GWT w Praktyce

ROBERT COOPER
CHARLES COLLINS

POWERnet

MANNING
Greenwich
(74° w. long.)

GWT in Practice
ISBN-13: 978-1933988290
ISBN-10: 1-933988-29-0

GWT w praktyce
ISBN: 978-83-924603-3-6

POWERnet Wydawnictwo Power Net
ul. Sekutowicza 5/3, 20-152 Lublin
biuro@powernet.pl
www.powernet.pl

Tłumaczenie: Marcin Leszczyński
Korekta merytoryczna: Krzysztof Antoń
Korekta: Anna Antoń
Skład, przygotowanie do druku: K3Creativ
Druk i oprawa: Gaudium, Lublin

Printed in Poland

spis treści

4 Podstawowa struktura aplikacji

Zawartość rozdziału

- Tworzenie kompozytowych komponentów widoku
- Wiązanie danych dla modelu i widoku
- Związek kontrolera z warstwą usług RPC
- Używanie JPA w modelu
- Kończenie pełnej aplikacji GWT, klienta i serwera

Musisz oduczyć się tego, czego się nauczyłeś.

– Yoda (*Imperium kontratakuje*)

Już od wielu lat tworzysz aplikacje internetowe. Z biegiem czasu błędy przy programowaniu stają się oczywiste, ale jeśli przypominasz autorów tej książki, to radzenie sobie z nimi stało się Twoją drugą naturą. Wiesz już, że GWT stanowi zupełnie nową jakość w programowaniu WWW, a przykład kalkulatora z drugiego rozdziału pokazał, że programowanie GWT znacznie bardziej przypomina tworzenie tradycyjnych aplikacji biurowych. Teraz możesz wcisnąć przycisk „przewiń do tyłu" w Twoim doświadczeniu oraz wiedzy i spojrzeć w nowy sposób na rozwijanie aplikacji WWW. W drugiej części książki wyjdziemy poza wstępny

materiał poprzednich rozdziałów i przyjrzymy się bardziej ukierunkowanym na cel technikom GWT.

Podczas pracy z GWT oczywiście już nie budujemy nawigacji ani stron tak, jak dawniej. Nawet bardziej ukierunkowane na moduły frameworki WWW, takie jak JSF, nadal zasadniczo bazują na stronie, a warstwy modelu, widoku i kontrolera działają wewnątrz serwera aplikacji. Musisz teraz popatrzeć na swoją aplikację, widząc grupy widżetów tworzących widok, a także kontroler, który organizuje działania, i inteligentną warstwę modelu.

W tym rozdziale zajmiemy się podstawowym przykładem aplikacji internetowej, który zapewne pisałeś wcześniej wielokrotnie: rejestracją użytkownika. Przeanalizujemy całą strukturę tego przykładu i zobaczymy, jak te części wiążą się w aplikacjach GWT. Zaczniemy od utworzenia warstwy modelu, która jest inteligentniejsza od większości modeli, których prawdopodobnie używałeś w innych aplikacjach WWW. Następnie przedstawimy, jak skonstruować komponenty widoku i połączyć je z modelem. W końcu, przyjrzymy się kontrolerowi i w jaki sposób używać warstwy kontrolera do interakcji z usługami na serwerze, a także jak wykorzystywać interfejs Java Persistence API (JPA) do przechowywania elementów w bazie danych.

4.1. Tworzenie modelu

Warstwa modelu w aplikacji GWT musi być trochę mądrzejsza od modeli z wielu tradycyjnych aplikacji WWW. Musi powiadamiać warstwę widoku o zmianach, jak również odbierać uaktualnienia swojej własnej struktury. W biurowym programowaniu w Javie uzyskiwane jest to poprzez wykorzystanie klas `PropertyChangeEvent` i `PropertyChangeListener`, które wiążą dane z modelu z komponentami widoku. Warstwa widoku „nasłuchuje" zmian w modelu i stosownie się uaktualnia, używając wzorca obserwatora. Można tak zrobić również w GWT. Widzieliśmy już niewielki przykład tego w rozdziale drugim w modelu kalkulatora, a teraz bliżej przyjrzymy się temu rozwiązaniu, używając bardziej formalnego i reprezentacyjnego podejścia, które pokazuje, co będziesz musiał robić setki razy.

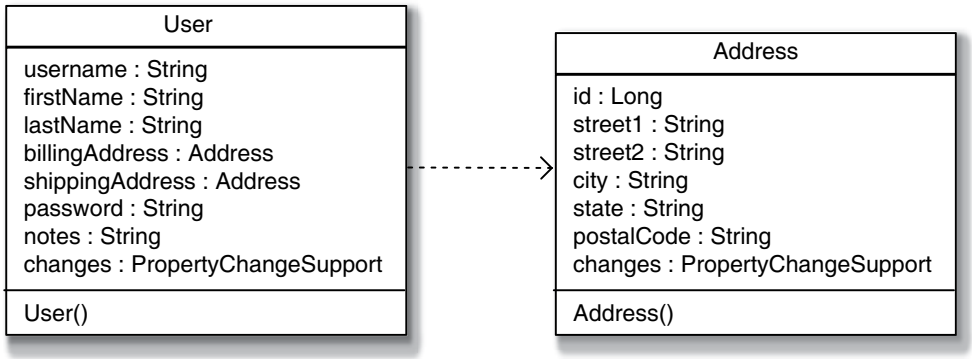
Rysunek 4.1 pokazuje podstawową klasę `User`, z którą będziemy pracować. Jest to prosta klasa zawierająca nazwę użytkownika i inne zwykłe pola, jak również dwa obiekty `Address` (jeden adres do wysyłki i drugi adres do faktury). Aby rozpocząć, musimy umożliwić im wiązanie danych.

PROBLEM

Musimy umożliwić wiązanie danych z elementami interfejsu użytkownika w naszym modelu.

ROZWIĄZANIE

Wiązanie danych pomiędzy modelem i widokiem opiera się na wywoływaniu zdarzeń w modelu, które powiadamiają widok o zmianach i odwrotnie. W przykładzie z kalkulatorem z rozdziału drugiego zbudowaliśmy od podstaw system zdarzeń. Zrobiliśmy tak, aby wszystko pokazać wprost i nie dokładać



Rysunek 4.1. Klasy `User` i `Address` służące jako model. Zwróćmy uwagę na atrybut `PropertyChangeSupport`, który będziemy używać do powiadamiania widoku o zmianach komponentów modelu.

dodatkowych zagadnień ani zależności w naszym pierwszym przykładzie. Jednak taka nieautomatyczna metoda jest nieco niewygodna.

Tutaj natomiast użyjemy klasy `java.beans.PropertyChangeSupport`. Nie jest to jeszcze klasa dostarczona przez GWT, ale jest dostępna jako część projektu GWTx (http://code.google.com/p/gwtx/), który do podstawowej Javy dodaje klasy dostarczone przez GWT. Listing 4.1 pokazuje obiekt `User` wykorzystujący klasę `PropertyChangeSupport`.

Listing 4.1. Używanie klasy `PropertyChangeSupport` w obiekcie `User`.

```
public class User implements IsSerializable {

   private String username;
   private String firstName;
   private String lastName;
   private Address billingAddress = new Address();
   private Address shippingAddress = new Address();
   private String password;
   private String notes;
   private transient PropertyChangeSupport changes =
      new PropertyChangeSupport(this);

   public User() {
      super();
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      String old = this.firstName;
      this.firstName = firstName;
      changes.firePropertyChange(
"firstName", old, firstName);
   }
```

❶ **Zaimplementuj interfejs IsSerializable**

❷ **Utwórz PropertyChangeSupport**

❸ **Dodaj PropertyChangeSupport do uruchomienia zmian**

```
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        String old = lastName;
        this.lastName = lastName;
        changes.firePropertyChange("lastName", old, lastName );
    }

    public Address getBillingAddress() {
        return billingAddress;
    }

    public void setBillingAddress(Address billingAddress) {
        Address old = this.billingAddress;
        this.billingAddress = billingAddress;
        changes.firePropertyChange(
"billingAddress", old, billingAddress);
    }

    // Pozostała część metod zwracających i ustawiających została dla
       skrótu usunięta.

    public void addPropertyChangeListener(
PropertyChangeListener l) {
        changes.addPropertyChangeListener(l);
    }

    public void addPropertyChangeListener(
        String propertyName,
PropertyChangeListener l) {
        changes.addPropertyChangeListener(
propertyName, l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l) {
        changes.removePropertyChangeListener(l);
    }

    public void removePropertyChangeListener(
        String propertyName, PropertyChangeListener l) {
        changes.removePropertyChangeListener(propertyName, l);
    }
}
```

❹ **Dodaj globalny odbiornik zmian**

❺ **Dodaj odbiornik konkretnych właściwości**

Dzięki temu uzyskujemy obiekt modelu potrafiący powiadamiać widok o zmianach i udostępniający metody do dołączania odbiorników.

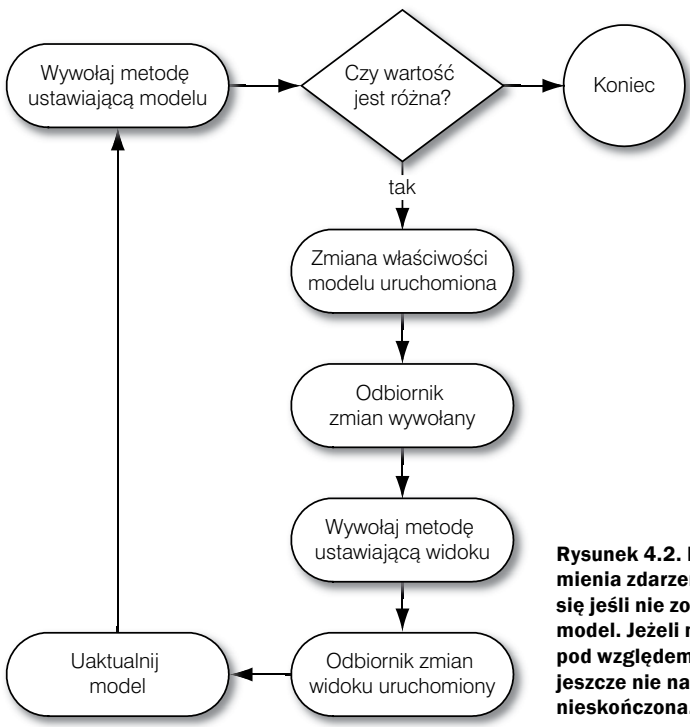
OMÓWIENIE

Może jest to znacznie więcej kodu, niż komponenty modelu tradycyjnej aplikacji WWW mogły nas do tego przyzwyczaić, ale w większości jest to szablonowy kod. Najpierw musimy zaimplementować `IsSerializable` lub `Serializable` ❶ (interfejsy GWT informujące kompilator, że dana klasa musi zostać zserializowana), ponieważ chcemy, aby ten obiekt poruszał się w obie strony pomiędzy klientem i serwerem. Tym razem jednak mamy właściwość, która nie poddaje się

serializacji: egzemplarz changes klasy `PropertyChangeSupport` jest oznaczony jako przejściowy ❷ i nie jest częścią zserializowanej wersji wysyłanej tam i z powrotem. Jest tworzony za każdym razem wraz z obiektem.

Kiedy już mamy klasę `PropertyChangeSupport`, musimy nakazać metodom ustawiającym, aby uruchomiły zdarzenia zmian. Sekwencja jest powtarzalna, ale bardzo ważna. Najpierw zapisujemy bieżącą wartość, a następnie ustawiamy nową wartość – i dopiero, gdy nowa wartość jest ustawiona, wtedy uruchamiamy zdarzenie zmian ❸. Jest to niezmiernie ważne, aby zdarzenia zmian były uruchamiane dopiero po ustawieniu wartości egzemplarza – a nie na początku metody ustawiającej. Powodem tego jest to, co się dzieje w klasie `PropertyChangeSupport`.

`PropertyChangeSupport` (PCS) dostarcza kolekcji i uruchamia zdarzenia zmian, tak jak interfejsy użyte w poprzednim przykładzie kalkulatora. Sprawdza każde wywołanie metody `firePropertyChange()`, aby upewnić się przed uruchomieniem zdarzenia, że stare i nowe wartości są różne. Rysunek 4.2 przedstawia kolejność wykonania dla uruchamianego zdarzenia zmian, które uaktualnia dany element dwustronnym wiązaniem.



Rysunek 4.2. Przebieg procesu uruchomienia zdarzeń zmiany właściwości zapętli się jeśli nie zostanie sprawdzony przez model. Jeżeli model nie sprawdza zmiany pod względem równości lub jeśli zmiana jeszcze nie nastąpiła, tworzy się pętla nieskończona.

Jeśli wywołalibyśmy metodę `firePropertyChange()` na początku metody ustawiającej, kiedy osiągnięty został ostatni krok tej sekwencji, wówczas bieżąca wartość egzemplarza obiektu modelu byłaby nadal taka sama i cały cykl zdarzeń popadłby w nieskończoną pętlę!

Na koniec, musimy dostarczyć metody umożliwiające dodanie odbiorników do obiektu modelu. Są dwa rodzaje odbiorników obsługiwanych przez klasę `PropertyChangeSupport`. Odbiorniki globalne ❹ odbierają zdarzenie za każdym razem, gdy powstanie zmiana. Odbiornik sprawdza wówczas wartość zwróconą przez wywołanie metody `getPropertyName()` z egzemplarza `PropertyChangeEvent`, aby zdecydować, co dalej robić. Drugim rodzajem odbiorników są odbiorniki konkretnych właściwości ❺. Są one uruchamiane tylko wówczas, gdy zmieniają się określone właściwości. I to one są zazwyczaj najbardziej przydatne. Jeśli jednak mamy skomplikowany model lub taki, który wymaga dużej ilości translacji pomiędzy danymi modelu i danymi potrzebnymi do poprawnego generowania elementów widoku, łatwiejsze może być po prostu nasłuchiwanie jakichkolwiek zmian i resetowanie całego komponentu widoku.

Stworzyliśmy już warstwę modelu naszej aplikacji. Sprawiliśmy, że może ona powiadamiać odbiorniki o zmianach, co może zostać wykorzystane do wiązania danych w warstwie widoku. Kolejną częścią aplikacji rejestrowania użytkownika, jaką musimy zbudować, jest sama warstwa widoku. Widok będzie zarówno nasłuchiwał zmian pochodzących z modelu, jak i uruchamiał swoje własne zmiany, aby uaktualnić model zgodnie z danymi wprowadzanymi przez użytkownika.

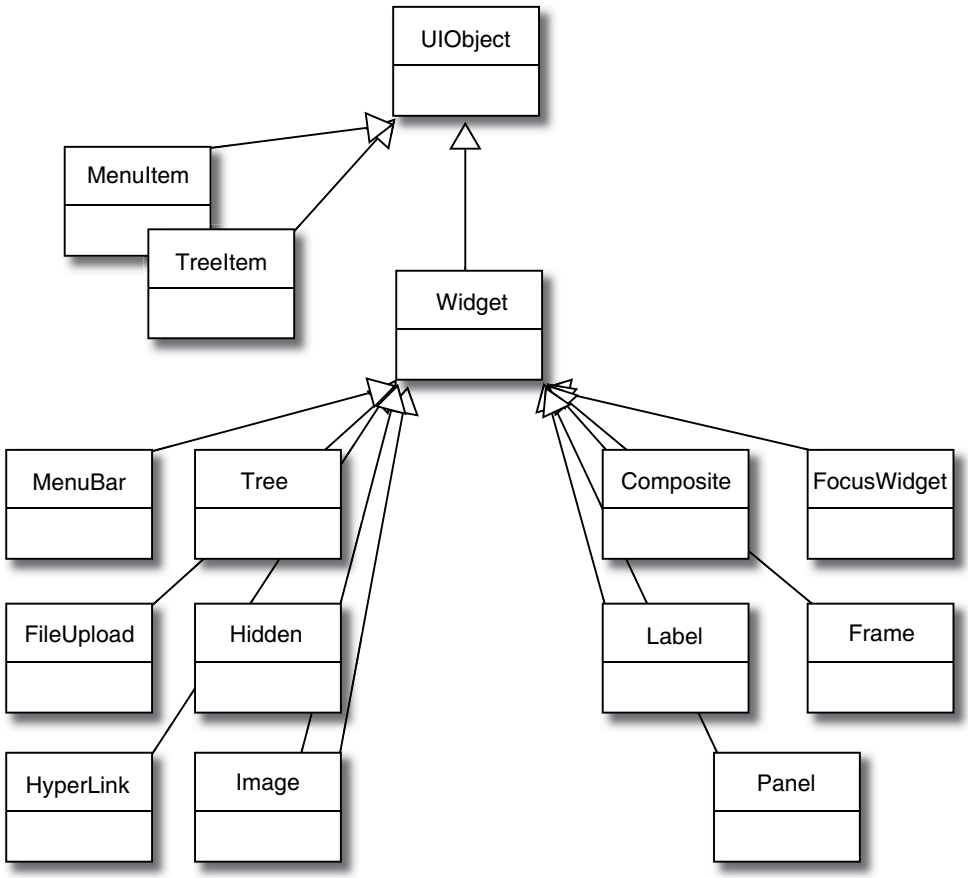
4.2. Tworzenie komponentów widoku

Warstwa widoku w aplikacji GWT – w przeciwieństwie do technologii opartych na szablonie czy stronie – składa się z klas Javy. W efekcie najlepszym sposobem stworzenia interfejsu użytkownika jest zbudowanie komponentu (lub komponentów) dla każdego elementu widoku. Mogą składać się one z kilku różnych rodzajów elementów widoku, tak jak w przypadku tworzenia szablonów stron do edytowania typu obiektów podczas używania takich tradycyjnych systemów, jak Struts Tiles.

Większość dobrze zaprojektowanych bibliotek API pochodzi od kilku podstawowych klas, które obejmują popularne funkcje. GWT nie różni się pod tym względem. Pakiet `com.google.gwt.user.client.ui` zawiera takie podstawowe elementy. Zazwyczaj nie są utworzone ich egzemplarze bezpośrednio, lecz stanowią one trzon interfejsu API. Pakiet otwiera bazowa klasa `UIObject`, która następnie rozdziela się na różne hierarchie komponentów tworzących pozostałą część biblioteki. Klasa `UIObject`, jak wskazuje jej nazwa, jest tym, od czego zaczynamy, kiedy budujemy komponenty widoku.

Rysunek 4.3 prezentuje uproszczony schemat klas z górnej części interfejsu API. Możemy zobaczyć `UIObject`, `Widget` i pozostałe pochodne elementy wykorzystywane do tworzenia komponentów widoku.

Interfejs API rozdziela się z klasy `UIObject` na trzech bezpośrednich potomków: podzieloną dalej klasę `Widget` i dwa pojedyncze odgałęzienia: `MenuItem` oraz `TreeItem`. Podklasy klasy `Widget` dzielą się na dwa różne zbiory klas: tych, które nie są dalej dzielone na podklasy (lewa strona na rysunku 4.3) i tych, w których można wyróżnić następne podklasy (prawa strona na rysunku 4.3).



Rysunek 4.3. Uogólniona hierarchia klas z pakietu `client.ui`. Zauważmy, że prawie wszystko to `Widget`, z wyjątkiem `MenuItem` i `TreeItem`, które mają ograniczone zastosowanie wewnątrz innych widżetów.

Są to podstawowe klasy używane do tworzenia aplikacji GWT. Przy konstruowaniu warstwy widoku istnieją dwa ogólne sposoby tworzenia elementów interfejsu UI: rozszerzanie podstawowego widżetu GWT lub tworzenie kompozytu. Przyjrzymy się obu tym metodom po kolei.

4.2.1. Rozszerzanie widżetów

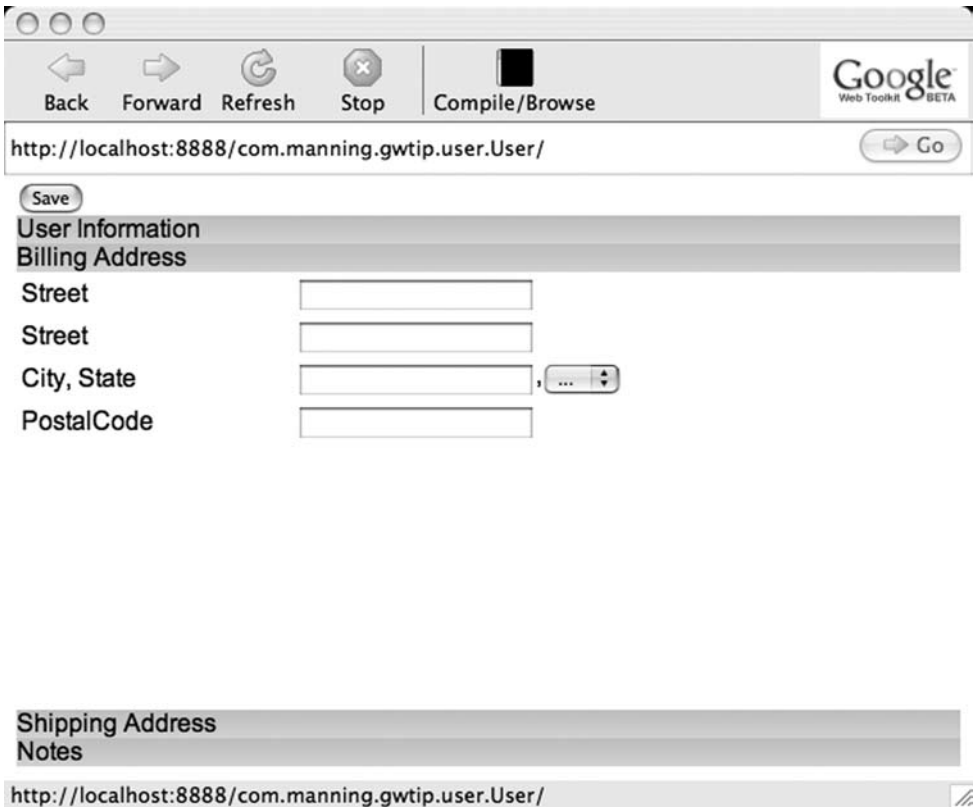
Zadeklarowanie klasy widoku jako rozszerzającej podstawowy widżet jest łatwym rozwiązaniem na początek. Możemy następnie dokonać własnej konfiguracji konstruktora i dodać metody związane z naszą implementacją, tak jak zrobiliśmy to w kalkulatorze z rozdziału drugiego. Zaczniemy tutaj w ten sam sposób, tworząc klasę Panel, która dostarcza widok edycji obiektu `Address`.

PROBLEM

Musimy skonstruować konkretny widok elementu modelu, który będzie wykorzystany wewnątrz naszego modułu.

ROZWIĄZANIE

Rozszerzenie podstawowego widżetu GWT jest z pewnością najprostszym rozwiązaniem na początek. Użyjemy tego sposobu do utworzenia klasy `AddressEdit`. Rysunek 4.4 przedstawia fragment edycji adresu większego komponentu służącego do edycji danych użytkownika, który ukończymy w tym rozdziale: widżet `UserEdit`.



Rysunek 4.4. Widżet `AddressEdit` przedstawiający edycję właściwości `billingAddress` zagnieżdżonej w `StackPanel`. `StackPanel` pokazuje pozostałe elementy większego komponentu `UserEdit`.

Klasa `AddressEdit` rozszerza tutaj po prostu widżet `FlexTable`, umożliwiając nam ułożenie dość standardowego formularza adresu. Listing 4.2 pokazuje, jak tworzymy ten formularz w konstruktorze klasy `AddressEdit`.

Listing 4.2. AddressEdit.java.

```
public class AddressEdit extends FlexTable {
   private Address address;
   private TextBox street1 = new TextBox();
   private TextBox street2 = new TextBox();
   private TextBox city = new TextBox();
   private ListBox state = new ListBox();
   private TextBox postalCode = new TextBox();

   public AddressEdit(final Address address) {
      super();

      state.addItem("...");
      state.addItem("AL");
      state.addItem("AK");
      // pozostała część pominięta.

      this.setStyleName("user-AddressEdit");
      this.address = address;
      this.setWidget(0,0, new Label("Street"));
      this.setWidget(0,1, street1);
      this.setWidget(1,0, new Label("Street"));
      this.setWidget(1,1, street2);
      this.setWidget(2,0, new Label("City, State"));
      HorizontalPanel hp = new HorizontalPanel();
      hp.add(city);
      hp.add(new Label(","));
      hp.add(state);
      this.setWidget(2,1, hp);

      this.setWidget(3,0, new Label("PostalCode"));
      this.setWidget(3,1, postalCode);
   }
}
```

❶ **Dołącz odwołanie do modelu**

❷ **Utwórz składowe widżety, których potrzebujemy dla klasy Address**

❸ **Rozłóż siatkę**

❹ **Utwórz podpanel tam, gdzie jest to potrzebne**

Klasa `AddressEdit` rozszerza prosty widżet `FlexTable` i ustawia elementy potrzebne do edycji obiektu `Address`.

OMÓWIENIE

Rozszerzenie widżetu `FlexTable` ❶ sprawia, że ustawienie widżetów w metodzie konstruktora jest bardzo łatwe. Tworzymy po prostu potrzebne nam elementy ❷ i dodajemy je. `FlexTable` działa bardzo podobnie do standardowej tabeli HTML, dynamicznie ponownie konfigurując się, kiedy elementy są dodawane do pozycji siatki ❸.

`FlexTable` i `Grid` są podklasami `HTMLTable` i jako takie używają tych samych metod do wstawiania elementów do struktury tabeli w określonym rzędzie i określonej kolumnie. Różnica polega na tym, że `FlexTable` dodaje metody, które umożliwiają nam dynamiczne wstawianie nowych komórek, podczas gdy `Grid` nie daje takiej możliwości. `Grid` pozwala na zmianę rozmiaru całej tabeli, rzędów

lub kolumn, ale nie umożliwia rozszerzenia w locie, tak jak `FlexTable`. Jest tak dlatego, ponieważ `Grid` z definicji musi utrzymywać swój prostokątny charakter, a `FlexTable` jest – jakby to powiedzieć – elastyczny.

Wewnątrz naszego widżetu korzystamy z `HorizontalPanel`, aby sformatować miasto i stan w tradycyjnej kolejności. Razem z `VerticalPanel` jest to najbardziej podstawowy `Widget` kontenera. Panel ten dodane do niego widżety ustawia poziomo w tej kolejności, w jakiej są dodawane (w tym wypadku dodano `TextBox`, Label i `ListBox`) ❹.

Rozszerzanie podstawowego widżetu GWT jest problematyczne dla widżetów, które chcemy udostępnić jako część interfejsu API. Ponieważ widżet `AddressEdit` rozszerza `FlexTable`, możliwe jest, że zewnętrzny kod wywoła metodę `setWidget()` poza konstruktorem i zmieni kompozycję widżetu. Nie chcemy, aby ktokolwiek tak robił, ponieważ `AddressEdit` jest przeznaczony wyłącznie do stosowania wewnątrz zakresu edycji użytkownika. Aby poradzić sobie z tym problemem, GWT dostarcza widżet `Composite`.

4.2.2. Rozszerzanie kompozytu

W naszym przykładzie `AddressEdit` z tego rozdziału i w przykładzie kalkulatora z rozdziału drugiego stworzyliśmy widżet, w niestandardowy sposób łącząc dostarczone panele i komponenty danych wpisywanych przez użytkownika. Chcieliśmy ponownie wykorzystać wszystkie funkcje `Widget` dostarczone przez GWT, a więc rozszerzyliśmy klasę `Panel` np. przez `VerticalPanel`. Chociaż takie rozwiązanie się sprawdza i jest proste (z tego właśnie powodu tak postępowaliśmy do tej pory), nie jest to zalecany sposób tworzenia niestandardowych komponentów GWT. Zamiast na ślepo tworzyć podklasy elementów panelu znacznie lepiej jest utworzyć klasę GWT `Composite`.

`Composite` jest klasą, która udostępnia tylko metodę `getElement()` wymaganą przez `UIObject` i dostarcza kilku chronionych metod do wykorzystania przez widżety, które dziedziczą po komponencie. Są to: `getWidget()`, `setWidget()` oraz `initWidget()`. Obiekty klasy `Composite` zawierają tylko pojedynczy podstawowy widżet, którym z oczywistych względów będzie zazwyczaj jakiś kontener. Widżety te są tworzone wewnątrz lokalnego konstruktora potomka `Composite`, a następnie dodawane do kompozytu za pomocą metody `initWidget()`. Wywołanie metody `initWidget()` musi zawsze być wykonane zanim klasa `Composite` może zostać dodana do kontenera, i powinno zatem być ostatnim wywołaniem w konstruktorze.

W tej części książki, aby zademonstrować `Composite`, utworzymy klasę `UserEdit` jako element kompozytowy.

PROBLEM

Musimy utworzyć widżet, ale nie chcemy udostępnić publicznych metod podstawowego widżetu GWT.

ROZWIĄZANIE

Zacznijmy od rozszerzenia klasy `Composite` i udostępnijmy jedynie te metody, które chcemy udostępnić, lub nie udostępniajmy żadnych w ogóle. Listing 4.3 przedstawia klasę `UserEdit` i konstrukcję widżetu `Composite`.

Listing 4.3. Tworzenie widżetu `UserEdit` klasy `Composite`.

```
public class UserEdit extends Composite {
   private User user;
   private StackPanel stack = new StackPanel();
   private TextBox username = new TextBox();
   private PasswordTextBox password = new PasswordTextBox();
   private PasswordTextBox passwordConfirm = new PasswordTextBox();
   private TextBox firstName = new TextBox();
   private TextBox lastName = new TextBox();
   private TextArea notes = new TextArea();
   private AddressEdit billingAddress;
   private AddressEdit shippingAddress;

  public UserEdit(final UserCreateListener controller, final User user) {
      super();
      this.user = user;
      stack.setHeight("350px");
      VerticalPanel basePanel = new VerticalPanel();
      Button save = new Button("Save");
      basePanel.add(save);
      basePanel.add(stack);
      FlexTable ft = new FlexTable();
      ft.setWidget(0,0, new Label("Username"));
      ft.setWidget(0,1, username);
      ft.setWidget(1,0, new Label("Password"));
      ft.setWidget(1,1, password);
      ft.setWidget(2,0, new Label("Confirm"));
      ft.setWidget(2,1, passwordConfirm );

      ft.setWidget(3,0, new Label("First Name"));
      ft.setWidget(3,1, firstName);
      ft.setWidget(4,0, new Label("Last Name"));
      ft.setWidget(4,1, lastName);

      stack.add(ft, "User Information" );
      billingAddress = new AddressEdit(
         user.getBillingAddress());
      stack.add(billingAddress, "Billing Address");
      shippingAddress = new AddressEdit(
         user.getShippingAddress());
      stack.add(shippingAddress, "Shipping Address");
      notes.setWidth("100%");
      notes.setHeight("250px");
      stack.add(notes, "Notes");

     this.initWidget(basePanel);
   }
}
```

❶ **Utwórz obiekt warstwy modelu dla widżetu**

Utwórz FlexTable do ustawienia podstawowych elementów

Utwórz klasy AddressEdit dla adresów

W tym przykładzie używamy naszej klasy `UserEdit` jako komponentu w większym widoku edycji dla obiektu modelu klasy `User`. Podczas gdy graf obiektu zawiera inne ustrukturyzowane obiekty (klasy `Address`), to klasa `UserEdit` zbiera je razem z edytorami dla innych części naszego modelu, które pojedynczo utworzyliśmy. Dostarcza również widżety edycji dla prostych wartości bezpośrednio dla klasy `User`. Na pierwszy rzut oka może to bardzo przypominać sposób, w jaki tworzyliśmy klasę `AddressEdit`, ale w rzeczywistości różni się jednak nieco.

OMÓWIENIE

Listing 4.3 tworzy nasz obiekt warstwy modelu ❶ i garść widżetów związanych z polami modelu. Zewnętrznym kontenerem jest po prostu `VerticalPanel`, a do oddzielenia różnych aspektów obiektu `User` używamy `StackPanel`. `StackPanel` jest specjalnym rodzajem kontenera, który udostępnia rozszerzający się i zwijający stos widżetów z etykietą, która przełącza pomiędzy nimi (bardzo podobnie jak pasek boczny w Microsoft Outlook).

Sposób wykorzystania `StackPanel` polega na przestrzeganiu pewnej nowej zasady, o której powinniśmy pamiętać, jeśli przeszliśmy od tradycyjnego programowania WWW: twórz komponenty interfejsu, a nie nawigacji. W tradycyjnej aplikacji WWW każdy element stosu może być osobną stroną i konieczne jest wówczas użycie konstrukcji `Session` na serwerze do zapisywania przejściowych stanów obiektu `User`. Tutaj możemy zwyczajnie wbudować cały proces konstrukcyjny obiektu `User` w jeden komponent, który pozwala użytkownikowi przemieszczać się pomiędzy tymi stanami obiektu. Oznacza to mniejsze wykorzystanie zasobów na serwerze, ponieważ kiedy przechodzimy pomiędzy różnymi sekcjami, radzimy sobie bez cyklu żądanie-odpowiedź i nie musimy więcej utrzymywać informacji o stanie dla każdego użytkownika korzystającego z aplikacji.

Stworzony przez nas obiekt `UserEdit` nie ma jeszcze żadnych udostępnionych metod poza `getElement()` i jest publiczny, a nie objęty zakresem pakietu, tak jak `AddressEdit`. Nie są to jednak ukończone klasy. Musimy jeszcze umożliwić im interakcję z warstwą modelu. Polega ona na obsłudze danych wprowadzanych przez użytkownika poprzez zdarzenia i na wprowadzaniu zmian do modelu w celu aktualizacji danych.

4.2.3. Wiązanie z modelem za pomocą zdarzeń

W podpunkcie 4.1 omówiliśmy, dlaczego potrzebujemy zdarzeń w warstwie modelu i jak dostarczyć tę funkcjonalność. Teraz musimy w warstwie widoku zaopatrzyć nasze widżety w logikę wiązania. GWT zawiera wiele podstawowych typów zdarzeń.

Tak naprawdę sporo widżetów GWT dostarcza większość powiadomień o zdarzeniach, jakich będziemy kiedykolwiek potrzebować. W naszym przykładzie z `UserEdit` do tej pory wykorzystaliśmy klasę `Button`, która rozszerza `FocusWidget`, który z kolei implementuje `SourcesClickEvent` i `SourcesFocusEvent`, aby zgłosić zdarzenia dla naszej implementacji `ClickListener`. Podobnie też

użyliśmy `TextBox`, który sam implementuje `SourcesKeyboardEvent`, `SourcesChangeEvent` oraz `SourcesClickEvent`. W interfejsie API GWT typy zdarzeń są określone przez te interfejsy zasobów (`Sources`), które informują programistów o tym, jakie zdarzenia obsługuje dany widżet. Możemy je wykorzystać, razem z `PropertyChangeEvent` z naszej warstwy modelu, aby dostarczyć dwustronnego powiązania z widokiem.

PROBLEM

Musimy powiązać dane z widżetu lub komponentu widoku z właściwością obiektu modelu.

ROZWIĄZANIE

Ponownie przyjrzymy się klasie kompozytowej `UserEdit`, aby zademonstrować wiązanie danych. Listing 4.4 pokazuje zmiany, jakie wprowadzamy do konstruktora, oraz nowe metody, które dodamy dla obsługi tego rozwiązania.

Listing 4.4. Klasa UserEdit.java zmodyfikowana tak, aby zawierała obsługę `PropertyChangeSupport`.

```
public class UserEdit extends Composite{
    // Pominięte wcześniej pokazane atrybuty
   private PropertyChangeListener[] listeners =
       new PropertyChangeListener[5];
public UserEdit(final UserCreateListener controller, final User user) {
       super();
   // Pominięte wcześniej pokazane tworzenie widżetu.
   listeners[0] = new PropertyChangeListenerProxy(
      "street1",
      new PropertyChangeListener() {
         public void propertyChange(
      PropertyChangeEvent propertyChangeEvent) {
         street1.setText(
         (String) propertyChangeEvent.getNewValue());
            }
         });
   address.addPropertyChangeListener(listeners[0]);
   street1.addChangeListener(
      new ChangeListener() {
         public void onChange(Widget sender) {
            address.setStreet1(street1.getText());
         }
      });
      // Powtarzający się wzór dla każdego z elementów
      save.addClickListener( new ClickListener() {
         public void onClick(Widget sender) {
            if(!password.getText().equals(
                  passwordConfirm.getText())) {
               Window.alert("Passwords do not match!");
               return;
```

Utwórz tablicę do przechowywania obiektów PropertyChangeListener

❶ **Utwórz PropertyChangeListener dla modelu**

Dodaj PropertyChangeListener do obiektu modelu

❷ **Powtórz dla każdej właściwości**

❸ **Utwórz odbiornik zmian dla widoku**

Uaktualnij model

❹ **Sprawdź passwordConfirm przed uaktualnieniem**

```
            }
            controller.createUser(user);   <-- 5 Wywołaj kontroler
         }
      });

      this.initWidget(basePanel);
   }
   public void cleanup(){                  <-- 6 Wyczyść odbiornik modelu
      for (int i=0; i < listeners.length; i++) {
         user.removePropertyChangeListener(listeners[i]);
      }
     billingAddress.cleanup();             <-- 7 Wyczyść elementy potomne widoku
      shippingAddress.cleanup();
   }
```

Mamy już teraz podstawowe wiązanie danych i operacje porządkowe w klasie `UserEdit`.

OMÓWIENIE

Dostarczanie dwustronnego wiązania danych wymaga, niestety, sporo powtarzalnego kodu ❷. W Swingu istnieje możliwość uproszczenia dużej części tego szablonowego kodu za pomocą klas narzędziowych opartych na refleksji. Ponieważ jednak interfejs API Reflection nie jest dostępny w kodzie GWT, musimy powtórzyć ten kod dla każdej właściwości. Najpierw tworzymy `PropertyChangeListener` ❶, który obserwuje model i uaktualnia widok, jeśli model się zmieni. Opakowujemy go w klasę `PropertyChangeListenerProxy`, która będzie filtrowała zdarzenia, zostawiając te, które chcemy obserwować. Chociaż nie jest to absolutnie niezbędne, to bardzo dobrze jest dostarczyć takie wiązanie w naszych widżetach. Zapewnia to, że jeśli inna część aplikacji uaktualni model, widok natychmiast to odzwierciedli i unikniemy bezładnego stanu, w którym różne komponenty będą „przyglądały się" tym samym obiektom.

UWAGA Klasa `PropertyChangeSupport` umożliwiając dodanie obiektów klasy `PropertyChangeListener` poprzez określenie nazwy właściwości, opakuje je wewnątrz klasy `PropertyChangeListenerProxy`. W takim wypadku tracimy możliwość wywoływania metody `removePropertyChangeListener()` bez określania nazwy właściwości. Ponieważ chcemy po prostu trawersować tablicę odbiorników w naszej metodzie `cleanup()`, dlatego opakowujemy je podczas ich tworzenia, tak iż metoda będzie działała zgodnie z oczekiwaniami.

Następnie tworzymy `ChangeListener` i dołączamy go do widżetu odpowiedzialnego za daną właściwość ❸. Wraz z każdą zmianą widżetu model będzie uaktualniany. W tym wypadku używamy widżetów `TextBox`, a więc wywołujemy metodę `getText()`, aby określić ich wartość. Jeśli nie jest Ci obce programowanie Ajax/DHTML, to wiesz, że popularnym wzorcem dla elementu `<input type="text">` jest domknięcie `onChange`, które jest uruchamiane tylko wówczas, gdy wartość

się zmieniła, a ten element traci aktywność. Trzeba niekiedy o tym pamiętać, ale ponieważ wiemy, że kiedy użytkownik kliknie przycisk „Save" („Zapisz"), to element straci aktywność, a więc nie musimy się o to tutaj martwić. Jeśli potrzebujemy takich szczegółów dotyczących zmian, możemy użyć `KeyboardListener` wobec widżetów `TextBox`, które zostaną uruchomione przy każdym naciśnięciu klawisza podczas gdy pole tekstu jest aktywne.

Dla niektórych widżetów może istnieć konieczność dostarczenia trochę kodu logicznej konwersji, aby wypełnić model. Poniżej znajduje się niewielka część klasy `AddressEdit`, w której uaktualniamy właściwość `state` obiektu `Address`.

```
listeners[4] = new PropertyChangeListener() {
   public void propertyChange(
      PropertyChangeEvent propertyChangeEvent) {
      for(int i=0; i < state.getItemCount(); i++) {
         if(state.getItemText(i).equals(
            propertyChangeEvent.getNewValue())) {
            state.setSelectedIndex(i);
            break;
         }
      }
   }
};
address.addPropertyChangeListener("state", listeners[4]);
state.addChangeListener(new ChangeListener() {
   public void onChange(Widget sender) {
      String value = state.getItemText(state.getSelectedIndex());
      if(!"...".equals(value)) {
         address.setState(value);
      }
   }
});
```

Przypomina to bardzo powtarzające się wzorce w widżetach `TextBox`, jednak w obu odbiornikach musimy określić wartość w odniesieniu do właściwości `SelectedIndex` (`state`) widżetu `ListBox`.

Kiedy użytkownik naciska przycisk „Save" („Zapisz"), musimy przeprowadzić wywołanie zwrotne do warstwy kontrolera, aby zapisać dane użytkownika ❺. Można zauważyć, że dokonujemy tutaj drobnego sprawdzania poprawności danych: sprawdzamy, czy wartości `password` i `passwordConfirm` są takie same ❹. Pole `passwordConfirm` nie jest właściwie częścią naszego modelu, ale jest po prostu pewnym drobiazgiem interfejsu użytkownika. To, gdzie dokonuje się sprawdzania danych, stanowi zagadnienie samo w sobie interesujące. W niektórych sytuacjach możemy znać poprawne wartości i po prostu zamieścić procedury weryfikacji w metodach ustawiających modelu oraz przechwycić wyjątki w odbiornikach `ChangeListener` widoku. Dostarcza to dużo bezpośrednich informacji kontrolnych dla użytkownika wówczas, gdy wypełnia on formularze. W przypadku czegoś większego, tak jak alternatywy rozłącznej czy spraw wymagających weryfikacji

serwera, dostarczanie w kontrolerze opcji sprawdzania poprawności jest najlepszym rozwiązaniem. Oczywiście, ponieważ GWT opiera się na Javie, możemy użyć tej samej logiki walidacji po stronie serwera i po stronie klienta, zaoszczędzając sobie trudu, który musielibyśmy włożyć w bardziej tradycyjnym programowaniu Ajaksa.

Ostatnią ważną rzeczą, na którą trzeba tutaj zwrócić uwagę, jest metoda `cleanup()` ❻. Przegląda ona po prostu obiekty klasy `PropertyChangeListener`, które dodaliśmy do klasy modelu, i usuwa je. Jest to istotne, ponieważ kiedy aplikacja skończy już z widżetem `UserEdit`, potrzebuje sposobów wyczyszczenia go. Gdybyśmy nie usunęli tych odbiorników z obiektu `User` o dłuższym żywocie, to odwołanie do `UserEdit` nigdy nie zostałoby poddane mechanizmowi oczyszczania pamięci i nadal uczestniczyłoby w wykonywaniu zdarzenia, niepotrzebnie zabierając cykle procesora. Oczywiście, ponieważ widżet `AddressEdit` również to wykonuje, musimy także oczyścić te odbiorniki ❼.

Dlaczego oczyszczamy obiekty `PropertyChangeListener`, a nie `ChangeListener` czy `ClickListener`, które wykorzystaliśmy w pochodnych widżetach? Dlatego że te odbiorniki zmian wyszłyby poza zasięg i byłyby poddane oczyszczaniu pamięci w tym samym czasie, co nasza klasa `UserEdit`. Ponieważ są one prywatnymi elementami, a kompozyt `UserEdit` przesłania wszystkie inne operacje, to klasy poza zakresem `UserEdit` nie mogą utrzymać odwołań do nich.

Mamy teraz model i widok oraz określiliśmy relacje pomiędzy nimi, więc musimy teraz ustawić kontroler i wysłać informację o rejestracji użytkownika z powrotem do serwera.

4.3. Kontroler i usługa

Mogliśmy zauważyć, że przekazaliśmy egzemplarz `UserCreateListener` do konstruktora `UserEdit`. Przy projektowaniu aplikacji ważne jest, aby nasze niestandardowe widżety uzewnętrzniały całą logikę biznesową. Jeśli chcemy działać na rzecz ponownego wykorzystania naszego kodu widoku, nie powinien być on niepotrzebnie związany z określonym zestawem logiki biznesowej. W tym przypadku nasza logika kontrolera jest jednak dość prosta.

W tej części książki stworzymy kontroler i serwlet usług, który zapisze naszego użytkownika w bazie danych, oraz wskażemy miejsca, gdzie możemy rozszerzyć projekt o inne funkcje.

4.3.1. Tworzenie prostego kontrolera

Ogólnym zadaniem kontrolera jest dostarczanie dostępu do logiki biznesowej i zapewnianie systemu kontroli dla stanu widoku. Pomyślmy przez chwilę o poziomie kontrolera komponentu `Action` w aplikacji Struts. Załóżmy, że jest on aktywowany na podstawie zdarzenia użytkownika – przesłania formularza. Następnie weryfikuje on dane i przesyła je do jakiejś logiki biznesowej (chociaż, niestety, wiele aplikacji Struts umieszcza logikę biznesową wprost w komponencie `Action`) i nakazuje widokowi uaktualnić się zgodnie z nowym stanem, przekierowując do

jakiejś innej strony. Powinniśmy myśleć o kontrolerze w aplikacji GWT, iż spełnia on taką rolę, ale robi to w zupełnie inny sposób.

Przyjrzymy się teraz prostemu kontrolerowi w postaci `UserCreateListener`.

PROBLEM

Musimy utworzyć kontroler do zarządzania zdarzeniami i stanem w naszej klasie widoku. Uruchomi to przypadki działań w naszej aplikacji.

ROZWIĄZANIE

Zaczniemy od utworzenia prostej implementacji interfejsu `UserCreateListener`, przedstawionej w listingu 4.5.

Listing 4.5 `UserCreateListenerImpl` – kontroler widżetu `UserEdit`.

```
package com.manning.gwtip.user.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
public class UserCreateListenerImpl implements UserCreateListener {
   private UserServiceAsync service =
      (UserServiceAsync) GWT.create(UserService.class);

   public UserCreateListenerImpl() {
      super();
      ServiceDefTarget endpoint = (ServiceDefTarget) service;
      endpoint.setServiceEntryPoint
            (GWT.getModuleBaseURL()+"UserService");
   }

   public void createUser(User user){
      if ("root"
               .equals(
                  user.getUsername())) {
            Window.alert("You can't be root!");
         return;
      }
      service.createUser(user, new AsyncCallback() {
         public void onSuccess(Object result) {
            Window.alert("User created.");
            // zmienimy tu widok według nowego stanu.
         }

         public void onFailure(Throwable caught) {
            Window.alert(caught.getMessage());
         }
      });
   }
}
```

❶ Utwórz usługę

❷ Powiąż adres serwera

Sprawdź poprawność danych

Przekaż użytkownikowi komunikat

Mamy już klasę kontrolera dla naszego widżetu `UserEdit`. Kontroler ten będzie przeprowadzał wywołania zwrotne do zdalnej usługi, kończąc interfejs użytkownika aplikacji.

OMÓWIENIE

Jest to prosty, a nawet nieco banalny, przykład, lecz pokazuje on logicznie następujące po sobie etapy, jakimi powinniśmy się kierować w naszej aplikacji. Najpierw uzyskujemy usługę ❶ i wiążemy ją ❷, jak widzieliśmy w rozdziale trzecim. Następnie implementujemy wymaganą metodę `createUser()`. Metoda ta rozpoczyna się od niewielkiego fragmentu weryfikacji danych i mogłoby to być zdecydowanie bardziej zaawansowane.

Dobrym rozwiązaniem byłoby utworzenie obiektu `UserValidator`, który mógłby wykonać dowolne sprawdzanie poprawności, jakie byłoby nam potrzebne. Ten prosty przykład prezentuje jedynie, gdzie to się dzieje. Po weryfikacji przeprowadzamy wywołanie zdalnej usługi i zajmujemy się wynikami. Gdyby była to część większej aplikacji, wówczas metoda `onSuccess()` mogłaby dokonać wywołania zwrotnego innej klasy, aby usunąć panel `UserEdit` z ekranu monitora i ustawić dla użytkownika inny panel, tak jak robi to „forward" w kontrolerze akcji Struts.

Inny przypadek walidacji pokazywałby użytkownikowi powiadomienie o błędzie, jeśli coś zawiedzie podczas wywołania serwera. Może to wskazywać na błąd lub dane, które nie przeszły walidacji na serwerze. Przykładowo: powtarzające się nazwy użytkowników nie mogą zostać łatwo sprawdzone po stronie klienckiej. Musimy to sprawdzić na etapie zamieszczania użytkownika w bazie danych.

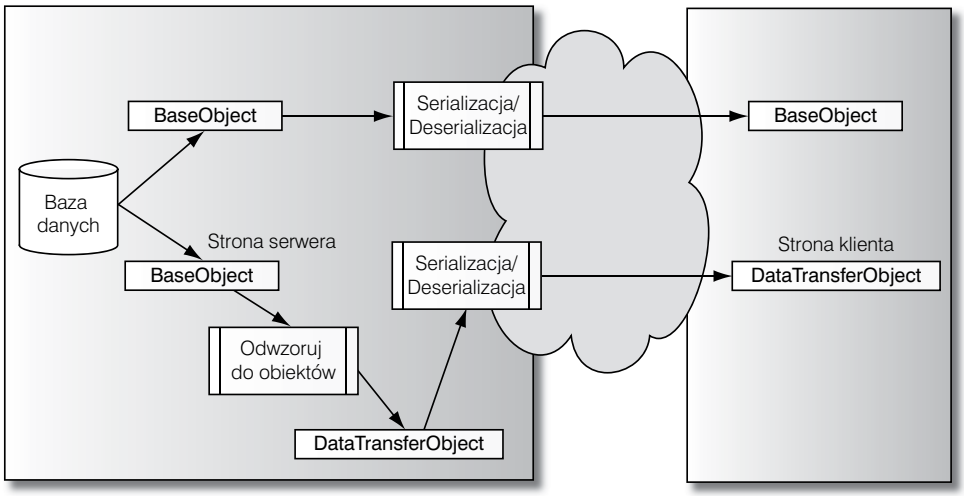
Wszystko to prowadzi nas do uzyskiwania dostępu do bazy danych w usłudze. Do tego celu użyjemy Java Persistence API.

4.3.2. Aktywowanie JPA w modelu

Jednym z najczęściej powtarzających się pytań w programowaniu GWT jest: „Jak mam się dostać do bazy danych?" W rozdziale trzecim widzieliśmy podstawy konfiguracji serwera Tomcat Lite, jednak większość osób chce zastosować w swojej bazie danych coś bardziej skomplikowanego niż surowy interfejs JDBC. Chociaż JDBC dobrze się sprawdza, jest jednak bardziej kłopotliwy w użytkowaniu niż obiektowe interfejsy API do utrwalania danych. Dzisiaj zazwyczaj będzie to jakiś dostawca JPA, taki jak Hibernate, Kodo, OpenJPA lub TopLink Essentials.

Istnieją dwa podstawowe sposoby używania JPA w środowisku GWT. Pierwszym z nich jest aktywowanie JPA w modelu dzielonym pomiędzy klientem i serwerem. Drugi polega na utworzeniu obiektów DTO (*Data Transfer Object*), które nadają się do wykorzystania po stronie klienta, i przekonwertowanie ich w usłudze na coś odpowiedniego do stosowania na serwerze. Rysunek 4.5 prezentuje różnicę w systemach pomiędzy tymi dwoma sposobami.

W każdym z tych sposobów trzeba pójść na pewne ustępstwa. Jeśli aktywujemy JPA we współdzielonym modelu, nasze klasy modelu będą wówczas ograniczone tym, co klasy emulacji GWT JRE obsługują, oraz ogólnymi restrykcjami wynikającymi z przekształcenia na GWT (na chwilę obecną – brak konstruktora argumentów, brak konstrukcji języka Java 5 itd.). Korzystanie z podejścia DTO oraz dokonywanie konwersji pomiędzy wieloma obiektami przesyłu danych komplikuje



Rysunek 4.5. Droga od serwera do klienta z obiektami `DataTransferObject` i bez nich. Zwróćmy uwagę, że dodatkowy etap odwzorowania jest potrzebny, jeśli korzystamy z DTO.

i potencjalnie znacznie poszerza kod naszej aplikacji, ale jednocześnie dostarcza płynniejszej kontroli nad modelem rzeczywiście używanym przez naszego klienta GWT.

Z powodu ograniczeń wiążących się z bezpośrednim użyciem JPA i ze względu na inne zalety warstwy DTO, często stosuje się podejście DTO do komunikowania się pomiędzy GWT a modelem strony serwera. Przyjrzymy się temu wzorcowi, używając obiektów transferowych i wszystkich innych aspektów z tym związanych, w rozdziale dziewiątym, w którym zajmiemy się większą aplikacją zarządzaną przez Springa. W tym rozdziale zajmiemy się aktywowaniem JPA w naszych komponentach GWT, co stanowi najłatwiejszą metodę w prostych i niezależnych aplikacjach.

Problem

Chcemy udostępnić komponenty modelu do wykorzystania z dostawcami JPA, aby utrwalić je w bazie danych.

Rozwiązanie

Jeśli używałeś wcześniej JPA i pamiętasz, że komponenty klienckie GWT są powiązane ze składniową strukturą Javy 1.4, zapewne myślisz sobie: „nie można dodawać adnotacji do tych komponentów!" Dobra intuicja – i miałbyś rację. Istnieje jednakże inny sposób opisu komponentów JPA, któremu zwykle nie poświęca się wiele uwagi, a który jest właśnie stworzony do tego typu sytuacji: wykorzystanie pliku odwzorowań orm.xml. Potrzebujemy także, oczywiście, pliku persistence.xml, aby zadeklarować jednostkę utrwalania. Listing 4.6 przedstawia definicję jednostki utrwalania.

Listing 4.6. Plik persistence.xml dla użytkownika.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
             http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
             version="1.0">

   <persistence-unit name="user-service"
      transaction-type="RESOURCE_LOCAL">
      <provider>
         org.hibernate.ejb.HibernatePersistence
      </provider>
      <class>com.manning.gwtip.user.client.User</class>
      <class>com.manning.gwtip.user.client.Address</class>
      <properties>
         <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect"/>
         <property name="hibernate.connection.driver_class"
         value="com.mysql.jdbc.Driver"/>
         <property name="hibernate.connection.username"
            value="userdb"/>
         <property name="hibernate.connection.password"
            value="userdb"/>
         <property name="hibernate.connection.url"
         value="jdbc:mysql://localhost/userdb"/>
         <property name="hibernate.hbm2ddl.auto"
         value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

Określ użycie frameworka Hibernate

Określ użycie MySQL

Usuń i utwórz bazę danych za każdym razem

Jeśli znasz już JPA, powyższy listing nie powinien Cię zdziwić. Nie używamy tutaj `DataSource`, lecz tworzymy po prostu bezpośrednie połączenia z bazą danych. Korzystamy również z frameworka Hibernate. Chociaż mamy już jako autorzy niniejszej książki doświadczenie z Hibernate i TopLink Essentials jako dostarczycielami JPA, to wybraliśmy dla tego przykładu Hibernate. Postąpiliśmy tak, ponieważ łatwiej pokazać Hibernate'a w powłoce GWT, mimo że wymaga on większej liczby zależności. TopLink również działa w powłoce, ale wymaga jeszcze dodatkowych kroków poza dołączaniem zależności, takich jak popierany mechanizm przesłaniania wbudowanej w Tomcata wersji Xerces oraz dołączanie agenta TopLink (będziemy używali pakietu TopLink w kilku innych przykładach w dalszej części książki).

Potrzebujemy teraz, aby plik orm.xml określił metadane, które normalnie podalibyśmy w adnotacjach. Listing 4.7 pokazuje plik odwzorowania dla naszych obiektów użytkownika.

Listing 4.7. Plik orm.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                version="1.0">
   <package>com.manning.gwtip.user.client</package>
   <entity class="User"
     metadata-complete="true" access="PROPERTY">
      <table name="USER"/>
      <named-query name="User.findUserByUsernameAndPassword">
         <query>select u from User u
            where u.username = :username
               and u.password = :password</query>
      </named-query>
      <attributes>
         <id name="username" />
         <one-to-one name="shippingAddress" >
            <cascade>
               <cascade-all />
            </cascade>
         </one-to-one>
         <one-to-one name="billingAddress" >
            <cascade>
               <cascade-all />
            </cascade>
         </one-to-one>
      </attributes>
   </entity>
   <entity class="Address" metadata-complete="true" access="PROPERTY">
      <table name="ADDRESS"/>
      <attributes>
         <id name="id">
            <generated-value strategy="IDENTITY"/>
         </id>
      </attributes>
   </entity>
</entity-mappings>
```

❶ **Pomiń kolejne kroki dzięki atrybutowi metadata-complete**

Kaskaduj do obiektów Address

Automatyczne zwiększanie wartości pola identyfikatora

Przypomina to bardzo adnotacje, które możemy udostępnić w samych plikach Javy. Rzeczywiście, plik orm.xml pokrywa się niemalże z adnotacjami. Należy zwrócić uwagę na atrybut `metadata-complete` elementu `<entity>` ❶. Wskazuje on zarządcy encjami, aby użył swojego domyślnego zachowania wobec jakichkolwiek właściwości obiektu, które nie są wprost przedstawione w pliku.

OMÓWIENIE

Dla właściwości identyfikatora id obiektu `Address` używamy strategii `IDENTITY`, która zastosuje automatyczne zwiększanie pól systemu MySQL. Jest to kolejny

element różniący Hibernate i TopLink. Mianowicie, TopLink nie obsługuje schematu `IDENTITY` z jego dialektem MySQL4. Musimy użyć wirtualnej sekwencji. W tym wypadku element `<entity>` adresu wyglądałby następująco:

```
<entity class="Address" metadata-complete="true" access="PROPERTY">
   <table name="ADDRESS"/>
   <sequence-generator
      name="addressId" sequence-name="ADDRESS_ID_SEQUENCE" />
   <attributes>
      <id name="id">
         <generated-value strategy="SEQUENCE" generator="addressId"/>
      </id>
   </attributes>
</entity>
```

MySQL nie obsługuje sekwencji jako struktura bazy danych, jednak TopLink tworzy tabelę, aby utrzymywać wartość sekwencji. Hibernate uchyla się od tej konfiguracji, ponieważ wie, że MySQL nie obsługuje sekwencji. W skrócie, nie oczekujmy, że te pliki konfiguracyjne raz napisane, można będzie wszędzie uruchomić. Wszystko, począwszy od zastosowanego dostawcy JPA po samą użytą bazę danych jest powiązane w naszej aplikacji. Miejmy nadzieję, że wraz z rozwojem implementacji EJB 3/JPA kwestie te zostaną rozwiązane.

Plik orm.xml nie jest dobrze udokumentowany. Najlepiej jest po prostu przyjrzeć się samemu plikowi schematu (http://java.sun.com/xml/ns/persistence/orm_1_0.xsd) i użyć edytora XML sprawdzającego poprawność. Innym rozwiązaniem jest użycie narzędzia inżynierii wstecznej OpenJPA (http://incubator.apache.org/openjpa/), które ma możliwość utworzenia pliku orm.xml z istniejącego schematu bazy danych.

Mamy już teraz odwzorowania JPA umożliwiające nam przechowywanie w bazie danych obiektów z naszego modelu, które utworzyliśmy na samym początku. Ostatnim krokiem jest utworzenie komponentu usług, który połączy kontroler kliencki (który utworzyliśmy w podpunkcie 4.3.1) oraz bazę danych.

4.3.3. Tworzenie usług z aktywowanym JPA

Wspomnieliśmy w rozdziale trzecim, że najczęściej najlepszym rozwiązaniem jest tworzenie oddzielnej usługi lokalnej i skierowanie do niej wywołań naszego pośrednika `RemoteServiceServlet`. Chociaż przyjrzymy się temu szczegółowo w rozdziale dziewiątym, to tutaj po prostu utworzymy nasz kod usługi w serwlecie. Ponieważ aktywujemy JPA w naszym modelu, to fakt, iż nasza usługa jest zależna od bibliotek GWT, nie niesie ze sobą żadnych ujemnych konsekwencji.

PROBLEM

Musimy utworzyć serwlet usług z aktywowanym JPA, aby nasza aplikacja przyjmowała obiekty modelu wysyłane z warstwy kontrolera klienta i zapisywała je w bazie danych.

ROZWIĄZANIE

Listing 4.8 przedstawia pośrednik `RemoteServiceServlet`, który będzie przyjmował obiekty modelu i utrwalał je w bazie danych.

Listing 4.8. `UserServiceServlet` wraz z wywołaniami JPA.

```
public class UserServiceServlet extends RemoteServiceServlet
   implements UserService {
   private EntityManagerFactory factory;

   public UserServiceServlet() {
      super();
      try{
         factory =
            Persistence.createEntityManagerFactory(
         "user-service");
      } catch(Exception e){
         e.printStackTrace();
      }
   }

   public void createUser(User user) throws UserServiceException {
      if("root".equals(user.getUsername())) {
         throw new UserServiceException(
            "You can't be root!");
      }
      try{
         EntityManager mgr = factory.createEntityManager();
         mgr.getTransaction().begin();
         mgr.persist(user);
         mgr.getTransaction().commit();
      } catch(RollbackException e) {
         throw new UserServiceException(
         "That username is taken. Try another!");
      } catch(PersistenceException p) {
         throw new UserServiceException(
         "An unexpected error occurred: "+p.toString());
      }
   }
}
```

Nie można użyć adnotacji @PersistenceContext ❶

Utwórz EntityManagerFactory ❷

Pamiętaj, nigdy nie ufaj klientowi ❸

Przechwyć wyjątki utrwalania

To prawdopodobnie wygląda znajomo dla każdego, kto pracował z JPA, jednak trzeba omówić kilka ważnych kwestii związanych z wykorzystaniem tutaj JPA.

OMÓWIENIE

Jest to dosyć prosta klasa, choć trzeba zwrócić uwagę na pewne istotne sprawy. Po pierwsze, nie używamy adnotacji `@PersistenceUnit` w celu uzyskania naszego obiektu `EntityManagerFactory` ❶. Byłby to „normalny" sposób uzyskania obiektów `EntityManagerFactory` w Javie EE 5. Tomcat w powłoce GWT – lub Tomcat w ogólności – nie obsługuje wstrzykiwania zależności Java EE 5. Możemy użyć Springa w zwykłym Tomcacie, ale ponieważ wszystkie serwlety w powłoce GWT są tworzone przez pośredniczącą usługę powłoki, nie możemy tego zrobić

w prosty sposób w trybie rozwojowym. Tworzymy zatem po prostu sami fabrykę w konstruktorze ❷.

Kolejną ważną rzeczą, na którą musimy zwrócić uwagę, jest to, że ponownie sprawdzamy poprawność danych użytkownika ❸. Pamiętajmy, że zawsze powinniśmy sprawdzać dane po obu stronach. Sprawdzenie ich po stronie klienta poprawia doznania użytkowników, natomiast sprawdzenie ich na serwerze zwiększa niezawodność aplikacji.

W końcu, dokonujemy sprawdzenia pod kątem wyjątku `RollBackException`, który jest zgłaszany, jeśli nazwa użytkownika już znajduje się w bazie danych. Jest to zepsuta część Hibernate'a. Gdybyśmy używali TopLink, przechwycilibyśmy „prawidłowy" wyjątek `javax.persistence.EntityExistsException`. Tego rodzaju różnice są kolejnym przykładem aktualnych wyzwań związanych z używaniem obecnej generacji dostawców JPA.

Mamy już teraz wszystkie komponenty potrzebne do stworzenia podstawowej aplikacji internetowej. Mamy warstwę modelu, która powiadamia odbiorniki o zmianach, warstwę widoku, która wiąże z modelem i dostarcza aktualizacji, oraz warstwę kontrolera, która przechwytuje nasze przypadki użycia i przekazuje żądania do usługi. Wreszcie, mamy usługę, która utrwala naszą warstwę modelu w bazie danych. Chociaż deweloper Struts lub JSF może być przyzwyczajony do budowania takich samych komponentów, to wyglądają one inaczej w kodzie i przynoszą ze sobą nowe zagadnienia projektowe, które musi rozważyć programista WWW.

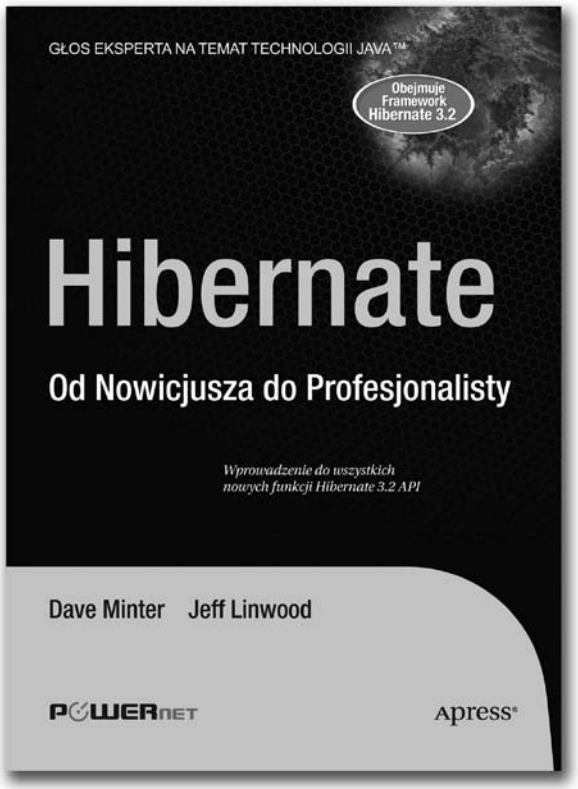
4.4. Podsumowanie

W tym rozdziale zapoznaliśmy się ze standardowymi wzorcami tworzącymi aplikację GWT. Chociaż typowy model MVC nadal obowiązuje, to jednak sposób jego użycia bardzo się różni od wzorca MVC po stronie serwera, który mogliśmy wykorzystywać w przeszłości w tradycyjnych szkieletach aplikacji WWW. Po pierwsze, obiekt modelu klienckiego jest znacznie inteligentniejszy od prostego obiektu wartości. Wiązanie danych jest przeprowadzane na podstawie zdarzeń, a wywołania do serwera są bardziej zorientowane na usługi, reprezentując przypadki użycia naszej aplikacji, a niekoniecznie „ekran".

Możesz już wiedzieć, jak zrobić niektóre rzeczy, jednak sposoby, w jakie chciałbyś je wykonać, mogą początkowo się nie sprawdzić w środowisku powłoki GWT. Przeanalizowaliśmy tutaj jeden ze sposobów integracji JPA z aplikacją GWT. W rozdziale dziewiątym przyjrzymy się innemu rozwiązaniu, które integruje Springa, JPA i DTO. Takie podejście pozwala na łatwiejszą i bardziej elegancką integrację z istniejącymi systemami JEE. Wykorzystany przez nas wzorzec aktywujący JPA i GWT jest bardziej odpowiedni dla prostych, niezależnych aplikacji.

Do tej pory skupialiśmy się na budowaniu aplikacji GWT za pomocą standardowego rozwiązania RPC do komunikowania się z serwerami. Nie jest to jednak jedyny sposób komunikowania się z innymi back-endami serwera, łącznie z tymi,

które nie są zamieszczone na serwlecie. W następnym rozdziale zajmiemy się dodatkowymi sposobami łączenia się z serwerami, takimi jak podstawowy HTTP i bardziej zaawansowane techniki typu XML, SOAP, REST, Flash oraz Comet. Po drodze zwrócimy uwagę na najważniejsze pojęcia związane z serializacją obiektów klient-serwer za pomocą Javy i JavaScriptu, a także przyjrzymy się kwestii bezpieczeństwa.



Hibernate

Od Nowicjusza do Profesjonalisty

Autorzy
Dave Minter, Jeff Linwood
ISBN: 978-83-924603-0-5
Tytuł oryginału:
Beginning Hibernate: From Novice to Professional

„*Hibernate. Od nowicjusza do profesjonalisty*" jest idealną pozycją dla tych, którzy mają już jakieś doświadczenie w Javie z bazami danych, ale stawiają dopiero pierwsze kroki z frameworkiem Hibernate (aktualnie najpopularniejszy framework do odwzorowywania obiektów na struktury danych składowane w relacyjnych bazach danych).
Książka ta zawiera najnowsze informacje o ostatnim wydaniu Hibernate 3.2.x. Dostarcza jasnego wprowadzenia do aktualnego standardu w dziedzinie odwzorowań obiektów w bazie danych.
Doświadczeni autorzy Dave Minter i Jeff Linwood dostarczają więcej dogłębnych przykładów niż jakiekolwiek inne książki o Hibernate dla początkujących. Autorzy prezentują materiał, rozwiązując problemy, z którymi spotkamy się nieraz w trakcie pracy z Javą i bazami danych. Ponieważ książka skupia się na Hibernate bez tracenia czasu na nieistotne narzędzia, będziecie Państwo w stanie bardzo szybko stworzyć aplikacje wykorzystujące nową technologię.