIKACJE WEBOWE!

Apress

ASP.NET MVC 3 Framework
Zaawansowane programowanie

Steven Sanderson, Adam Freeman

Helion

Tytuł oryginału: Pro ASP.NET MVC 3 Framework

Tłumaczenie: Paweł Gonera

ISBN: 978-83-246-4822-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/aspmv3.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/aspmv3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 18

Kontrola poprawności modelu

W poprzednim rozdziale przedstawiliśmy, w jaki sposób na podstawie żądań HTTP biblioteka MVC tworzy obiekty modelu w procesie dołączania modelu. Zakładaliśmy, że dane wprowadzone przez użytkownika były prawidłowe. W rzeczywistości użytkownicy często wprowadzają dane, z których nie możemy skorzystać, i tym zajmiemy się teraz — *kontrolą poprawności danych modelu*.

Kontrola poprawności modelu jest procesem, dzięki któremu upewniamy się, że otrzymane dane nadają się do użycia w modelu, a jeżeli nie, dostarczamy użytkownikom informacje pomagające rozwiązać problem.

Pierwsza część procesu — sprawdzanie otrzymanych danych — jest jednym ze sposobów zapewnienia integralności danych w modelu domeny. Przez odrzucenie danych, które nie mają sensu w kontekście naszej domeny, zapobiegamy powstawaniu dziwnych i niechcianych stanów aplikacji. Druga część — pomoc użytkownikowi w rozwiązaniu problemu — jest równie ważna. Jeżeli nie zapewnimy użytkownikowi informacji i narzędzi potrzebnych do interakcji z aplikacją w oczekiwany przez nas sposób, szybko spowodujemy jego frustrację. W przypadku aplikacji dostępnych publicznie często powoduje to wycofywanie się użytkowników, natomiast w przypadku aplikacji korporacyjnych może to skutkować spowolnieniem ich działań. Każda z tych sytuacji jest niepożądana.

Na szczęście biblioteka MVC zapewnia rozbudowaną obsługę kontroli poprawności modelu. Pokażemy teraz, w jaki sposób korzystać z podstawowych funkcji, a następnie przejdziemy do bardziej zaawansowanych technik sterowania procesem kontroli poprawności.

Tworzenie projektu

Potrzebna jest nam prosta aplikacja MVC, w której będziemy mogli stosować różne techniki kontroli poprawności modelu. Na jej potrzeby utworzyliśmy klasę modelu widoku o nazwie `Appointment`, zamieszczoną na listingu 18.1.

Listing 18.1. *Klasa modelu widoku*

```
using System;
using System.ComponentModel.DataAnnotations;

namespace MvcApp.Models {
    public class Appointment {

        public string ClientName { get; set; }
        [DataType(DataType.Date)]
        public DateTime Date {get; set;}
```

```
        public bool TermsAccepted { get; set; }
    }
}
```

Na listingu 18.2 zamieszczony jest widok generujący edytory dla klasy `Appointment`, zapisany w pliku *MakeBooking.cshtml*.

Listing 18.2. *Widok edytora*

```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)</p>
    <p>Data wizyty: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki</p>

    <input type="submit" value="Wyślij rezerwację" />
}
```

Na listingu 18.3 zamieszczona jest klasa kontrolera, `AppointmentController`, posiadająca metody akcji operujące na obiektach `Appointment`.

Listing 18.3. *Klasa AppointmentController*

```
public class AppointmentController : Controller {
    private IAppointmentRepository repository;

    public AppointmentController(IAppointmentRepository repo) {
        repository = repo;
    }

    public ViewResult MakeBooking() {
        return View(new Appointment { Date = DateTime.Now });
    }

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {

        repository.SaveAppointment(appt);
        return View("Completed", appt);
    }
}
```

Kontroler realizuje znany już nam wzorzec. Metoda akcji `MakeBooking` generuje widok *MakeBooking.cshtml*. Widok ten zawiera formularz, którego dane są przesyłane do drugiej wersji metody `MakeBooking`, oczekującej parametru `Appointment`. Łącznik modelu tworzy obiekt `Appointment` na podstawie wartości elementów formularza HTML, a następnie przekazuje go do metody akcji, zapisującej nowy obiekt w repozytorium, które zostanie dostarczone poprzez wstrzykiwanie zależności (utworzyliśmy tylko szkielet repozytorium — dane wizyty są ignorowane, ponieważ chcemy się tu skupić wyłącznie na kontroli poprawności). Po przekazaniu wizyty do repozytorium kontroler generuje widok *Completed.cshtml*, który pozwala przekazać informację zwrotną dla użytkownika. Na rysunku 18.1 przedstawiona jest ta sekwencja widoków.

Rysunek 18.1. *Kolejność widoków w przykładowej aplikacji*

Obecnie nasza aplikacja będzie akceptowała dowolne przesłane przez użytkownika dane, ale aby zachować integralność aplikacji i modelu domeny, wymagamy spełnienia trzech warunków przed zaakceptowaniem przesłanych przez użytkownika danych wizyty:

- użytkownik musi podać nazwisko,
- użytkownik musi podać przyszłą datę (w formacie `mm-dd-rrrr`),
- użytkownik musi zaznaczyć pole wyboru informujące o zaakceptowaniu zasad.

Kontrola poprawności modelu jest procesem umożliwiającym wymuszenie tych wymagań. W kolejnych punktach pokażemy różne techniki pozwalające weryfikować dane przesłane przez użytkownika oraz przekazanie użytkownikowi informacji, dlaczego nie możemy wykorzystać dostarczonych danych.

Jawna kontrola poprawności modelu

Najbardziej bezpośrednim sposobem kontroli poprawności modelu jest wykonanie tej operacji w metodzie akcji. Realizacja tego zadania jest zamieszczona na listingu 18.4.

Listing 18.4. *Jawna kontrola poprawności modelu*

```
[HttpPost]
public ViewResult MakeBooking(Appointment appt) {

    if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date) {
        ModelState.AddModelError("Date", "Proszę podać przyszłą datę.");
    }

    if (!appt.TermsAccepted) {
        ModelState.AddModelError("TermsAccepted", "Zaakceptowanie zasad jest wymagane.");
    }

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}
```

Sprawdzamy tu wartości przypisane przez łącznik modelu do właściwości obiektu parametru i rejestrujemy wszystkie znalezione błędy we właściwości `ModelState`, którą kontroler dziedziczy po swojej klasie bazowej. Dla przykładu przeanalizujemy, w jaki sposób sprawdzamy właściwość `ClientName`:

```
if (string.IsNullOrEmpty(appt.ClientName)) {
    ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
}
```

Chcemy, aby użytkownik podał wartość tej właściwości, więc do jej sprawdzenia wykorzystujemy statyczną metodę `string.IsNullOrEmpty`. Jeżeli nie otrzymamy wartości, wywołujemy metodę `ModelState.AddModelError`, podając nazwę właściwości, której dotyczy błąd (`ClientName`), oraz komunikat, jaki powinien być wyświetlony użytkownikowi, aby mu pomóc przy rozwiązaniu problemu (*Proszę podać swoje nazwisko.*).

Za pomocą właściwości `ModelState.IsValidField` jesteśmy w stanie sprawdzić, czy łącznik modelu był w stanie przypisać wartość do właściwości. Wykonaliśmy to dla właściwości `Date`, aby upewnić się, czy łącznik modelu skonwertował przekazaną przez użytkownika wartość; jeżeli nie, nie ma sensu wykonywać dodatkowych sprawdzeń i raportować kolejnych błędów.

Po sprawdzeniu wszystkich właściwości w obiekcie modelu odczytujemy właściwość `ModelState.IsValid` w celu sprawdzenia, czy wystąpiły błędy. Właściwość ta ma wartość `false`, jeżeli zarejestrowane zostały jakiekolwiek błędy lub jeżeli łącznik modelu napotkał problemy:

```
if (ModelState.IsValid) {
    repository.SaveAppointment(appt);
    return View("Completed", appt);
} else {
    return View();
}
```

Jeżeli nie wystąpiły błędy, zapisujemy wizytę za pomocą repozytorium i generujemy widok *Completed*. Jeżeli pojawiły się problemy, wywołujemy po prostu metodę `View` bez parametrów. Powoduje to ponowne wygenerowanie bieżącego widoku, dzięki czemu użytkownik zobaczy informacje o błędach i będzie mógł skorygować wprowadzane dane.

Metody pomocnicze widoku szablonowego używane do wygenerowania edytorów dla właściwości modelu obsługują błędy kontroli poprawności. Jeżeli dla właściwości zostanie zaraportowany błąd, metoda pomocnicza doda do pola tekstowego klasę CSS o nazwie `input-validation-error`. Plik *~/Content/Site.css* zawiera następującą domyślną definicję tego stylu:

```
.input-validation-error {
    border: 1px solid #ff0000;
    background-color: #ffeeee;
}
```

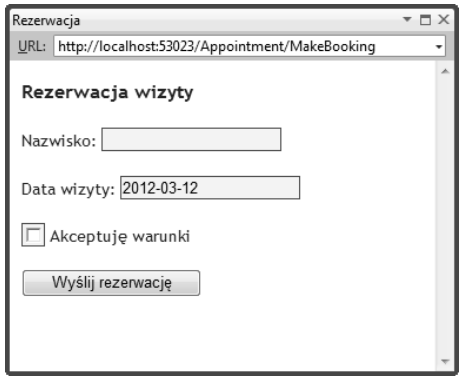
Powoduje ona ustawienie czerwonej ramki oraz różowego tła w elemencie, w którym wystąpił błąd. Na rysunku 18.2 przedstawiony jest ten efekt dla wszystkich trzech właściwości.

Wyświetlanie komunikatów kontroli poprawności

Style CSS dołączane przez metody widoku szablonowego do elementów tekstowych informują o wystąpieniu problemu w polu, ale nie pozwalają przekazać mu danych o samym problemie. Na szczęście dostępne są inne metody pomocnicze HTML wspierające nas w tym zadaniu. Na listingu 18.5 pokazane jest użycie jednej z tych metod w widoku `MakeBooking`.

Listing 18.5. *Użycie metody pomocniczej podsumowania*

```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
```



Rysunek 18.2. *Błędy powodują wyróżnienie elementów*

Nadawanie stylu polom wyboru

Niektóre przeglądarki, w tym Chrome i Firefox, ignorują style nadawane polom tekstowym, co prowadzi do powstania niespójnego wyglądu strony. Rozwiązaniem jest zastąpienie szablonu edytora `Boolean` własnym, umieszczonym w *~/Views/Shared/EditorTemplates/Boolean.cshtml*, w którym pole wyboru jest osadzone w elemencie `div`. Poniżej jest zamieszczony używany przez nas szablon, ale możesz go dostosować do swoich aplikacji:

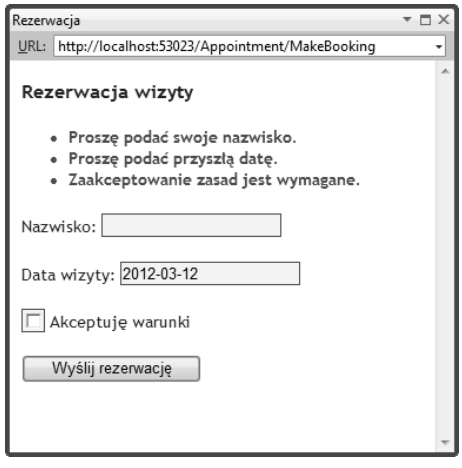
```
@model bool?

@if (ViewData.ModelMetadata.IsNullableValueType) {
    @Html.DropDownListFor(m => m, new SelectList(new [] {"Nie ustawiono", "Prawda", "Fałsz"},
Model))
} else {
    ModelState state = ViewData.ModelState[ViewData.ModelMetadata.PropertyName];
    bool value = Model ?? false;

    if (state != null && state.Errors.Count > 0) {
        <div class="input-validation-error" style="float:left">
            @Html.CheckBox("", value)
        </div>
    } else {
        @Html.CheckBox("", value)
    }
}
```

W szablonie tym umieściliśmy pole wyboru w elemencie `div`, do którego dołączany jest styl `input-validation-error`, jeżeli zostały zarejestrowane błędy modelu skojarzone z właściwością obsługiwaną przez szablon. Więcej informacji na temat wymiany szablonów edytorów znajduje się w rozdziale 16.

```
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    @Html.ValidationSummary()

    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)</p>
```

```
    <p>Data wizyty: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki</p>

    <input type="submit" value="Wyślij rezerwację" />
}
```

Metoda pomocnicza `Html.ValidationSummary` pozwala wyświetlić podsumowanie błędów kontroli poprawności zarejestrowanych dla strony. Jeżeli nie ma błędów, metoda pomocnicza nie generuje żadnego kodu HTML. Na rysunku 18.3 przedstawione jest działanie pola podsumowania.



Rysunek 18.3. *Wyświetlanie podsumowania kontroli poprawności*

Pole podsumowania wyświetla komunikaty zarejestrowane przez metodę akcji w obiekcie `ModelState`. Na listingu 18.6 zamieszczony jest kod HTML wygenerowany przez tę metodę.

Listing 18.6. *Kod HTML wygenerowany przez metodę pomocniczą ValidationSummary*

```
<div class="validation-summary-errors" data-valmsg-summary="true">
    <ul>
        <li>Proszę podać swoje nazwisko.</li>
        <li>Proszę podać przyszłą datę.</li>
        <li>Zaakceptowanie zasad jest wymagane.</li>
    </ul>
</div>
```

Informacja o błędach są wyrażane jako lista umieszczona w elemencie `div`, do którego jest dołączona klasa CSS `validation-summary-errors`. Styl jest zdefiniowany w pliku *~/Content/Site.css* i w razie potrzeby może być zmieniony. Domyślny styl powoduje użycie pogrubionego, czerwonego tekstu:

```
.validation-summary-errors {
    font-weight: bold;
    color: #ff0000;
}
```

Metoda `ValidationSummary` posiada kilka przeciążonych wersji; najprzydatniejsze są zamieszczone w tabeli 18.1.

Niektóre z przeciążonych wersji metody `ValidationSummary` pozwalają nam na wskazanie, że powinny być wyświetlone wyłącznie *komunikaty o błędach na poziomie modelu*. Błędy, jakie rejestrowaliśmy do tej pory w `ModelState`, były *błędami na poziomie właściwości*, co oznacza, że wystąpił problem z wartością podaną w danej właściwości i jej zmiana może rozwiązać problem.

Tabela 18.1. *Najprzydatniejsze przeciążone wersje metody ValidationSummary*

Przeciążona metoda	Opis
`Html.ValidationSummary()`	Generuje podsumowanie dla wszystkich błędów.
`Html.ValidationSummary(bool)`	Jeżeli parametr `bool` ma wartość `true`, to wyświetlane są tylko komunikaty o błędach poziomu modelu (patrz wyjaśnienie pod tabelą). Jeżeli parametr ma wartość `false`, wyświetlane są komunikaty o wszystkich błędach.
`Html.ValidationSummary(string)`	Wyświetla komunikat (przekazany w parametrze typu `string`) przed podsumowaniem wszystkich błędów.
`Html.ValidationSummary(bool, string)`	Wyświetla komunikat przed błędami. Jeżeli parametr `bool` ma wartość `true`, wyświetlane są wyłącznie komunikaty o błędach poziomu modelu.

Z kolei błędy na poziomie modelu mogą być wykorzystywane, jeżeli powstaje problem z interakcją pomiędzy dwoma właściwościami (lub większą ich liczbą). Wyobraźmy sobie, że klient Jan nie może składać rezerwacji w poniedziałki. Na listingu 18.7 pokazany jest sposób wymuszenia tej zasady i raportowania problemów za pomocą błędów kontroli poprawności na poziomie modelu.

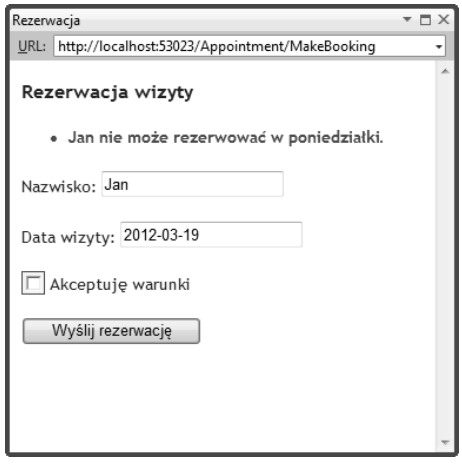
Listing 18.7. *Błąd kontroli poprawności na poziomie modelu*

```
public ViewResult MakeBooking(Appointment appt) {

    if (string.IsNullOrEmpty(appt.ClientName)) {
        ModelState.AddModelError("ClientName", "Proszę podać swoje nazwisko.");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date) {
        ModelState.AddModelError("Date", "Proszę podać przyszłą datę.");
    }

    if (!appt.TermsAccepted) {
        ModelState.AddModelError("TermsAccepted", "Zaakceptowanie zasad jest wymagane.");
    }

    if (ModelState.IsValidField("ClientName") && ModelState.IsValidField("Date")
        && appt.ClientName == "Jan" && appt.Date.DayOfWeek == DayOfWeek.Monday) {
            ModelState.AddModelError("", "Jan nie może rezerwować w poniedziałki.");
    }

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}
```

Zanim sprawdzimy, czy Jan próbuje zarezerwować wizytę na poniedziałek, korzystamy z metody `ModelState.IsValidField` w celu upewnienia się, że mamy prawidłowe wartości pól `ClientName` oraz `Date`. Oznacza to, że nie możemy wygenerować błędu poziomu modelu do momentu spełnienia warunków dla właściwości. Błędy poziomu modelu rejestrujemy, podając pusty ciąg znaków (`""`) jako pierwszy parametr metody `ModelState.AddModelError`:

```
ModelState.AddModelError("", "Jan nie może rezerwować w poniedziałki.");
```

Użycie metody `ValidationSummary` z parametrem `bool` pozwala nam wyświetlić informacje wyłącznie o błędach na poziomie modelu, co jest pokazane na rysunku 18.4.



Rysunek 18.4. *Wyświetlanie wyłącznie informacji o błędach na poziomie modelu*

Jak można zauważyć na rysunku, mamy tu dwa błędy kontroli poprawności. Pierwszym jest błąd na poziomie modelu, który wystąpił, gdy Jan próbował zarezerwować wizytę na poniedziałek. Drugi wynika z braku zaznaczenia pola akceptacji zasad. Ponieważ w podsumowaniu wyświetlamy wyłącznie komunikaty o błędach poziomu modelu, użytkownik nie zobaczy żadnej informacji o braku zaznaczenia opcji.

Wyświetlanie komunikatów kontroli poprawności poziomu właściwości

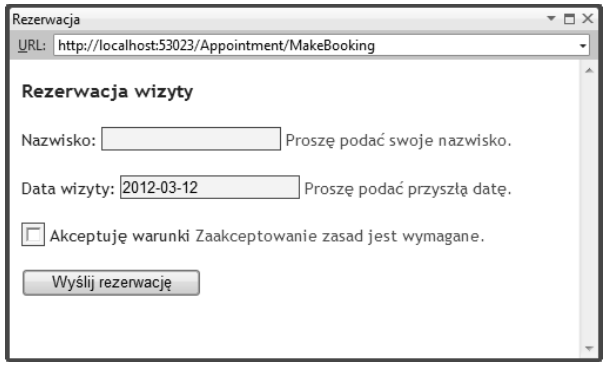
Powodem ograniczania komunikatów o błędach wyświetlanych w podsumowaniu do błędów poziomu modelu jest możliwość wyświetlania komunikatów o błędach poziomu właściwości obok pól. W takim przypadku nie chcemy powielać komunikatów z poziomu właściwości. Na listingu 18.8 zamieszczony jest zaktualizowany widok `MakeBooking`, w którym komunikaty o błędach poziomu modelu są wyświetlane w podsumowaniu, a o błędach na poziomie właściwości — obok odpowiedniego pola.

Metoda pomocnicza `Html.ValidationMessageFor` wyświetla komunikaty o błędach dla poszczególnych właściwości modelu. Na rysunku 18.5 przedstawiony jest wynik działania tej metody.

Listing 18.8. *Użycie komunikatów o błędach poziomu właściwości*

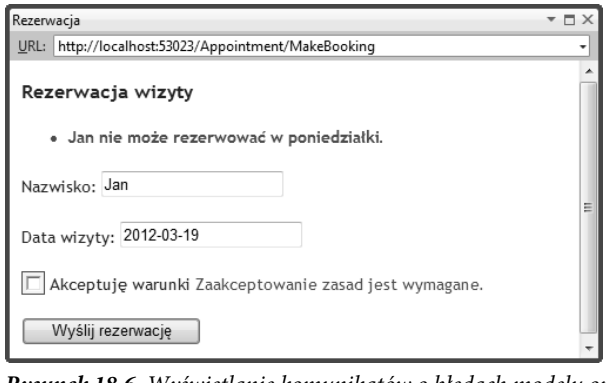
```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    @Html.ValidationSummary(true)

    <p>
        Nazwisko: @Html.EditorFor(m => m.ClientName)
```

```
        @Html.ValidationMessageFor(m => m.ClientName)
    </p>
    <p>
        Data wizyty: @Html.EditorFor(m => m.Date)
        @Html.ValidationMessageFor(m => m.Date)
    </p>
    <p>
        @Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki
        @Html.ValidationMessageFor(m => m.TermsAccepted)
    </p>

    <input type="submit" value="Wyślij rezerwację" />
}
```



Rysunek 18.5. *Użycie komunikatów kontroli poprawności dla pojedynczych właściwości*

Oczywiście nie ma sensu wyświetlanie informacji o błędach z poziomu modelu obok określonej właściwości, więc umieścimy je w osobnej sekcji, wygenerowanej za pomocą metody pomocniczej `Html.ValidationSummary`, jak pokazano na rysunku 18.6.



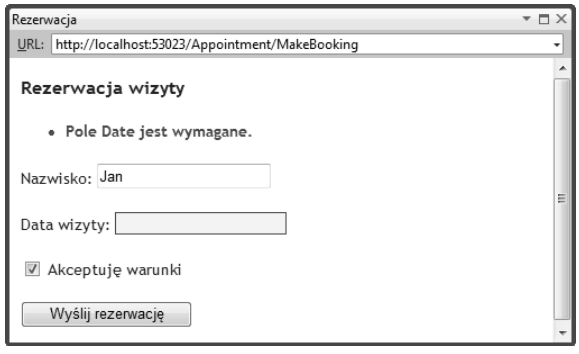
Rysunek 18.6. *Wyświetlanie komunikatów o błędach modelu oraz właściwości*

Użycie alternatywnych technik kontroli poprawności

Wykonywanie kontroli poprawności w metodzie akcji jest tylko jedną z technik dostępnych w bibliotece MVC. W kolejnych punktach przedstawimy inne podejścia do tego zagadnienia.

Kontrola poprawności w łączniku modelu

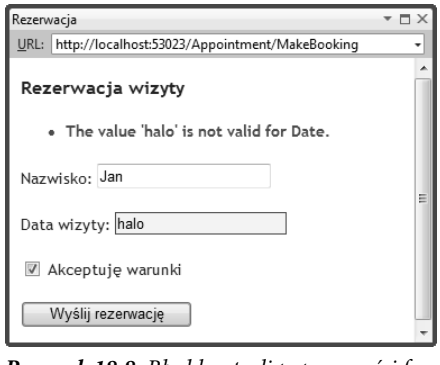
Domyślny łącznik modelu wykonuje kontrolę poprawności w ramach procesu dołączania. Na rysunku 18.7 pokazane jest, co się stanie, gdy wyczyścimy pole `Date` i wyślemy dane formularza.



Rysunek 18.7. *Komunikat kontroli poprawności z łącznika modelu*

Zauważ, że dla pola `Date` wyświetlany jest komunikat o błędzie (wróciliśmy tu do wyświetlania wszystkich komunikatów o błędach w podsumowaniu). Komunikat ten został dodany przez łącznik modelu, ponieważ nie był on w stanie utworzyć obiektu `DateTime` z pustego pola formularza.

Łącznik modelu realizuje podstawową kontrolę poprawności dla każdej właściwości w obiekcie modelu. Jeżeli wartość nie zostanie dostarczona, będzie wygenerowany komunikat pokazany na rysunku 18.7. Jeżeli podamy wartość, której nie da się przekształcić na typ właściwości modelu, wyświetlony zostanie inny komunikat (rysunek 18.8).



Rysunek 18.8. *Błąd kontroli poprawności formatu wyświetlany przez łącznik modelu*

Wbudowana klasa łącznika modelu, `DefaultModelBinder`, posiada kilka użytecznych metod, które możemy nadpisać w celu dodania kontroli poprawności do łącznika. Metody te są opisane w tabeli 18.2.

Możemy nadpisać metody zamieszczone w tabeli 18.2, dodając w ten sposób logikę kontroli poprawności do łącznika. Przykład jest zamieszczony na listingu 18.9. Ilustruje on jedynie możliwość użycia tego rozwiązania, ale zalecamy korzystanie z dostawców kontroli poprawności przedstawionych w dalszej części rozdziału.

Kontrola poprawności w łączniku modelu wygląda na bardziej skomplikowaną, niż jest w rzeczywistości. Logika kontroli poprawności jest taka sama jak w metodzie akcji. Kontrolę poprawności na poziomie właściwości realizujemy w metodzie `SetProperty`. Metoda ta jest wywoływana dla każdej właściwości modelu. Z punktu widzenia procesu dołączania obiekt modelu nie został w pełni utworzony, więc przy kontroli poprawności korzystamy z parametrów metody. Odczytujemy nazwę właściwości z parametru `PropertyDescriptor`, przypisaną wartość z parametru `object`, a dostęp do `ModelState` mamy poprzez parametr `BindingContext`.

Tabela 18.2. *Metody klasy DefaultModelBinder pozwalające na dodanie kontroli poprawności do procesu dołączania modelu*

Metoda	Opis	Domyślna implementacja
`OnModelUpdated`	Wywoływana w momencie, gdy łącznik próbuje przypisać wartości do wszystkich właściwości obiektu modelu.	Stosuje zasady poprawności zdefiniowane przez metadane modelu oraz rejestruje wszystkie błędy w `ModelState`. Użycie metadanych do kontroli poprawności zostanie opisane w dalszej części rozdziału.
`SetProperty`	Wywoływana, gdy łącznik chce przypisać wartość do określonej właściwości.	Jeżeli właściwość nie może zawierać wartości `null`, a nie ma wartości do przypisania, w `ModelState` rejestrowany jest błąd *The <nazwa> field is required* (rysunek 18.7). Jeżeli istnieje wartość, która nie może być skonwertowana, to rejestrowany jest błąd *The value <wartość> is not valid for <nazwa>* (rysunek 18.8).

Listing 18.9. *Dziedziczenie po klasie DefaultModelBinder w celu dodania kontroli poprawności do procesu dołączania*

```
using System;
using System.ComponentModel;
using System.Web.Mvc;
using MvcApp.Models;
namespace MvcApp.Infrastructure {
    public class ValidatingModelBinder : DefaultModelBinder {

        protected override void SetProperty(ControllerContext controllerContext,
            ModelBindingContext bindingContext, PropertyDescriptor propertyDescriptor,
            object value) {

            // upewnij się, że zostanie wywołana implementacja bazowa
            base.SetProperty(controllerContext, bindingContext, propertyDescriptor, value);

            // wykonanie naszej kontroli poprawności na poziomie właściwości
            switch (propertyDescriptor.Name) {
                case "ClientName":
                    if (string.IsNullOrEmpty((string)value)) {
                        bindingContext.ModelState.AddModelError("ClientName",
                            "Proszę podać nazwisko.");
                    }
                    break;
                case "Date":
                    if (bindingContext.ModelState.IsValidField("Date") &&
                        DateTime.Now > ((DateTime)value)) {
                        bindingContext.ModelState.AddModelError("Date",
                            "Proszę podać przyszłą datę.");
                    }
                    break;
                case "TermsAccepted":
                    if (!((bool)value)) {
                        bindingContext.ModelState.AddModelError("TermsAccepted",
                            "Zaakceptowanie warunków jest obowiązkowe.");
                    }
                    break;
            }
        }

        protected override void OnModelUpdated(ControllerContext controllerContext,
            ModelBindingContext bindingContext) {
```

```
            // upewnij się, że zostanie wywołana implementacja bazowa
            base.OnModelUpdated(controllerContext, bindingContext);

            // pobierz model
            Appointment model = bindingContext.Model as Appointment;
            // wykonaj kontrolę poprawności na poziomie modelu
            if (model != null &&
                bindingContext.ModelState.IsValidField("ClientName") &&
                bindingContext.ModelState.IsValidField("Date") &&
                model.ClientName == "Jan" &&
                model.Date.DayOfWeek == DayOfWeek.Monday) {
                bindingContext.ModelState.AddModelError("",
                    "Jan nie może rezerwować w poniedziałki.");
            }
        }
    }
}
```

Kontrolę poprawności na poziomie modelu realizujemy w metodzie `OnModelUpdate`. W tym momencie procesu dołączania wartości są przypisane do obiektu modelu, więc możemy odczytywać właściwości w celu wykonania kontroli.

■ **Ostrzeżenie** Przy korzystaniu z łącznika do realizacji kontroli poprawności jest ważne, aby wywołać bazowe implementacje metod `SetProperty` oraz `OnModelUpdated`. W przeciwnym razie stracimy część ważnych funkcji, takich jak kontrola poprawności modelu z użyciem metadanych (przedstawiona w dalszej części rozdziału).

Kolejnym krokiem jest zarejestrowanie naszego łącznika w metodzie `Application_Start` znajdującej się w pliku *Global.asax*:

```
protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    ModelBinders.Binders.Add(typeof(Appointment), new ValidatingModelBinder());

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

Ponieważ przenieśliśmy logikę kontroli poprawności klasy modelu `Appointment` do łącznika, możemy uprościć metodę akcji do postaci pokazanej na listingu 18.10.

Listing 18.10. *Uproszczona metoda akcji*

```
[HttpPost]
public ViewResult MakeBooking(Appointment appt) {

    if (ModelState.IsValid) {
        repository.SaveAppointment(appt);
        return View("Completed", appt);
    } else {
        return View();
    }
}
```

Wszystkie błędy zarejestrowane w łączniku będą dostępne w metodzie akcji, w kolekcji `ModelState`. Dzięki temu możemy sprawdzić właściwość `ModelState.IsValid`, która będzie miała wartość `false` w przypadku, gdy łącznik zarejestruje błędy kontroli poprawności. Jeżeli nie występowały błędy, zapisujemy obiekt `Appointment` za pomocą repozytorium i generujemy widok *Completed*. Jeżeli wystąpiły błędy walidacji, generujemy bieżący widok i wyświetlamy komunikaty o błędach.

Definiowanie zasad poprawności za pomocą metadanych

Nie musimy definiować logiki kontroli poprawności w sposób programowy. Biblioteka MVC pozwala na użycie metadanych do definiowania zasad poprawności modelu. Zaletą użycia metadanych jest wymuszenie zasad kontroli poprawności w każdym procesie dołączania danych do klasy modelu — w przeciwieństwie do wykorzystywania pojedynczych metod akcji. Atrybuty kontroli poprawności są wykrywane i wymuszane przez wbudowaną klasę łącznika modelu, `DefaultModelBinder`. Atrybuty te są stosowane w klasie modelu w sposób pokazany na listingu 18.11.

Listing 18.11. *Użycie atrybutów do definiowania zasad poprawności*

```
using System;
using System.ComponentModel.DataAnnotations;
using MvcApp.Infrastructure;

namespace MvcApp.Models {

    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        [Required(ErrorMessage="Proszę podać datę")]
        public DateTime Date { get; set; }
        [Range(typeof(bool), "true", "true", ErrorMessage="Zaakceptowanie warunków jest
        ↪obowiązkowe")]
        public bool TermsAccepted { get; set; }
    }
}
```

Użyliśmy tu dwóch atrybutów kontroli poprawności — `Required` oraz `Range`. Atrybut `Required` powoduje powstanie błędu kontroli poprawności, jeżeli użytkownik nie poda wartości dla właściwości. Atrybut `Range` pozwala określić akceptowalny zakres wartości. W tabeli 18.3 zamieszczone są wbudowane atrybuty kontroli poprawności.

Wszystkie atrybuty kontroli poprawności pozwalają nam określić własny komunikat o błędzie przez ustawienie właściwości `ErrorMessage`, tak jak w poniższym przykładzie:

```
[Required(ErrorMessage="Proszę podać datę.")]
```

Jeżeli nie podamy własnego komunikatu o błędzie, to będzie użyty domyślny, jak pokazano na rysunkach 18.7 i 18.8. Atrybuty kontroli poprawności są dosyć proste i pozwalają wyłącznie na kontrolę poprawności na poziomie właściwości. Mimo to mamy sporo narzędzi, aby zapewnić spójne działanie tego mechanizmu. Weźmy pod uwagę atrybut zastosowany dla właściwości `TermsAccepted`.

```
[Range(typeof(bool), "true", "true", ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
public bool TermsAccepted { get; set; }
```

Chcemy upewnić się, że użytkownik zaznaczy pole wyboru zaakceptowania warunków. Nie możemy użyć atrybutu `Required`, ponieważ metody pomocnicze widoku szablonowego generują ukryte pole HTML w celu upewnienia się, że otrzymamy wartość nawet w przypadku, gdy pole nie jest zaznaczone. Aby ominąć ten problem,

Tabela 18.3. *Wbudowane atrybuty kontroli poprawności*

Atrybut	Przykład	Opis
`Compare`	`[Compare("MyOtherProperty")]`	Dwie właściwości muszą mieć taką samą wartość. Jest to przydatne, jeżeli prosimy użytkownika o dwukrotne podanie tej samej danej, na przykład adresu e-mail lub hasła.
`Range`	`[Range(10, 20)]`	Wartość numeryczna (lub właściwość typu implementującego `IComparable`) musi mieć wartość pomiędzy podanym minimum i maksimum. Aby zdefiniować granicę tylko z jednej strony, należy użyć stałych `MinValue` lub `MaxValue` — na przykład `[Range(int.MinValue, 50)]`.
`RegularExpression`	`[RegularExpression("wzorzec")]`	Wartość znakowa musi pasować do zdefiniowanego wyrażenia regularnego. Zwróć uwagę, że wzorzec musi pasować do całego wyrażenia podanego przez użytkownika, a nie tylko do jego fragmentu. Domyślnie dopasowania rozpoznają wielkie i małe litery, ale można to zmienić, dodając modyfikator `(?i)` — czyli `[RegularExpression("(?i)wzorzec")]`.
`Required`	`[Required]`	Wartość musi być niepusta lub być ciągiem znaków zawierającym tylko spacje. Jeżeli chcesz traktować białe znaki jako prawidłową zawartość, użyj `[Required(AllowEmptyStrings = true)]`.
`StringLength`	`[StringLength(10)]`	Wartość znakowa musi być nie dłuższa niż podana wartość maksymalna. Możemy również określić minimalną długość: `[StringLength(10, MinimumLength=2)]`.

użyliśmy atrybutu `Range`, który pozwala określić typ oraz górną i dolną granicę w postaci wartości znakowych. Przez ustawienie obu granic na `true` utworzyliśmy odpowiednik atrybutu `Required` dla właściwości `bool` edytowanej za pomocą pola wyboru.

■ **Wskazówka** Atrybut `DataType` nie może być używany do kontroli poprawności danych użytkownika — stanowi wyłącznie podpowiedź przy emitowaniu wartości z zastosowaniem metod pomocniczych widoku szablonowego (opisanych w rozdziale 16.). Dlatego nie należy oczekiwać, że atrybut `DataType(DataType.EmailAddress)` pozwoli wymusić odpowiedni format.

Tworzenie własnego atrybutu kontroli poprawności

Sztuczka z użyciem atrybutu `Range` do odtworzenia działania atrybutu `Required` jest nieco dziwna. Na szczęście nie jesteśmy ograniczeni wyłącznie do wbudowanych atrybutów; można również tworzyć własne przez odziedziczenie po klasie `ValidationAttribute` i implementację własnej logiki kontroli poprawności. Przykład takiej klasy jest zamieszczony na listingu 18.12.

Listing 18.12. *Tworzenie własnego atrybutu kontroli poprawności*

```
public class MustBeTrueAttribute : ValidationAttribute {

    public override bool IsValid(object value) {
```

```
        return value is bool && (bool)value;
    }
}
```

Nasz nowy atrybut zmienia metodę `IsValid` z klasy bazowej. Jest to metoda wywoływana przez łącznik, do której przekazywana jest wartość wprowadzona przez użytkownika. W tym przykładzie nasza logika kontroli poprawności jest prosta — wartość jest prawidłowa, jeżeli jest typu `bool` o wartości `true`. Aby poinformować, że wartość jest prawidłowa, zwracamy `true` z metody `isValid`. Atrybutu tego możemy użyć w poniższy sposób:

```
[MustBeTrue(ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
public bool TermsAccepted { get; set; }
```

Jest to przyjemniejsze i łatwiejsze niż nadużywanie atrybutu `Range`. Możemy również dziedziczyć po wbudowanych atrybutach kontroli poprawności w celu rozszerzania zakresu ich funkcji. Na listingu 18.13 pokazujemy, jak wykorzystać atrybut `Required` do upewnienia się, że została podana data spełniająca nasze wymagania.

Listing 18.13. *Dziedziczenie po wbudowanym atrybucie kontroli poprawności*

```
public class FutureDateAttribute : RequiredAttribute {

    public override bool IsValid(object value) {
        return base.IsValid(value) &&
            value is DateTime &&
            ((DateTime)value) > DateTime.Now;
    }
}
```

Oprócz funkcji zawartej w klasie bazowej dodaliśmy własną logikę kontroli poprawności. Atrybutu tego używamy identycznie jak poprzedniego:

```
[DataType(DataType.Date)]
[FutureDate(ErrorMessage="Proszę podać przyszłą datę.")]
public DateTime Date { get; set; }
```

Tworzenie własnego atrybutu kontroli poprawności modelu

Przedstawione do tej pory atrybuty poprawności odnoszą się do poszczególnych właściwości modelu, co oznacza, że możemy generować tylko błędy kontroli poprawności na poziomie właściwości. Możemy również użyć metadanych do kontroli poprawności całego modelu w sposób pokazany na listingu 18.14.

Listing 18.14. *Atrybuty kontroli poprawności modelu*

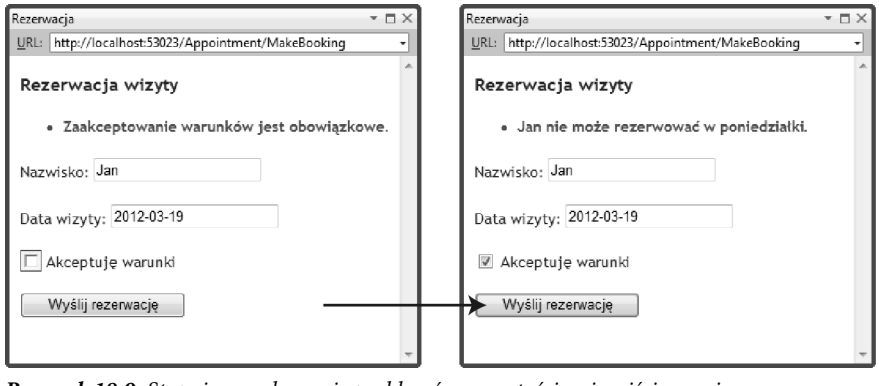
```
public class AppointmentValidatorAttribute : ValidationAttribute {

    public AppointmentValidatorAttribute() {
        ErrorMessage = "Jan nie może rezerwować w poniedziałki.";
    }

    public override bool IsValid(object value) {

        Appointment app = value as Appointment;

        if (app == null || string.IsNullOrEmpty(app.ClientName) || app.Date == null) {
            // nie mamy modelu właściwego typu lub nie mamy
            // wartości wymaganych właściwości ClientName oraz Date
            return true;
        } else {
            return !(app.ClientName == "Jan" && app.Date.DayOfWeek == DayOfWeek.Monday);
```

```
        }
    }
}
```

Parametr `object` przekazany przez łącznik modelu do metody `IsValid` powinien być typu `Appointment`. Nasz atrybut kontroli poprawności modelu dołączamy do samej klasy modelu:

```
[AppointmentValidator]
public class Appointment {

    [Required]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [FutureDate(ErrorMessage="Proszę podać przyszłą datę.")]
    public DateTime Date { get; set; }

    [MustBeTrue(ErrorMessage="Zaakceptowanie warunków jest obowiązkowe.")]
    public bool TermsAccepted { get; set; }
}
```

Nasz atrybut kontroli poprawności modelu nie będzie używany, jeżeli jakikolwiek z atrybutów poziomu właściwości zarejestrował błąd poprawności. Nie jest to dokładnie ten sam efekt, jaki osiągnęliśmy, umieszczając kod kontroli poprawności w metodzie akcji. Powoduje to ryzyko wydłużania procesu poprawiania błędów przez użytkownika. Jeżeli użytkownik na przykład poda wartości dla właściwości `ClientName` oraz `Date`, ale nie zaznaczy pola dla warunków, to atrybut `MustBeTrue` spowoduje wyświetlenie komunikatu o błędzie poprawności; jest on pokazany w lewej części rysunku 18.9.



Rysunek 18.9. *Stopniowe zgłaszanie problemów z wartościami wejściowymi*

Użytkownik poprawia błąd i ponownie wysyła dane — po czym otrzymuje inny komunikat, pokazany w prawej części rysunku 18.9. Z perspektywy użytkownika niejawnie akceptujemy wartość dla nazwiska i daty, nie oznaczając ich jako błędnych w pierwszym panelu. Może się to wydawać niewielkim problemem, ale warto przeanalizować każdą sytuację, która może frustrować użytkowników.

Definiowanie modeli samokontrolujących się

Inną techniką kontroli poprawności jest utworzenie modeli samokontrolujących się, w których kod weryfikacji poprawności jest częścią klasy modelu. Samokontrolujące się klasy modelu tworzymy przez zaimplementowanie interfejsu `IValidatableObject` w sposób pokazany na listingu 18.15.

Listing 18.15. *Samokontrolująca się klasa modelu*

```
public class Appointment : IValidatableObject {

    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext) {

        List<ValidationResult> errors = new List<ValidationResult>();

        if (string.IsNullOrEmpty(ClientName)) {
            errors.Add(new ValidationResult("Proszę podać nazwisko."));
        }

        if (DateTime.Now > Date) {
            errors.Add(new ValidationResult("Proszę podać przyszłą datę."));
        }

        if (errors.Count == 0 && ClientName == "Jan"
            && Date.DayOfWeek == DayOfWeek.Monday) {

            errors.Add(new ValidationResult("Jan nie może rezerwować w poniedziałki."));
        }

        if (!TermsAccepted) {
            errors.Add(new ValidationResult("Zaakceptowanie warunków jest obowiązkowe."));
        }

        return errors;
    }
}
```

Interfejs `IValidatableObject` definiuje jedną z metod, `Validate`. Metoda ta otrzymuje parametr `ValidationContext`; choć typ ten nie jest specyficzny dla MVC, to nie ma problemów z jego użyciem. Wynikiem metody `Validate` jest kolekcja obiektów `ValidationResult`, z których każdy reprezentuje pojedynczy błąd kontroli poprawności.

Jeżeli nasza klasa modelu implementuje interfejs `IValidatableObject`, to metoda `Validate` zostanie wywołana po przypisaniu wartości do każdej z właściwości modelu. Podejście to łączy ze sobą elastyczność umieszczenia logiki kontroli poprawności w metodzie akcji ze spójnością wywoływania przez proces dołączania modelu przy tworzeniu obiektów modelu. Niektórzy programiści nie lubią umieszczać logiki kontroli poprawności w klasie modelu, ale uważamy, że nieźle pasuje to do wzorca projektowego MVC — dodatkowo podoba się nam elastyczność i spójność.

Tworzenie własnego dostawcy kontroli poprawności

Alternatywnym podejściem do kontroli poprawności jest utworzenie własnego *dostawcy kontroli poprawności*. Realizujemy to przez odziedziczenie po klasie `ModelValidationProvider` i nadpisanie metody `GetValidators`. Na listingu 18.16 zamieszczony jest nasz dostawca kontroli poprawności, którego nazwaliśmy `CustomValidationProvider`.

Listing 18.16. *Własna klasa dostawcy kontroli poprawności*

```
public class CustomValidationProvider : ModelValidatorProvider {

    public override IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata,
        ControllerContext context) {

        if (metadata.ContainerType == typeof(Appointment)) {
            return new ModelValidator[] {
            new AppointmentPropertyValidator(metadata, context)
        };
        } else if (metadata.ModelType == typeof(Appointment)) {
            return new ModelValidator[] {
            new AppointmentValidator(metadata, context)
        };
        }
        return Enumerable.Empty<ModelValidator>();
    }
}
```

Metoda `GetValidation` jest wywoływana jeden raz dla każdej właściwości modelu, a następnie dla całego modelu. Wynikiem metody jest lista obiektów `ModelValidator`. Każdy ze zwróconych obiektów `ModelValidator` będzie użyty do zweryfikowania właściwości lub modelu.

Możemy zdefiniować metodę `GetValidation` w dowolny sposób. Jeżeli nie chcemy oferować kontroli poprawności dla właściwości lub modelu, to po prostu zwracamy pustą listę. Aby zademonstrować funkcję dostawcy kontroli poprawności, zdecydowaliśmy się zaimplementować jeden obiekt dla właściwości klasy `Appointment` oraz jeden dla klasy `Appointment`.

Aby sprawdzić, do jakich celów został wywołany obiekt kontroli poprawności, odczytujemy wartość właściwości `ModelMetadata` przekazanej do metody jako parametr. Dostępne trzy parametry tej metody są opisane w tabeli 18.4.

Tabela 18.4. *Przydatne właściwości klasy ModelMetadata*

Właściwość	Opis
`ContainerType`	Gdy zostaniemy poproszeni o dostarczenie obiektu kontroli poprawności dla właściwości, zwraca typ zawierającego ją modelu. Jeżeli na przykład jesteśmy proszeni o obiekt kontroli poprawności dla właściwości `Appointment.ClientName`, `ContainerType` będzie zwracać typ klasy `Appointment`.
`PropertyName`	Zwraca nazwę właściwości, dla której dostarczany jest obiekt kontroli poprawności. Jeżeli na przykład jesteśmy proszeni o obiekt kontroli poprawności dla właściwości `Appointment.ClientName`, `PropertyType` będzie zwracać wartość `ClientName`.
`ModelType`	Jeżeli jesteśmy proszeni o obiekt kontroli poprawności dla modelu, to ta właściwość będzie zwracać typ obiektu modelu.

■ **Wskazówka** Poniższy przykład pokazujemy wyłącznie jako ilustrację sposobu dołączania własnych dostawców kontroli poprawności do biblioteki. *Nie powinieneś* używać tej techniki w zwykłych scenariuszach kontroli poprawności, w których zastosowanie atrybutów metadanych lub `IValidatableObject` będzie wystarczające i znacznie prostsze. Z dostawców kontroli poprawności powinno się korzystać wyłącznie w skomplikowanych scenariuszach, na przykład przy dynamicznym ładowaniu zasad poprawności z bazy danych lub implementowaniu własnej biblioteki kontroli poprawności.

Na listingu 18.17 zamieszczona jest nasza klasa dziedzicząca po `ModelBinder`, używana dla właściwości.

Listing 18.17. *Obiekt kontroli poprawności działający na poziomie właściwości*

```
public class AppointmentPropertyValidator : ModelValidator {

    public AppointmentPropertyValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context) {
    }

    public override IEnumerable<ModelValidationResult> Validate(object container) {

        Appointment appt = container as Appointment;

        if (appt != null) {

            switch (Metadata.PropertyName) {
                case "ClientName":
                    if (string.IsNullOrEmpty(appt.ClientName)) {
                        return new ModelValidationResult[] {
                        new ModelValidationResult {
                            MemberName = "ClientName",
                            Message = "Proszę podać nazwisko."
                        }};
                    }
                    break;
                case "Date":
                    if (appt.Date == null || DateTime.Now > appt.Date) {
                        return new ModelValidationResult[] {
                        new ModelValidationResult {
                        MemberName = "Date",
                        Message = "Proszę podać przyszłą datę."
                    }};
                    }
                    break;
                case "TermsAccepted":
                    if (!appt.TermsAccepted) {
                        return new ModelValidationResult[] {
                        new ModelValidationResult {
                            MemberName = "TermsAccepted",
                            Message = "Zaakceptowanie warunków jest obowiązkowe."
                        }};
                    }
                    break;
            }
        }
        return Enumerable.Empty<ModelValidationResult>();
    }
}
```

Obiekt `ModelData` przekazany do konstruktora informuje o tym, którą właściwość powinna kontrolować dana instancja klasy `AppointmentPropertyValidator`. Po wywołaniu metody `Validate` przełączamy się na wartość `Metadata.PropertyName` i wykonujemy odpowiednią weryfikację. Jeżeli wystąpi błąd, zwracamy listę zawierającą jeden obiekt `ModelValidationResult`. System kontroli poprawności modelu pozwala na zwrócenie w razie potrzeby różnych błędów, ale nie potrzebujemy tego, ponieważ wykonujemy tylko jedno sprawdzenie dla każdej właściwości klasy `Appointment`.

Proces kontroli poprawności całego modelu jest nieco inny. Na listingu 18.18 zamieszczona jest klasa kontroli poprawności.

Listing 18.18. *Obiekt kontroli poprawności działający na poziomie modelu*

```
public class AppointmentValidator : ModelValidator {

    public AppointmentValidator(ModelMetadata metadata, ControllerContext context)
        : base(metadata, context) {
    }

    public override void Validate(Appointment container,
        IList<ModelValidationResult> errors) {

        Appointment appt = (Appointment)Metadata.Model;

        if (appt.ClientName == "Jan" && appt.Date.DayOfWeek == DayOfWeek.Monday) {
            errors.Add(new ModelValidationResult {
                MemberName = "",
                Message = "Jan nie może rezerwować w poniedziałki."
            });
        }
    }
}
```

W czasie kontroli modelu nie istnieje kontener, więc jego parametr ma wartość `null`. Na początek odczytujemy model z właściwości `Modeldata.Model`, a następnie wykonujemy kontrolę poprawności. Aby zaraportować błąd na poziomie modelu, ustawiamy właściwość `MemberName` w obiekcie `ModelValidationResult` na pusty ciąg znaków (`""`).

■ **Wskazówka** Biblioteka MVC wywoła nasz obiekt kontroli poprawności modelu wyłącznie wtedy, gdy żaden z obiektów kontroli poprawności nie zwróci błędu. Jest to rozsądne, ponieważ bazuje na założeniu, że nie można sprawdzać poprawności modelu, jeżeli wystąpiły błędy we właściwościach.

Rejestrowanie własnego dostawcy kontroli poprawności

Musimy teraz zarejestrować własnego dostawcę kontroli poprawności w bibliotece MVC, co realizujemy w metodzie `Application_Start` znajdującej się w pliku *Globlal.asax*. Na listingu 18.19 zamieszczony jest sposób rejestracji naszej klasy dostawcy kontroli poprawności dla klasy `Appointment`.

Listing 18.19. *Rejestrowanie własnego dostawcy kontroli poprawności*

```
protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    ModelValidatorProviders.Providers.Add(new CustomValidationProvider());

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

Naszą klasę dodajemy do zbioru dostawców kontroli poprawności za pomocą metody `ModelValidatorProviders.Providers.Add`. Nasz dostawca będzie użyty równolegle z wbudowanymi dostawcami, co oznacza, że można również wykorzystać inne techniki przedstawione w tym rozdziale. Jeżeli chcesz usunąć pozostałych dostawców, możesz to zrobić za pomocą metody `Clear`, zanim dodasz własnego dostawcę.

```
ModelValidatorProviders.Providers.Clear();
```

Użycie kontroli poprawności po stronie klienta

Do tej pory demonstrowaliśmy techniki kontroli poprawności będące przykładami *kontroli poprawności po stronie serwera*. Techniki te wymagają przesłania danych na serwer, skontrolowania ich na serwerze, a następnie odesłania wyniku kontroli (informacji o udanym przetworzeniu danych lub listy błędów wymagających skorygowania).

W aplikacji WWW użytkownicy wymagają zwykle szybkiej kontroli poprawności — bez konieczności wysyłania czegokolwiek na serwer. Taki mechanizm *kontroli poprawności po stronie klienta* jest zwykle implementowany z użyciem JavaScriptu. Dane wprowadzone przez użytkownika są kontrolowane przed wysłaniem na serwer, dzięki czemu reakcja aplikacji jest natychmiastowa i użytkownik od razu może korygować problemy.

Biblioteka MVC wspiera *nieprzeszkadzającą kontrolę poprawności po stronie klienta*. Słowo *nieprzeszkadzająca* oznacza, że zasady kontroli poprawności są wyrażane za pomocą atrybutów dodawanych do generowanych elementów HTML. Są one interpretowane przez bibliotekę JavaScript będącą częścią biblioteki MVC, wykorzystującą wartości atrybutów do skonfigurowania biblioteki jQuery Validation, która realizuje faktyczną kontrolę poprawności.

■ **Wskazówka** W MVC 3 skorzystano z biblioteki jQuery Validation, natomiast we wcześniejszych wersjach używane były biblioteki JavaScript napisane w firmie Microsoft. Nie były one najlepsze i choć są nadal dostępne w bibliotece, nie ma powodu z nich korzystać.

Słowo *nieprzeszkadzający* jest używane powszechnie w kontekście kodu JavaScript. Jest to luźny termin wskazujący na trzy cechy kodu. Pierwszą jest oddzielenie kodu JavaScript realizującego kontrolę poprawności od elementów HTML, dzięki czemu nie musimy dołączać logiki kontroli poprawności do naszych widoków, a generowany kod HTML jest czytelniejszy.

Drugą cechą jest realizacja kontroli poprawności z użyciem *progresywnego rozszerzania*. Oznacza to, że w przypadku braku obsługi wszystkich funkcji JavaScript przez przeglądarkę kontrola poprawności będzie realizowana z zastosowaniem prostszych technik. Jeżeli użytkownik na przykład zablokował JavaScript, to będzie wykorzystywana kontrola poprawności po stronie serwera bez wpływania w jakikolwiek sposób na działanie aplikacji (nie będą wyświetlane nieprzyjemne komunikaty o błędach ani nie będzie wymagane wykonanie dodatkowych akcji).

Trzecią cechą jest użycie zbioru praktyk pozwalających na ograniczenie wpływu niespójności w działaniu przeglądarek. Do tego tematu wrócimy w czasie przedstawiania biblioteki jQuery w rozdziale 20.

W kolejnych punktach pokażemy, w jaki sposób działają wbudowane mechanizmy kontroli poprawności, oraz zademonstrujemy sposoby ich rozszerzenia w celu zapewnienia własnych mechanizmów kontroli poprawności po stronie klienta.

■ **Wskazówka** Kontrola poprawności po stronie klienta skupia się na weryfikowaniu pojedynczych właściwości. Faktycznie trudno jest skonfigurować kontrolę poprawności po stronie klienta dla modelu, używając mechanizmów dostarczonych w bibliotece MVC. Z tego powodu większość aplikacji MVC 3 korzysta z kontroli poprawności po stronie klienta dla właściwości i bazuje na kontroli poprawności po stronie serwera dla całego modelu, stosując jedną z technik przedstawionych w poprzedniej części tego rozdziału.

Aktywowanie i wyłączanie kontroli poprawności po stronie klienta

Kontrola poprawności po stronie klienta jest sterowana za pomocą dwóch ustawień w pliku *Web.config*, pokazanych na listingu 18.20.

Listing 18.20. *Sterowanie kontrolą poprawności po stronie klienta*

```
<configuration>
  <appSettings>
    <add key="ClientValidationEnabled" value="true"/>
```

```
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
  </appSettings>
...
```

Aby kontrola poprawności po stronie klienta działała, oba te ustawienia muszą mieć wartość `true`. Podczas generowania projektu MVC 3 Visual Studio tworzy te wpisy i przypisuje im wartość `true`. Możemy na przykład programowo kontrolować te ustawienia w pliku *Global.asax* w sposób pokazany na listingu 18.21.

Listing 18.21. *Programowe sterowanie kontrolą poprawności po stronie klienta*

```
protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    HtmlHelper.ClientValidationEnabled = true;
    HtmlHelper.UnobtrusiveJavaScriptEnabled = true;

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

Możemy również włączać i wyłączać kontrolę poprawności po stronie klienta dla pojedynczych widoków. Powoduje to nadpisanie pokazanych powyżej opcji konfiguracji. Na listingu 18.22 przedstawiony jest sposób sterowania programowego wewnątrz bloku kodu Razor, w którym wyłączamy kontrolę poprawności dla danego widoku.

Listing 18.22. *Sterowanie kontrolą poprawności po stronie klienta dla pojedynczego widoku*

```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
    HtmlHelper.ClientValidationEnabled = false;
}
...
```

Oba te ustawienia muszą mieć wartość `true`, aby działała kontrola poprawności po stronie klienta, co oznacza, że wystarczy jedno z nich ustawić na `false`, aby wyłączyć funkcję. Ponadto należy upewnić się, że są dodane odwołania do trzech bibliotek JavaScript, wyróżnionych na listingu 18.23.

Listing 18.23. *Odwołania do bibliotek JavaScript wymaganych do kontroli poprawności po stronie klienta*

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
        type="text/javascript"></script>

    <script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
        type="text/javascript"></script>

    <script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
        type="text/javascript"></script>
</head>
<body>
```

```
    @RenderBody()
</body>
</html>
```

Możemy dodawać te pliki do każdego widoku, w którym chcemy użyć kontroli poprawności po stronie klienta, ale zwykle prościej jest umieścić te odwołania w pliku układu, tak jak na zamieszczonym listingu.

■ **Ostrzeżenie** Kolejność, w której są umieszczone pliki jQuery, jest znacząca. Jeżeli ją zmienisz, okaże się, że kontrola poprawności po stronie klienta nie działa.

Katalog *scripts* zawiera dwie wersje każdej z bibliotek JavaScript. Wersje, których nazwy kończą się na *min.js*, są wersjami *zminimalizowanymi*, co oznacza, że wszystkie odstępy, komentarze i inne niekrytyczne dane są usunięte, co zmniejsza rozmiar pliku biblioteki. Wersje zminimalizowane mogą być znacznie mniejsze i zwykle są używane w środowiskach produkcyjnych w celu ograniczenia ilości danych pobieranych przez klienta. W czasie tworzenia aplikacji zwykle są wykorzystywane wersje niezminimalizowane, dzięki czemu można debugować (lub po prostu czytać) JavaScript w przypadku wystąpienia problemów.

Użycie CDN dla bibliotek JavaScript

Na listingu 18.22 odwołujemy się do bibliotek jQuery znajdujących się w folderze *~/Scripts* naszej aplikacji. Alternatywnym podejściem jest załadowanie tych plików z sieci Content Delivery Network (CDN) firmy Microsoft. Jest to usługa bezpłatna. Dostępnych jest kilka rozsianych geograficznie serwerów, które obsługują żądania pobrania plików bibliotek JavaScript dla aplikacji MVC, używając serwera znajdującego się najbliżej użytkownika.

Istnieje kilka zalet korzystania z CDN. Otóż czas, jaki zajmuje załadowanie aplikacji przez przeglądarkę użytkownika, może się zmniejszyć, ponieważ serwery CDN są szybsze i znajdują się bliżej użytkownika niż serwery aplikacji. Dodatkowo zalety te łączą się, jeżeli użytkownik pobrał już wymagane pliki w innej aplikacji, która korzysta z plików z tej samej lokalizacji CDN. Kolejną zaletą jest zmniejszenie obciążenia serwera oraz zużycia łącza wykorzystywanego do dostępu do aplikacji. Pliki jQuery są zwykle największymi elementami, jakie musimy dostarczyć do przeglądarki w ramach aplikacji MVC, więc w przypadku pobierania tych plików z serwerów firmy Microsoft zmniejszamy własne koszty. Aby skorzystać z CDN, musimy zmienić atrybut `src` znacznika `script`, by wskazywał na następujące adresy URL:

```
http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1.min.js
http://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js
http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.min.js
```

Dla przykładu znacznik `script` odwołujący się do podstawowej biblioteki jQuery będzie wyglądał następująco:

```
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1.min.js"
type="text/javascript"></script>
```

Serwery CDN obsługują wiele obsługują wiele wersji biblioteki jQuery, więc dla każdego pliku musimy wybrać prawidłowy adres URL. Listę dostępnych plików (oraz ich adresów URL) można zobaczyć na stronie *www.asp.net/ajaxlibrary/cdn.ashx*.

Usługa CDN jest przydatna w aplikacjach udostępnianych w internecie, ale jeżeli projekt jest przeznaczony dla intranetu, to użycie CDN nie ma większego sensu. W takim przypadku szybciej i taniej jest pobrać biblioteki JavaScript z serwera aplikacji.

Użycie kontroli poprawności po stronie klienta

Po włączeniu kontroli poprawności po stronie klienta i upewnieniu się, że w widoku istnieją odwołania do biblioteki jQuery, możemy zacząć weryfikację danych użytkownika. Najprostszym sposobem realizacji tego zadania jest użycie atrybutów metadanych wykorzystywanych wcześniej do kontroli poprawności po stronie serwera, takich jak `Required`, `Range` czy `StringLength`. Na listingu 18.24 zamieszczona jest klasa modelu `Appointment` z dodanymi tymi adnotacjami.

Listing 18.24. *Atrybuty kontroli poprawności zastosowane w klasie modelu Appointment*

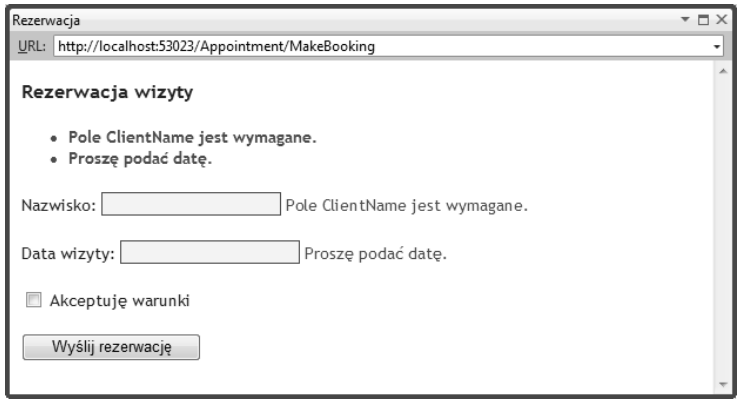
```
public class Appointment {

    [Required]
    [StringLength(10, MinimumLength=3)]
    public string ClientName { get; set; }

    [DataType(DataType.Date)]
    [Required(ErrorMessage="Proszę podać datę")]
    public DateTime Date { get; set; }

    public bool TermsAccepted { get; set; }
}
```

Więcej informacji na temat tych atrybutów można znaleźć w punkcie „Definiowanie zasad poprawności za pomocą metadanych", we wcześniejszej części rozdziału. Mechanizmy kontroli poprawności działające po stronie klienta wyświetlają błędy kontroli poprawności nieco inaczej w przypadku użycia podsumowań oraz komunikatów, więc zmodyfikowaliśmy widok `MakingBooking`, aby korzystał z obu mechanizmów (listing 18.25).

Listing 18.25. *Dodanie podsumowania kontroli poprawności oraz elementów do formularza HTML*

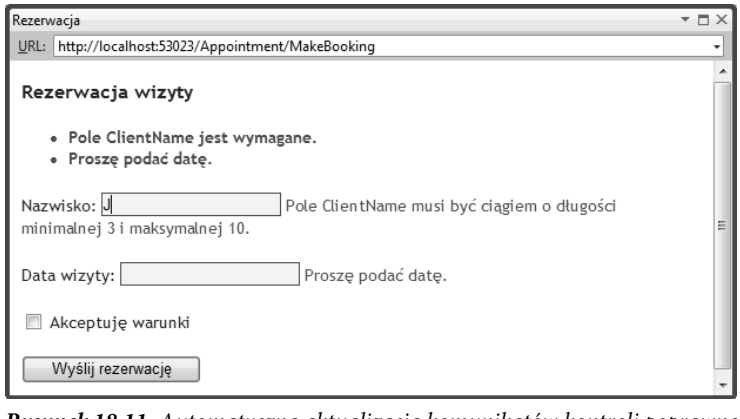
```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    @Html.ValidationSummary()

    <p>Nazwisko: @Html.EditorFor(m => m.ClientName)
        @Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Data wizyty: @Html.EditorFor(m => m.Date)
        @Html.ValidationMessageFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akcepuję warunki
        @Html.ValidationMessageFor(m => m.TermsAccepted)</p>

    <input type="submit" value="Wyślij rezerwację" />

}
```

Gdy otrzymamy żądanie wywołujące metodę akcji `MakeBooking` z naszego kontrolera, generujemy widok *MakeBooking.cshtml* — jest to standardowe działanie. Jednak gdy klikniemy przycisk wysłania danych, zasady kontroli poprawności danych dodane do klasy `Appointment` będą wykorzystywane w przeglądarce z użyciem JavaScript, jak pokazano na rysunku 18.10.



Rysunek 18.10. *Komunikaty o błędach kontroli poprawności po stronie klienta*

Wyświetlane komunikaty wyglądają identycznie jak te generowane po stronie serwera, ale do ich wygenerowania nie była konieczna komunikacja z serwerem. Jeżeli masz serwer i przeglądarkę zainstalowaną na tym samym komputerze, trudno jest zauważyć różnicę w szybkości działania, ale w przypadku połączeń internetowych lub zatłoczonych sieci korporacyjnych różnica może być spora. Istnieje również jeszcze jedna różnica. Kontrola poprawności po stronie klienta jest wykonywana przy pierwszej próbie przesłania danych formularza, ale później jest realizowana po każdym naciśnięciu klawisza lub zmianie fokusu w formularzu HTML. Gdy wpiszemy literę *J* do pola `ClientName`, spowoduje to spełnienie warunku z atrybutu `Required` dodanego do tej właściwości w klasie `Appointment`. Jednak zauważone jest naruszenie zasady z atrybutu `StringLength`, więc obok pola pojawia się następny komunikat, pokazany na rysunku 18.11.

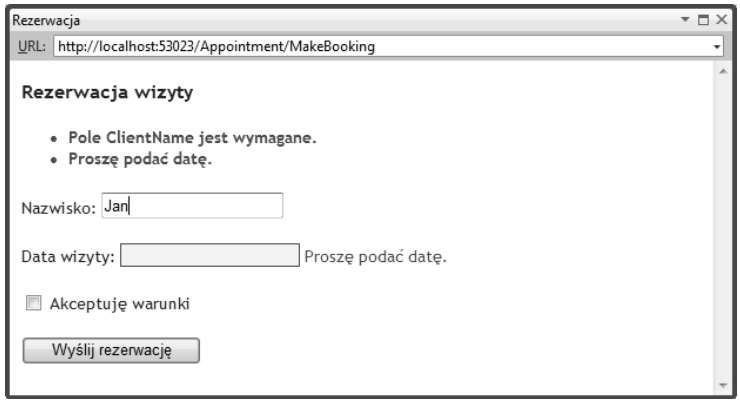


Rysunek 18.11. *Automatyczna aktualizacja komunikatów kontroli poprawności po stronie klienta*

Komunikat o błędzie obok pola się zmienił. Nie wykonaliśmy żadnych specjalnych akcji, aby się to stało; po prostu wpisaliśmy znak do pola. Kontrola poprawności została wykonana ponownie i wyświetlił się komunikat o nowym błędzie.

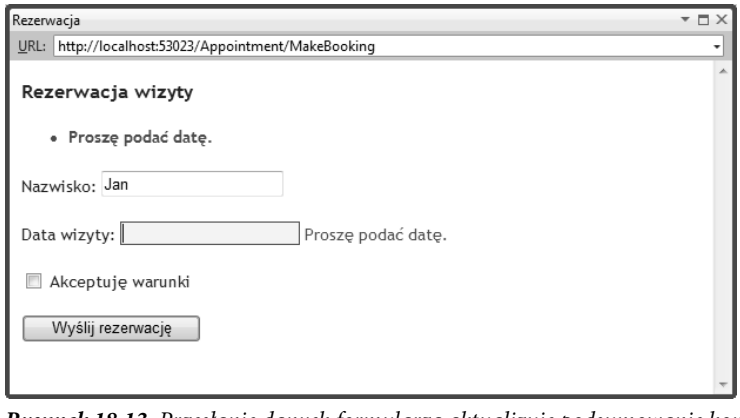
■ **Wskazówka** Zwróć uwagę, że podsumowanie kontroli poprawności utworzone za pomocą metody `Html.ValidationSummary` nie zmienia się — nie jest ono aktualizowane do momentu ponownego kliknięcia przycisku *Wyślij rezerwację*. To właśnie mieliśmy na myśli, mówiąc, że podsumowania działają inaczej niż komunikaty dla właściwości w przypadku użycia kontroli poprawności po stronie klienta.

Kontynuujemy pisanie i gdy dojdziemy do litery *n* w słowie *Jan*, spełnione zostaną obie zasady poprawności zdefiniowane dla właściwości `ClientName` w klasie `Appointment`. Została podana wartość mająca długość od trzech do dziesięciu znaków. W tym momencie komunikaty o błędach przy polu znikają, a wyróżnienie pola jest usuwane, jak pokazano na rysunku 18.12.



Rysunek 18.12. *Automatyczne usuwanie sygnalizacji błędów kontroli poprawności*

Użytkownik otrzymuje natychmiastową informację o błędzie kontroli poprawności, ponadto komunikat o błędzie jest aktualizowany w czasie wprowadzania danych. Trudno to pokazać w postaci serii ekranów, ale jest to znacznie bardziej przyjazny sposób kontroli poprawności danych. Nie tylko zapewnia lepsze działanie interfejsu użytkownika, ale również oznacza, że aplikacja otrzymuje mniej żądań POST, które muszą być przetworzone przez serwer. Gdy mamy satysfakcjonującą nas wartość właściwości `ClientName`, ponownie przesyłamy dane formularza. Jest to jedyny moment, w którym jest aktualizowane podsumowanie, co jest pokazane na rysunku 18.13.



Rysunek 18.13. *Przesłanie danych formularza aktualizuje podsumowanie kontroli poprawności*

Rzadsza aktualizacja podsumowań powoduje, że są one mniej atrakcyjne niż komunikaty przy polach, szczególnie gdy dla pola jest zdefiniowanych kilka zasad poprawności. Nie oznacza to, że nie powinniśmy używać podsumowań, ale należy to robić rozważnie, biorąc pod uwagę efekt, jaki zobaczą użytkownicy.

Jak działa kontrola poprawności po stronie klienta?

Jedną z zalet użycia funkcji kontroli poprawności po stronie klienta w bibliotece MVC jest brak konieczności tworzenia kodu JavaScript. Zamiast tego zasady poprawności są definiowane za pomocą atrybutów HTML.

Poniżej zamieszczony jest kod HTML wygenerowany przez metodę pomocniczą `Html.EditorFor` dla właściwości `ClientName` przy wyłączonej kontroli poprawności po stronie klienta.

```
<input class="text-box single-line" id="ClientName" name="ClientName"
    type="text" value="" />
```

Po włączeniu kontroli poprawności i wygenerowaniu edytora otrzymamy następujący kod:

```
<input class="text-box single-line" data-val="true" data-val-length="The field ClientName must be
↪a string with a minimum length of 3 and a maximum length of 10." data-val-length-max="10"
↪data-val-length-min="3" data-val-required="The ClientName field is required." id="ClientName"
↪name="ClientName" type="text" value="" />
```

Kontrola poprawności po stronie klienta nie generuje bezpośrednio żadnego kodu JavaScript ani danych JSON sterujących procesem kontroli poprawności; podobnie jak w pozostałych częściach biblioteki MVC są wykorzystywane konwencje.

Pierwszym dodanym atrybutem jest `data-val`. Biblioteka jQuery Validation identyfikuje pola wymagające kontroli poprawności przez wyszukiwanie tego atrybutu.

Poszczególne zasady poprawności są specyfikowane za pomocą atrybutu w postaci `data-val-`*<nazwa>*, gdzie *nazwa* jest zasadą do zastosowania. Na przykład atrybut `Required` dodany do klasy modelu powoduje wygenerowanie w HTML-u atrybutu `data-val-required`. Wartość skojarzona z atrybutem stanowi komunikat o błędzie skojarzony z zasadą. Niektóre zasady wymagają dodatkowych atrybutów. Można zauważyć, że definicja długości pola została przekształcona na atrybuty `data-val-length-min` oraz `data-val-length-max`, w których zdefiniowana została minimalna i maksymalna długość napisu.

Interpretacja zasad poprawności jest realizowana przez bibliotekę jQuery Validation, na bazie której są zbudowane mechanizmy kontroli poprawności po stronie klienta.

Jedną z lepszych cech kontroli poprawności po stronie klienta jest użycie tych samych atrybutów do tworzenia zasad poprawności wykorzystywanych po stronie klienta *oraz* po stronie serwera. Oznacza to, że dane w przeglądarkach nieobsługujących JavaScriptu podlegają takiej samej kontroli poprawności jak w przeglądarkach obsługujących ten język, bez wykonywania żadnych dodatkowych operacji.

Modyfikowanie kontroli poprawności na kliencie

Wbudowana obsługa kontroli poprawności na kliencie sprawdza się bardzo dobrze, ale jest ograniczona do sześciu atrybutów zamieszczonych w tabeli 18.3, we wcześniejszej części rozdziału. Biblioteka jQuery Validation obsługuje nieco więcej zasad poprawności, a biblioteka nieprzeszkadzającej kontroli poprawności w MVC pozwala nam skorzystać z nich przy stosunkowo niewielkim nakładzie pracy.

Jawne tworzenie atrybutów kontroli poprawności w HTML

Najbardziej bezpośrednią metodą wykorzystania dodatkowych zasad poprawności jest ręczne wygenerowanie w widoku wymaganych atrybutów, jak pokazano na listingu 18.26.

Listing 18.26. *Ręczne generowanie atrybutów kontroli poprawności w HTML*

```
@model MvcApp.Models.Appointment

@{
    ViewBag.Title = "Rezerwacja";
}

<h4>Rezerwacja wizyty</h4>

@using (Html.BeginForm()) {

    @Html.ValidationSummary()
```

Kontrola poprawności po stronie klienta w MVC a kontrola poprawności w jQuery

Funkcja kontroli poprawności po stronie klienta w bibliotece MVC jest zbudowana w oparciu o bibliotekę jQuery Validation i jeżeli masz takie życzenie, możesz korzystać z niej bezpośrednio, ignorując funkcje MVC. Biblioteka ta jest bardzo elastyczna i bogata w możliwości. Warto się z nią zapoznać, choćby po to, aby zrozumieć, jak można modyfikować funkcje MVC w celu najlepszego możliwego wykorzystania dostępnych opcji kontroli poprawności. Do użycia biblioteki jQuery Validation niezbędna jest pewna znajomość JavaScriptu. Poniżej zamieszczony jest przykład wymaganego skryptu.

```
$(document).ready(function () {

    $('form').validate({
        errorLabelContainer: '#validtionSummary',
        wrapper: 'li',
        rules: {
            ClientName: {
                required: true,
            }
        },
        messages: {
            ClientName: "Proszę podać nazwisko."
        }
    });
});
```

Funkcje kontroli poprawności po stronie klienta dostępne w MVC ukrywają kod JavaScript, a dodatkowo pozwalają realizować identyczne funkcje kontroli poprawności na kliencie i serwerze. Oba podejścia mogą być wykorzystane w aplikacji MVC, choć powinniśmy ostrożnie podchodzić do ich mieszania w jednym widoku, ponieważ mogą pojawić się niechciane interakcje. Więcej informacji na temat biblioteki jQuery Validation można znaleźć na stronie *http://bassistance.de/jquery-plugins*.

```
    <p>Nazwisko:
        @Html.TextBoxFor(m => m.ClientName, new { data_val = "true",
            data_val_email = "Podaj prawidłowy adres e-mail.",
            data_val_required = "Podaj swoje nazwisko."})

        @Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Data wizyty: @Html.EditorFor(m => m.Date)
        @Html.ValidationMessageFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) Akceptuję warunki
        @Html.ValidationMessageFor(m => m.TermsAccepted)</p>

    <input type="submit" value="Wyślij rezerwację" />
}
```

Jeżeli chcemy dodać kolejne atrybuty do znacznika, nie możemy użyć metod pomocniczych widoku szablonowego do wygenerowania właściwości dla edytora, więc zamiast tego zastosowaliśmy metodę `Html.TextBoxFor` w wersji akceptującej typ anonimowy z atrybutami HTML.

■ **Wskazówka** Segmenty nazw atrybutów HTML są rozdzielane łącznikami (-), ale łącznik jest znakiem nielegalnym w nazwach zmiennych C#. Aby obejść ten problem, podajemy nazwy atrybutów z użyciem podkreśleń (_), które są automatycznie konwertowane na łączniki w czasie generowania kodu HTML.

Po wygenerowaniu widoku otrzymamy następujący kod HTML dla właściwości `ClientName`:

```
<input data-val="true" data-val-email="Podaj prawidłowy adres e-mail."
    data-val-required="Podaj swoje nazwisko." id="ClientName"
    name="ClientName" type="text" value="" />
```

Zasada `required` jest również generowana przez atrybut `Required`. Zasada `email` pozwala upewnić się, że wartość prowadzona do pola ma format prawidłowego adresu e-mail. W tabeli 18.5 opisane są zasady kontroli poprawności, z których możemy skorzystać.

Tabela 18.5. *Użyteczne zasady kontroli poprawności dostępne w jQuery*

Zasada poprawności	Atrybut kontroli poprawności	Opis
`Required`	`Required`	Wymaga podania wartości; jest to zasada wykorzystywana przez atrybut `Required`.
`Length`	`StringLength`	Liczba znaków w podanej wartości musi być większa lub równa minimum i (lub) mniejsza lub równa maksimum. Minimalna długość jest definiowana za pomocą atrybutu `data-val-length-min`, a maksymalna długość za pomocą atrybutu `data-val-length-max`. Możemy podać jedynie atrybut `–min` lub `–max`, co spowoduje ograniczenie tylko jednego aspektu długości wartości.
`Range`	`Range`	Wartość musi znajdować się pomiędzy granicami zdefiniowanymi za pomocą atrybutów `data-val-required-min` i `data-val-required-max`. Można podać tylko jeden z tych atrybutów, co spowoduje kontrolę tylko górnego lub dolnego limitu wartości.
`Regex`	`RegularExpression`	Wartość musi pasować do wyrażenia regularnego zdefiniowanego za pomocą atrybutu `data-val-regexp-pattern`.
`Equalto`	`Compare`	Wartość musi być taka sama jak wartość w elemencie wskazanym za pomocą atrybutu `data-val-equalto-other`.
`Email`	-	Wartość musi być prawidłowym adresem e-mail.
`Url`	-	Wartość musi być prawidłowym adresem URL.
`Date`	-	Wartość musi być prawidłową datą.
`Number`	-	Wartość musi być liczbą (może zawierać cyfry po przecinku).
`Digits`	-	Wartość musi zawierać wyłącznie cyfry.
`Creditcard`	-	Wartość musi być prawidłowym numerem karty kredytowej.

Zasady kontroli poprawności, dla których nie istnieją odpowiednie atrybuty C#, pozwalają na sprawdzenie *formatu*, ale nie są w stanie sprawdzić, czy wartość jest naprawdę prawidłowa. Na przykład zasada poprawności `creaditcard` sprawdza, czy wartość jest w prawidłowym formacie dla numeru karty kredytowej oraz czy wartość spełnia schemat kodowania *Luhn*, który jest używany do sprawdzania sumy kontrolnej. Oczywiście nie ma gwarancji, że wartość podana przez użytkownika reprezentuje kartę kredytową faktycznie wydaną przez instytucję finansową.

Podobnie zasady `email` oraz `url` pozwalają sprawdzić, czy format adresu e-mail lub URL jest prawidłowy, ale nie sprawdzają, czy konto e-mail lub strona WWW są dostępne. Jeżeli potrzebujesz bardziej rygorystycznej kontroli, to przydatną funkcją jest *zdalna kontrola poprawności*, którą przedstawimy w dalszej części rozdziału.

Tworzenie atrybutów modelu obsługujących kontrolę poprawności po stronie klienta

Dodawanie atrybutów HTML do naszych elementów widoku jest proste, ale powoduje, że kontrola poprawności jest realizowana wyłącznie po stronie klienta. Możemy temu przeciwdziałać przez wykonanie tej samej kontroli w metodzie akcji lub łączniku, ale lepszą techniką jest utworzenie własnego atrybutu kontroli poprawności, który działa w taki sam sposób jak wbudowane atrybuty i uruchamia kontrolę poprawności realizowaną na serwerze i kliencie. Na listingu 18.27 zamieszczony jest atrybut kontroli poprawności realizujący kontrolę formatu adresu e-mail na serwerze.

Listing 18.27. *Atrybut kontroli poprawności realizujący kontrolę poprawności formatu adresu e-mail na serwerze*

```
public class EmailAddressAttribute : ValidationAttribute {
    private static readonly Regex emailRegex = new Regex(".+@.+\\..+");

    public EmailAddressAttribute() {
        ErrorMessage = "Podaj prawidłowy adres e-mail.";
    }

    public override bool IsValid(object value) {
        return !string.IsNullOrEmpty((string)value) &&
            emailRegex.IsMatch((string)value);
    }
}
```

Jest to podejście takie same jak w przypadku tworzenia atrybutów kontroli poprawności po stronie serwera, które przedstawialiśmy we wcześniejszej części rozdziału. Tworzymy klasę dziedziczącą po `ValidationAttribute` i zmieniamy metodę `IsValid`, w której umieszczamy naszą logikę kontroli poprawności.

■ **Uwaga** Aby zachować prostotę, w przykładzie tym użyliśmy bardzo prostego wyrażenia regularnego do kontroli adresu e-mail. W sieci można łatwo znaleźć bardziej zaawansowane wzorce.

Aby umożliwić działanie kontroli poprawności po stronie klienta, musimy zaimplementować interfejs `IClientValidatable`, zamieszczony na listingu 18.28.

Listing 18.28. *Interfejs IClientValidatable*

```
public interface IClientValidatable {
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}
```

Interfejs ten definiuje jedną metodę, `GetClientValidationRules`, która zwraca listę obiektów `ModelClientValidationRule`. Każdy obiekt `ModelClientValidationRule` opisuje zasadę kontroli poprawności po stronie klienta, jaką należy zastosować, komunikat o błędzie wyświetlany w przypadku złamania tej zasady oraz wszystkie inne parametry wymagane do jej działania. Na listingu 18.29 zamieszczony jest kod ilustrujący sposób dodania kontroli poprawności po stronie klienta w klasie `EmailAddressAttribute` z listingu 18.27.

Listing 18.29. *Dodanie obsługi strony klienckiej w klasie EmailAddressAttribute*

```
public class EmailAddressAttribute : ValidationAttribute, IClientValidatable {
    private static readonly Regex emailRegex = new Regex(".+@.+\\..+");


    public EmailAddressAttribute() {
```

```
        ErrorMessage = "Podaj prawidłowy adres e-mail.";
    }

    public override bool IsValid(object value) {
        return !string.IsNullOrEmpty((string)value) &&
            emailRegex.IsMatch((string)value);
    }

    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context) {

        return new List<ModelClientValidationRule> {
            new ModelClientValidationRule {
                ValidationType = "email",
                ErrorMessage  = this.ErrorMessage
            },
            new ModelClientValidationRule {
                ValidationType = "required",
                ErrorMessage = this.ErrorMessage
            }
        };
    }
}
```

Możemy zwrócić tyle obiektów `ModelClientValidationRule` potrzebnych do zdefiniowania zbioru zasad po stronie klienta, ile potrzeba do wymuszenia naszych zasad poprawności. W przedstawionym przykładzie użyliśmy zasad `email` oraz `required` (ustawiając właściwość `ValidationType` w obiekcie `ModelClientValidationRule`), które korzystają z komunikatu o błędzie zdefiniowanego w atrybucie (za pomocą właściwości `ErrorMessage`). Nasz nowy atrybut możemy przypisać do klasy modelu identycznie jak każdy inny atrybut kontroli poprawności:

```
public class Appointment {

    [EmailAddress]
    public string ClientName { get; set; }
...
```

Gdy tworzony jest edytor dla właściwości `ClientName`, silnik widoku analizuje użyte przez nas metadane, wyszukuje naszą implementację `IClientValidatable` i generuje atrybuty HTML przedstawione w poprzednim punkcie. Przesłane dane są ponownie sprawdzane za pomocą naszej metody `IsValid`. Nasz nowy atrybut jest używany do kontroli poprawności zarówno na serwerze, jak i na kliencie, co jest przyjemniejsze, bezpieczniejsze i bardziej spójne niż jawne generowanie atrybutów HTML.

Tworzenie własnych zasad poprawności na kliencie

Przedstawione w tabeli 18.5 wbudowane zasady kontroli poprawności na kliencie są użyteczne, ale nie wyczerpują wszystkich możliwości. Na szczęście możesz przygotować własne zasady, jeżeli jesteś w stanie napisać kilka wierszy kodu JavaScript.

Obsługa kontroli poprawności na kliencie jest ograniczona w bibliotece MVC do zasad dostępnych w bazowej bibliotece jQuery. Sprowadza się to do tego, że możemy dostosowywać istniejące zasady na wiele sposobów, ale jeżeli chcemy utworzyć coś bardziej skomplikowanego, musimy zostawić wbudowaną obsługę kontroli poprawności z MVC i operować bezpośrednio na jQuery. Mimo że mamy narzucone ograniczenie, możemy tworzyć nowe, użyteczne mechanizmy.

Na przykład funkcje kontroli poprawności bazujące na kliencie nie obsługują pól wyboru w sposób prawidłowy, podobnie jak przedstawione wcześniej atrybuty kontroli poprawności na serwerze. Możemy utworzyć nową, kliencką zasadę poprawności, która będzie korzystała z zasady jQuery `rule` dla pól wyboru, jak pokazano na listingu 18.30.

Listing 18.30. *Tworzenie własnego odwzorowania pomiędzy funkcjami kontroli poprawności w MVC i jQuery*

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.5.1.min.js"
type="text/javascript"></script>
<script src="http://ajax.aspnetcdn.com/ajax/jquery.validate/1.7/jquery.validate.min.js"
type="text/javascript"></script>
<script src="http://ajax.aspnetcdn.com/ajax/mvc/3.0/jquery.validate.unobtrusive.js"
type="text/javascript"></script>

<script type="text/javascript">
    jQuery.validator.unobtrusive.adapters.add("checkboxtrue", function (options) {
        if (options.element.tagName.toUpperCase() == "INPUT" &&
            options.element.type.toUpperCase() == "CHECKBOX") {

            options.rules["required"] = true;
            if (options.message) {
                options.messages["required"] = options.message;
            }
        }
    });

</script>

</head>

<body>
    @RenderBody()
</body>
</html>
```

Utworzyliśmy tu nową zasadę o nazwie `checkboxtrue`, która za pomocą zasady `required` z jQuery Validation ma za zadanie sprawdzić, czy pole wyboru jest zaznaczone. Skrypt ten dodaliśmy do pliku układu w projekcie (*_Layout.cshtml*), dzięki czemu jest on dostępny we wszystkich widokach.

■ **Uwaga** Dodawanie nowych zasad kontroli poprawności jest zaawansowanym procesem, który wymaga dobrej znajomości biblioteki jQuery Validation oraz obsługi kontroli poprawności po stronie klienta z biblioteki MVC. Nie będziemy wyjaśniać, jak działa skrypt z listingu 18.29, ale jeżeli chcesz nauczyć się dodawać nowe zasady poprawności, to na początek warto zapoznać się z kodem źródłowym z pliku *jQuery.validate.unobtrusive.js*.

Po utworzeniu klienckiej zasady poprawności możemy utworzyć korzystający z niej atrybut. We wcześniejszej części rozdziału pokazaliśmy, jak utworzyć serwerowy atrybut kontroli poprawności, który pozwala upewnić się, że pole wyboru jest zaznaczone. Na listingu 18.29 rozszerzyliśmy tę klasę atrybutu o obsługę strony klienckiej, wykorzystując zasadę `checkboxtrue` z listingu 18.31.

Listing 18.31. *Dodanie kontroli poprawności po stronie klienckiej w klasie MustBeTrueAttribute*

```
public class MustBeTrueAttribute : ValidationAttribute, IClientValidatable {

    public override bool IsValid(object value) {
        return value is bool && (bool)value;
```

```
    }
    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context) {

        return new ModelClientValidationRule[] {
            new ModelClientValidationRule {
                ValidationType = "checkboxtrue",
                ErrorMessage = this.ErrorMessage
            }};
    }
}
```

Teraz możemy korzystać z atrybutu `MustBeTrue` dla właściwości typu `bool` w klasach modelu, dzięki czemu upewnimy się, że użytkownik zaznaczy pole wyboru przed przesłaniem danych na serwer.

Wykonywanie zdalnej kontroli poprawności

Ostatnią funkcją kontroli poprawności, jaką zajmiemy się w tym rozdziale, jest *zdalna kontrola poprawności*. Jest to technika kontroli poprawności po stronie klienta, która wywołuje na serwerze metodę akcji wykonującą kontrolę poprawności.

Częstym przykładem stosowania zdalnej kontroli poprawności jest sprawdzanie w czasie rejestracji, czy nazwa użytkownika jest dostępna. Po przesłaniu danych przez użytkownika wykonywana jest kontrola poprawności na serwerze. W procesie tym do serwera wysyłane są żądania Ajax, za pomocą których sprawdzamy, czy podana przez użytkownika nazwa jest dostępna. Jeżeli nazwa użytkownika jest zajęta, wyświetlany jest komunikatu o błędzie kontroli poprawności, a użytkownik może podać inną wartość.

Może to wyglądać jak zwykła kontrola poprawności po stronie klienta, ale podejście to ma kilka zalet. Po pierwsze, tylko część właściwości trzeba kontrolować zdalnie; w przypadku pozostałych możemy korzystać z zalet kontroli na kliencie. Po drugie, żądania są względnie niewielkie i skupione na kontroli poprawności, a nie przetwarzaniu całego modelu obiektu. Oznacza to, że możemy minimalizować spadek wydajności, jaki będą wywoływać te żądania.

Trzecią różnicą jest to, że zdalna kontrola poprawności jest realizowana w tle. Użytkownik nie musi klikać przycisku *Wyślij* i czekać na wygenerowanie i załadowanie nowego widoku. Dzięki temu interfejs użytkownika szybciej reaguje, szczególnie gdy połączenie pomiędzy przeglądarką a serwerem jest powolne.

Trzeba pamiętać, że zdalna kontrola poprawności jest kompromisem; pozwala zachować równowagę pomiędzy kontrolą poprawności na kliencie i serwerze, ale wymaga wysłania żądań do serwera aplikacji i nie jest tak szybka jak kontrola poprawności na serwerze.

■ **Wskazówka** Obsługę Ajax oraz JSON w bibliotece MVC przedstawiamy w rozdziale 19.

Pierwszym krokiem w kierunku użycia zdalnej kontroli poprawności jest utworzenie metody akcji, która będzie sprawdzała poprawność jednej z właściwości modelu. W naszej klasie modelu `Appointment` będziemy kontrolować poprawność właściwości `Date`, upewniając się, że podana data jest datą przyszłą (jest to początkowa zasada poprawności, jaką zdefiniowaliśmy na początku tego rozdziału). Na listingu 18.32 pokazana jest metoda `ValidateDate`, którą dodaliśmy do klasy `AppointmentController`.

Listing 18.32. *Dodanie metody kontroli poprawności do kontrolera*

```
public class AppointmentController : Controller {
    private IAppointmentRepository repository;

    public AppointmentController(IAppointmentRepository repo) {
        repository = repo;
```

```
    }

    public ViewResult MakeBooking() {
        return View(new Appointment { Date = DateTime.Now });
    }

    public JsonResult ValidateDate(string Date) {

        DateTime parsedDate;

        if (!DateTime.TryParse(Date, out parsedDate)) {
            return Json("Proszę podać prawidłową datę (mm/dd/rrrr).",
                JsonRequestBehavior.AllowGet);
        } else if (DateTime.Now > parsedDate) {
            return Json("Prosze podać przyszłą datę.", JsonRequestBehavior.AllowGet);
        } else {
            return Json(true, JsonRequestBehavior.AllowGet);
        }
    }

    [HttpPost]
    public ViewResult MakeBooking(Appointment appt) {

        if (ModelState.IsValid) {
            repository.SaveAppointment(appt);
            return View("Completed", appt);
        } else {
            return View();
        }
    }
}
```

Metody akcji obsługujące kontrolę poprawności muszą zwracać wartość typu `JsonResult`, a nazwa parametru metody musi być taka sama jak kontrolowane pole. W naszym przypadku jest to `Date`. Musimy upewnić się, że możemy uzyskać obiekt `DateTime` z przesłanej przez użytkownika wartości, a jeżeli się to uda, sprawdzamy, czy data jest datą przyszłą.

■ **Wskazówka** Moglibyśmy skorzystać z dołączania modelu, definiując parametr tej metody akcji jako `DateTime`, ale w takim przypadku nasza metoda kontroli poprawności nie będzie wywołana, jeżeli użytkownik poda nieprawidłową wartość, jak na przykład *jabłko*. Dzieje się tak, ponieważ łącznik modelu nie będzie w stanie utworzyć obiektu `DateTime` z podanej wartości i zgłosi wyjątek. Funkcja zdalnej kontroli poprawności nie ma możliwości przedstawienia wyjątku, więc zakończy działanie. W takim przypadku pole nie zostanie wyróżnione, więc użytkownik będzie miał wrażenie, że wprowadzona wartość jest prawidłowa. Jako naczelną zasadę w zdalnej kontroli poprawności należy przyjąć, że metody akcji oczekują parametru typu `string` i jawnie wykonują konwersje typów, parsowanie lub dołączanie modelu.

Wyniki kontroli poprawności wyrażamy za pomocą metody `Json`, która tworzy wynik JSON analizowany przez skrypt zdalnej kontroli poprawności działający na serwerze. Jeżeli przetwarzana wartość spełnia nasze wymagania, przekazujemy `true` jako parametr do metody `Json` w następujący sposób:

```
return Json(true, JsonRequestBehavior.AllowGet);
```

Jeżeli wartość nie spełnia naszych oczekiwań, przekazujemy komunikat kontroli poprawności w następujący sposób:

```
return Json("Proszę podać przyszłą datę.", JsonRequestBehavior.AllowGet);
```

W obu przypadkach musimy również przekazać wartość `JsonRequestBehavior.AllowGet` jako parametr. Jest to wymagane, ponieważ biblioteka MVC domyślnie nie pozwala na żądania GET zwracające JSON, więc musimy zmienić to ustawienie. Bez tego parametru żądania kontroli poprawności zostaną bez żadnego komunikatu odrzucone i informacje o błędach nie będą wyświetlane na kliencie.

■ **Ostrzeżenie** Metoda akcji kontroli poprawności będzie wywołana przy pierwszym przesłaniu formularza, a następnie za każdym razem, gdy użytkownik zmieni dane. Każde naciśnięcie klawisza spowoduje wysłanie żądania do serwera. W przypadku niektórych aplikacji może to spowodować znaczną liczbę żądań, więc trzeba o tym pamiętać przy określaniu mocy obliczeniowej serwera oraz przepustowości łącza wymaganego dla aplikacji. Dodatkowo można zdecydować, aby *nie* korzystać ze zdalnej kontroli poprawności właściwości, które są zbyt kosztowne (na przykład jeżeli musimy odpytać powolną usługę sieciową w celu sprawdzenia, czy nazwa jest unikatowa).

Podsumowanie

W rozdziale tym przedstawiliśmy szeroką gamę technik pozwalających na realizowanie kontroli poprawności modelu, dzięki którym możemy upewnić się, że podane przez użytkownika dane spełniają ograniczenia, jakie zdefiniowaliśmy dla naszego modelu danych.

Kontrola poprawności jest ważnym zagadnieniem — utworzenie prawidłowego mechanizmu weryfikacji danych w aplikacji jest niezbędne, aby oszczędzić użytkownikom frustracji przy pracy z aplikacją. Równie ważne jest to, że zachowujemy spójność danych modelu i w naszym systemie nie zapisujemy danych o niskiej jakości.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

WYKORZYSTAJ NOWOCZESNE WZORCE W TWOICH PROJEKTACH!

ASP.NET MVC 3 to kolejny krok w ewolucji platformy ASP.NET firmy Microsoft. W tej bibliotece programowania witryn WWW połączono efektywność i schludność architektury model-widok-kontroler (MVC), najnowsze pomysły i techniki programowania zwinnego oraz najlepsze części istniejącej platformy ASP.NET. Jest ona więc alternatywą dla tradycyjnych projektów ASP.NET Web Forms i ma nad tą platformą znaczną przewagę. Szkielet ASP.NET MVC 3 pozwoli Ci błyskawicznie stworzyć skalowalne, łatwe w utrzymaniu i rozwijaniu aplikacje internetowe, a jego wykorzystanie da Ci niepowtarzalną szansę dziecinnie łatwego stosowania testów jednostkowych i podejścia TDD (ang. Test Driven Development). Co jeszcze możesz zyskać dzięki ASP.NET MVC 3?

O tym przekonasz się w trakcie lektury tej wyjątkowej książki. Jej autorzy przyjęli słuszną zasadę, że dobry przykład mówi więcej niż kilka stron suchej teorii. Zobaczysz więc, jak wzorzec MVC sprawdza się w praktyce, jakie są jego zalety i wady oraz jak unikać typowych pułapek. Nauczysz się korzystać z filtrów, konfigurować kontrolery oraz projektować wydajny model. W dzisiejszych czasach bezpieczeństwo aplikacji jest stawiane na pierwszym miejscu, dlatego szczególną uwagę powinieneś zwrócić na rozdział poświęcony zabezpieczaniu stworzonego przez Ciebie rozwiązania. Ta książka to kompletna pozycja dla każdego programisty ASP.NET.
Sięgnij po nią i przekonaj się, że:

- wzorzec MVC ułatwia pracę
- jego zastosowanie korzystnie wpływa na jakość kodu
- bezpieczeństwo aplikacji jest najważniejsze
- implementacja wzorca MVC nie musi być trudna!

Apress®

Nr katalogowy: 11679

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

helion.pl
księgarnia internetowa

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena 99,00 zł
ISBN 978-83-246-4822-1
9 788324 648221

Informatyka w najlepszym wydaniu