GWT の魅力: 第 1 回 Google Web Toolkit を利用して各地を訪ね回る

Java コードでデスクトップのような Web アプリケーションを実装する方法

David Geary
President
Clarity Training, Inc.

2009年 9月 01日

GWT (Google Web Toolkit) では、ブラウザーで動作するリッチ・クライアントのユーザー・インターフェースを Java™ 言語で実装することができます。この 2 回の連載記事では、David Geary が最新バージョンの GWT に関する情報を提供し、デスクトップのような Web アプリケーションを実装する方法を説明します。

このシリーズの他の記事を見る

1990年代、私は Swing をかなり使っていました。Swing がお気に入りだった理由は、想像し得るものはどんなものでも実装することができたからです。何かを頭で考え、画面上でそれに命を吹き込むことは、私にとってまさにソフトウェア開発の醍醐味です。Swing の API を使うことで、ドラッグ・アンド・ドロップからゲーム・センターのゲームに至るまで、あらゆるアプリケーションが手に届く範囲に入ってきました。

その後、サーバー・サイド Java と Struts のようなプリミティブなフレームワークの登場によって、なんと、1960年代に戻ってメインフレームのようなフォームを実装することが可能になりました。ドラッグ・アンド・ドロップもなければ、ゲーム・センターのビデオ・ゲームもない、ほとんど面白味のないプログラミングの石器時代に再び突入したというわけです。

だからこそ、私は GWT (Google Web Toolkit) に魅せられています。GWT では以前のように Swing のような API を使って、想像し得るどんなものでも、今度はブラウザーで実現できるからです。もちろん Web アプリケーション・フレームワークの現状は Struts 1.0 からは大幅に変わっていて、今では JSF 2、Ruby on Rails、そして Lift といったフレームワークにより、メインフレーム風のフォームという枠を遙かに超えたものを実装できるようになっています。けれども、お馴染みの言語とお馴染みの API を使用して JavaScript そのままの威力を利用できるという点では、GWT に並ぶものはありません。ブラウザーで実行されるデスクトップのようなアプリケーションを実装したいのであれば、GWT はクライアント・サイドでだけでも、その手段として真剣に考えるべき候補となります。

この 2 回の連載記事では GWT の基本をおさらいし、現行の GWT API での変更に関する情報を提供するとともに、GWT を使ったアプリケーション開発の高度な側面について見て行きます。GWT の初期リリースについての包括的な説明は、「Java 開発者のための Ajax: Google Web Toolkit を探る」を読んでください。

この短い連載では、デスクトップのようなアプリケーションを実装する手順を通して、GWT の学習曲線を辿ります。今回の記事では、以下の内容を取り上げます。

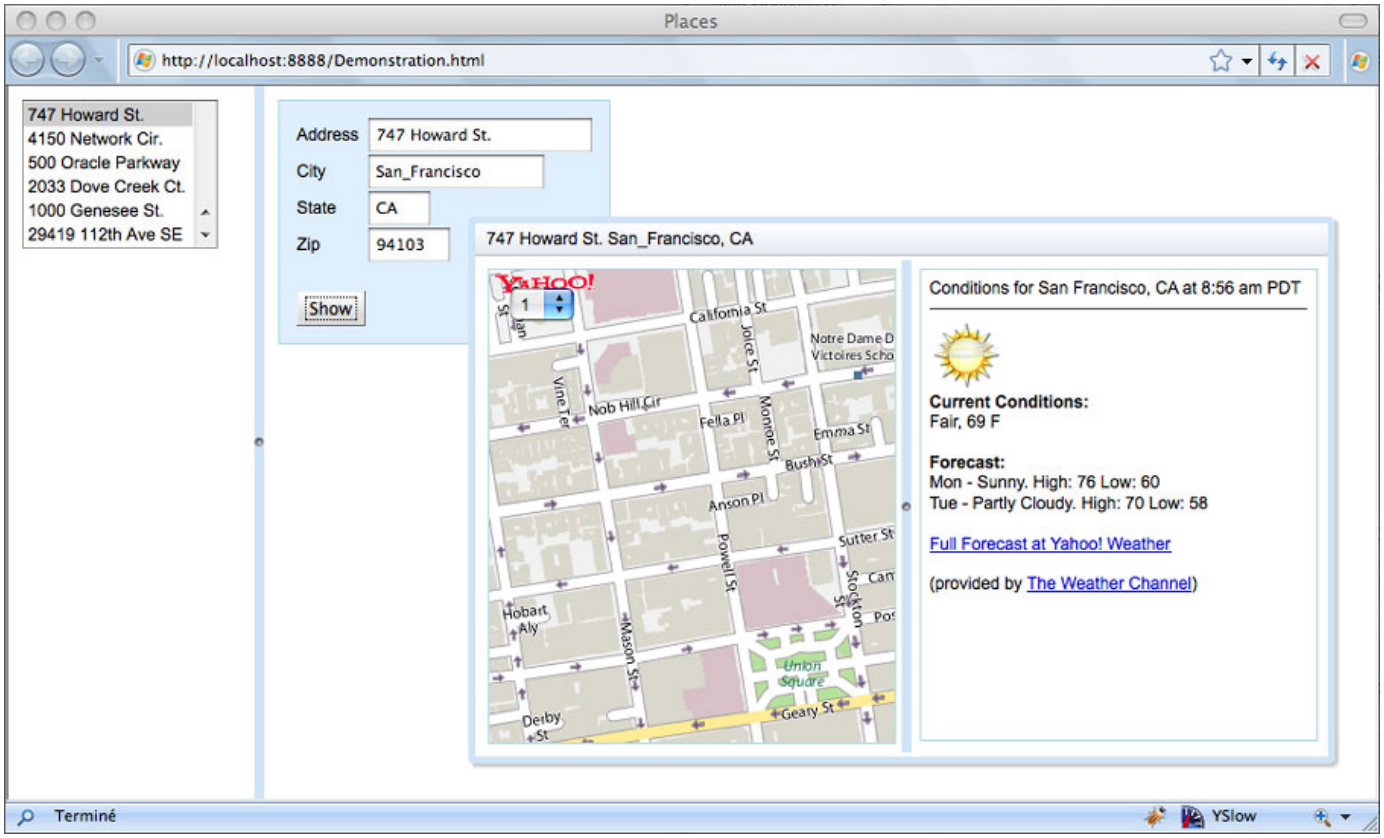
- ウィジェット
- リモート・プロシージャー呼び出し (RPC) とデータベース統合
- 複合ウィジェット
- イベント・ハンドラー
- Ajax のテスト

第 2 回では、カスタム・ウィジェットの実装と、イベント・プレビューの使用やタイマーを使って行う画像のアニメーション化などといった高度な手法を詳しく検討します。完全なサンプル・アプリケーションのソース・コードは、ダウンロードすることができます。

Places: Ajax による、データベースがサポートする Web サービスのマッシュアップ

これから GWT で作成するのは、さまざまな場所を表示できる Places アプリケーションです。このアプリケーションでは、図 1 に示すように、指定された場所についての地図と天気の情報の組み合わせとして場所を定義します (図 1 を参照)。

図 1. Places アプリケーション: 1 つの場所を表示する



Places アプリケーションには 6 つの住所があらかじめ組み込まれています。アプリケーションは起動時にこれらの住所を MySQL データベースから取得して、左側にあるリスト・ボックスに表示します。リスト・ボックス内の住所をクリックすると、アプリケーションは選択された住所でリスト・ボックスの右側にあるグリッドを更新します。

GWT と JSF (JavaServer Faces) との比較

この記事の Places アプリケーションは、3 回からなる連載「JSF 2 の魅力」で説明した同名のアプリケーションと同様のものです。両方の連載を読むと、この 2 つのフレームワークがどのように違っているかを感じ取ることができます。

グリッドの Show ボタンをクリックすると、アプリケーションがウィンドウを開きます。このウィンドウには左右に分割されたパネルがあり、左側に地図、右側に天気の情報が表示されます。アプリケーションはこの地図と天気の情報を Yahoo! の Web サービスから取得します。このアプリケーションではウィンドウを一度に複数表示することも、分割パネルのサイズを調整することもできます (図 2 を参照)。

図 2. Places アプリケーション: 複数の場所を表示する

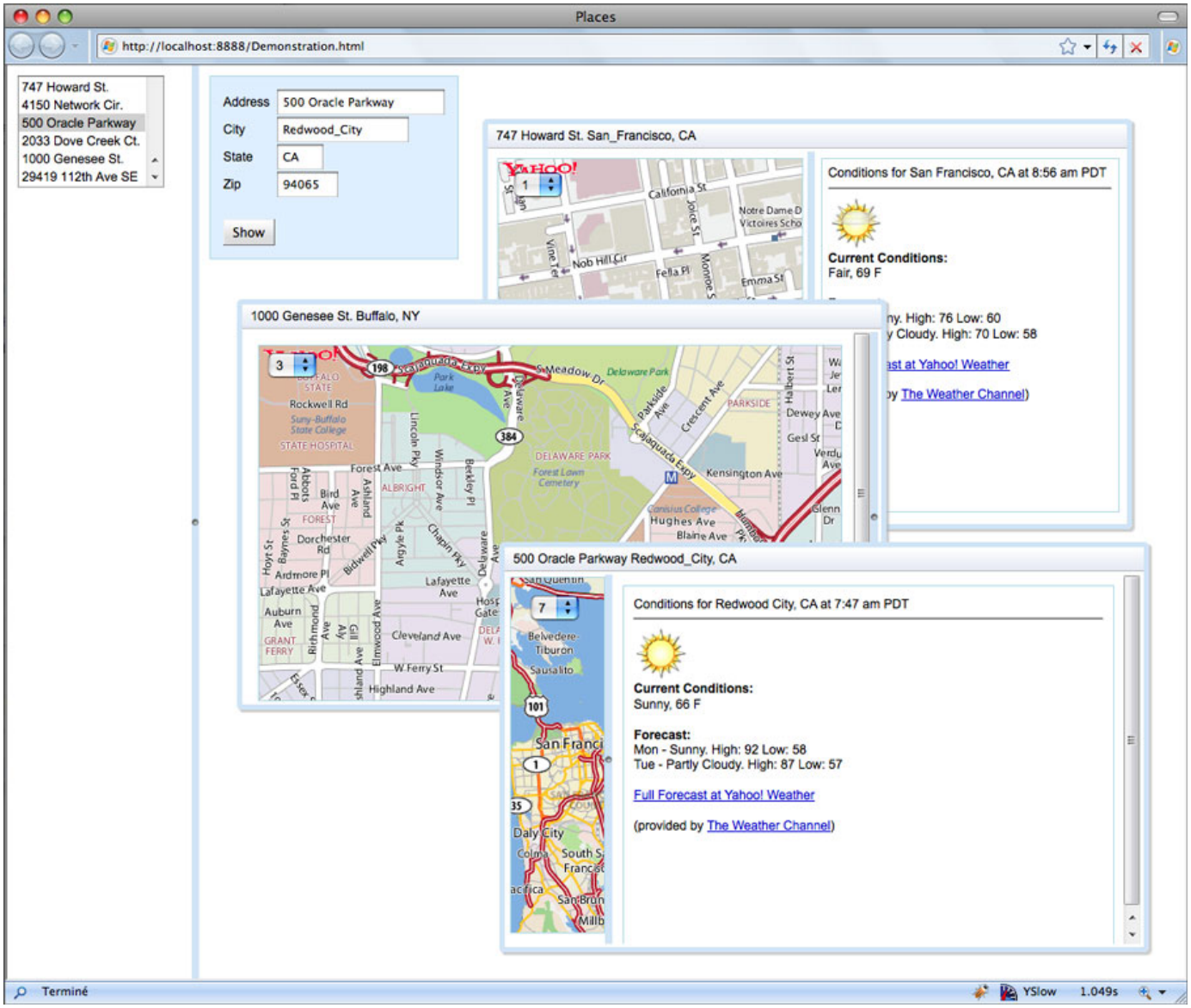


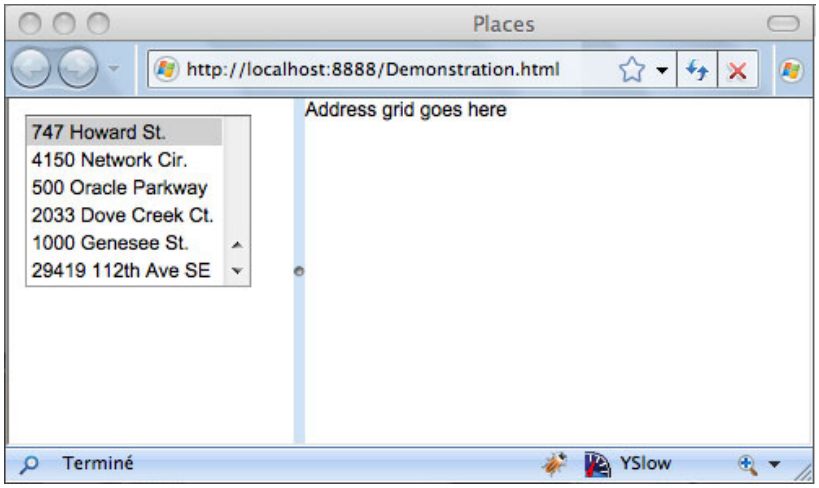
図 1 と図 2 の静的なスクリーンショットでは見て取れない点が 1 つあります。それは、地図はビューポート内にあるため、地図を分割パネルのなかでドラッグできるということです。地図を素早くドラッグすると (1 秒未満)、ビューポートはマウスがドラッグされた方向にビューを継続的に移動することによって、ビューをアニメーション化します。画像がビューポートの端に到達するとその端で跳ね返り、ビューポート内でマウスをクリックするまでビューのアニメーション化が続きます。この動作を実際に試してみるには、アプリケーションをダウンロードしてください。

ビューポートについては第 2 回で取り上げ、アニメーション化にタイマーを適用したり、イベント・プレビューを使用したりするなどの高度な GWT の手法について説明します。

ウィジェット

この記事の残りでは、Places アプリケーションを一から段階的に実装していきます。まず始めに取り掛かるのは、データベースから住所を取得して、リスト・ボックスに表示することです (図 3 を参照)。

図 3. ウィジェットとデータベース・アクセス



リスト 1 に、図 3 に表示されたアプリケーションのコードを記載します。

リスト 1. Places.java、テイク 1

```
package com.clarity.client;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.HorizontalSplitPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.ListBox;
import com.google.gwt.user.client.ui.RootPanel;

public class Places implements EntryPoint {
  final ListBox addresses = new ListBox();
  final ArrayList<Address> addressList = new ArrayList<Address>();
  final HorizontalSplitPanel hsp = new HorizontalSplitPanel();

  public void onModuleLoad() {
    hsp.add(addresses);
    hsp.add(new Label("Address grid goes here"));
    hsp.setSplitPosition("175px");

    getAddresses();

    RootPanel.get().add(hsp);
  }

  public void getAddresses() {
    // Instantiate the address service
    AddressServiceAsync as = (AddressServiceAsync) GWT
```

```
        .create(AddressService.class);

   // Use the address service to fetch addresses and populate the listbox
    ...
  }
}
```

すべての GWT アプリケーションは GWT モジュールであり、すべての GWT モジュールが `EntryPoint` インターフェースを実装します。このインターフェースは、`public void onModuleLoad()` というメソッドを実装します。このメソッドはデスクトップ・アプリケーションでの `main()` メソッドのようなものです。Places アプリケーションの場合には、この `onModuleLoad()` メソッドが住所のリスト・ボックスとラベルを左右に分割されたパネルに追加した後、データベースから住所を取得して、分割パネルをアプリケーションのルート・パネルに追加します (ルート・パネルはアプリケーションの HTML ページの本体を表します)。

ウィジェットを作成して構成するのは、GWT では簡単なことですが、値をデータベースから取得して表示するとなると、多少複雑になってきます。

RPC とデータベース統合

データベースから住所を取得するには、GWT の RPC を使用します。まず、`AddressServiceAsync` のインスタンスを `GWT.create()` メソッドでインスタンス化した後、このインスタンスを使って RPC を呼び出し、その結果を取り込みます (リスト 2 を参照)。

リスト 2. Places.java、テイク 2

```
package com.clarity.client;

public class Places implements EntryPoint {
  final ListBox addresses = new ListBox();
  final final ArrayList<Address> addressList = new ArrayList<Address>();
  final HorizontalSplitPanel hsp = new HorizontalSplitPanel();

  public void onModuleLoad() {
    hsp.add(addresses);
    hsp.add(new Label("Address grid goes here"));
    hsp.setSplitPosition("175px");

    getAddresses();

    RootPanel.get().add(hsp);
  }

  public void getAddresses() {
    // Instantiate the address service
    AddressServiceAsync as = (AddressServiceAsync) GWT
        .create(AddressService.class);

    as.getAddresses(new AsyncCallback<List<Address>>() {
      public void onFailure(Throwable caught) {
        GWT.log("Can't access database", caught);
      }

      public void onSuccess(List<Address> result) {
        Iterator<Address> it = result.iterator();

        while (it.hasNext()) {
          Address address = it.next();
          addresses.addItem(address.getAddress());
```

```
          addressList.add(address);
        }
        addresses.setVisibleItemCount(result.size());
      }
    });
}
```

GWT の RPC は、2 つのインターフェースによって定義されます。その 1 つはクライアントでユーザーが呼び出す非同期インターフェースで、もう 1 つはサーバーで GWT が呼び出すリモート・インターフェースです。この住所サービスでの非同期インターフェースとリモート・インターフェースに相当するのは、それぞれ `AddressServiceAsync`、`AddressService` です。リスト 2 では、リモート・インターフェースのクラスを渡して `GWT.create` を呼び出すと、GWT が非同期インターフェースのインスタンスを返します。

`AddressService` インターフェースのコードは、リスト 3 のとおりです。

リスト 3. AddressService.java

```
package com.clarity.client;

import java.util.List;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

@RemoteServiceRelativePath("address")
public interface AddressService extends RemoteService {
  public List<Address> getAddresses();
}
```

`AddressServiceAsync` インターフェースのコードは、リスト 4 のとおりです。

リスト 4. AddressServiceAsync.java

```
package com.clarity.client;

import java.util.List;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface AddressServiceAsync {
  public void getAddresses(AsyncCallback<List<Address>> callback);
}
```

クライアントで非同期インターフェースを使って `getAddresses()` メソッドを呼び出すと、GWT はサーバー上のリモート・インターフェースを使って対応するメソッドを呼び出します。リスト 2 に示されているように、GWT はサーバー上のメソッドが完了するのを待機してから非同期実装のコールバックを呼び出します。

最後に、Web アプリケーションのデプロイメント記述子にリモート・サーブレットを宣言します (リスト 5 を参照)。

リスト 5. WEB-INF/web.xml

```
package com.clarity.client;
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Servlets -->
  <servlet>
    <servlet-name>address</servlet-name>
    <servlet-class>com.clarity.server.AddressServiceImpl</servlet-class>
  </servlet>

  ...

  <servlet-mapping>
    <servlet-name>address</servlet-name>
    <url-pattern>/places/address</url-pattern>
  </servlet-mapping>

  ...
</web-app>
```

address というサーブレット名は、リスト 3 で @RemoteServiceRelativePath アノテーションに指定した値と同じであることに注目してください。この名前が一致することによって、リスト 3 で定義されたリモート RPC インターフェースは、リスト 6 で実装されたサーブレットと対応付けらます。

サーバー・サイドのデータベース・コード

`AddressServiceAsync` インターフェースと `AddressService` インターフェースは、どちらもクライアント上にあります。サーバー上には、POJO と Hibernate を使用して、データベースに格納された住所にアクセスするための何の変哲もないコードを実装しました。この場合の POJO は、リスト 6 のとおりです。

リスト 6. Address.java

```
package com.clarity.client;

import java.io.Serializable;

public class Address implements Serializable {
  private static final long serialVersionUID = 1L;
  private Long id;
  private String description, address, city, state, zip;

  public Address() {
    // you must implement a no-arg constructor
  }

  public Address(Long id, String address, String city) {
    this.address = address;
    this.city = city;
    this.id = id;
  }

  public String toString() {
    return address + " " + city + ", " + state + zip;
  }

  // Setters and getters for String properties are omitted in the interest of brevity
}
```

リスト 7 に、上記に対応する Hibernate コードを示します。

リスト 7. AddressServiceImpl.java

```
package com.clarity.server;

import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Session;

import com.clarity.client.AddressService;
import com.clarity.client.Address;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

@SuppressWarnings("unchecked")
public class AddressServiceImpl extends RemoteServiceServlet
  implements AddressService {
  private static final long serialVersionUID = 1L;

  public List<Address> getAddresses() {
    List<Address> addresses = null;
    try {
      Session session = HibernateUtil.getSessionFactory()
        .getCurrentSession();
      session.beginTransaction();
      addresses = (List<Address>)session.createQuery("from Address Address ")
        .list();
      session.getTransaction().commit();
    } catch (HibernateException e) {
      e.printStackTrace();
    }
    return addresses;
  }
}
```

リスト 7 に記載する `AddressServiceImpl` クラスが、住所サービスのリモート・インターフェースである `AddressService` を実装し、GWT の `RemoteServiceServlet` を継承します。`getAddresses()` メソッドは、データベースからすべての住所を返します。

RPC の非同期性

住所グリッドに、データベースから返された 1 番目の住所をを取り込む方法としては、以下のコードが考えられます。

```
public class Places implements EntryPoint {
final ListBox addresses = new ListBox();
final private AddressGrid addressGrid =
    new AddressGrid(addresses, addressList);
  ...

  public void onModuleLoad() {
    ...
    getAddresses();

    // this won't work
    addressGrid.setAddress(addressList.get(0));
    ...
    RootPanel.get().add(hsp);
  }
  ...
}
```

けれども、これでは上手くいきません。なぜなら、`getAddresses()` の呼び出しは非同期だからです。代わりに、以下のように RPC から制御が返されてから住所グリッドにデータを取り込まなければなりません。

```
public class Places implements EntryPoint {
  ...
  public void onModuleLoad() {
    ...
    getAddresses();
    ...
    RootPanel.get().add(hsp);
  }
  public void getAddresses() {
    AddressServiceAsync as = (AddressServiceAsync) GWT
        .create(AddressService.class);

    as.getAddresses(new AsyncCallback<List<Address>>() {
      ...
      public void onSuccess(List<Address> result) {
        ...
        addressGrid.setAddress(addressList.get(0));
      }
    });
  }
}
```

複合ウィジェット

これまでの手順で、いくつかのウィジェットを作成し、リスト・ボックスにデータベースから住所を取り込みました。今度は、住所グリッドをアプリケーションの分割パネルの右側に追加します (図 4 を参照)。

図 4. 住所グリッドを追加する

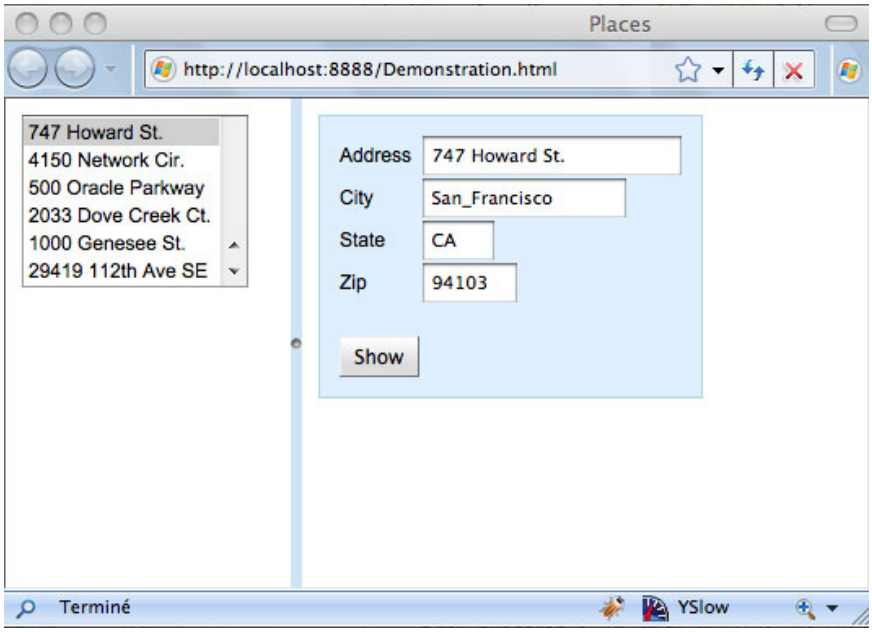


図 4 の右側に示されている住所グリッドは、複合ウィジェットです。リスト 8 に、`AddressGrid` クラスの実装を記載します。

リスト 8. AddressGrid.java

```
package com.clarity.client;

import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Grid;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.ListBox;
import com.google.gwt.user.client.ui.TextBox;

public class AddressGrid extends Composite {
  private Grid grid = new Grid(6,2);
  private Label streetAddressLabel = new Label("Address");
  private TextBox streetAddressTextBox = new TextBox();
  private Label cityLabel = new Label("City");
  private TextBox cityTextBox = new TextBox();
  private TextBox stateTextBox = new TextBox();
  private Label zipLabel = new Label("Zip");
  private TextBox zipTextBox = new TextBox();
  private Label stateLabel = new Label("State");
  private Button button = new Button();
  private Address address;

  public AddressGrid(final ListBox addresses, String buttonText,
    ClickHandler buttonClickHandler) {
    initWidget(grid);
    button.setText(buttonText);

    grid.addStyleName("addressGrid");

    stateTextBox.setVisibleLength(3);
    zipTextBox.setVisibleLength(5);
    cityTextBox.setVisibleLength(15);

    grid.setWidget(0, 0, streetAddressLabel); grid.setWidget(0, 1, streetAddressTextBox);
    grid.setWidget(1, 0, cityLabel);          grid.setWidget(1, 1, cityTextBox);
    grid.setWidget(2, 0, stateLabel);         grid.setWidget(2, 1, stateTextBox);
    grid.setWidget(3, 0, zipLabel);           grid.setWidget(3, 1, zipTextBox);
    grid.setWidget(5, 0, button);

    button.addClickHandler(buttonClickHandler);
  }

  void setAddress(Address address) {
    this.address = address;
    streetAddressTextBox.setText(address.getAddress());
    cityTextBox.setText(address.getCity());
    stateTextBox.setText(address.getState());
    zipTextBox.setText(address.getZip());
  }

  public Address getAddress() {
    return address;
  }
  public Button getButton() {
    return button;
  }
}
```

複合ウィジェットとは、その名前からわかるように、複数のウィジェットで構成されたウィジェットのことです。この住所グリッドを構成する `Grid` というウィジェットには、ラベル、テキスト・ボックス、ボタンが取り込まれます。ボタンに表示するテキストと、ボタンのクリック・ハンドラーは、住所グリッドの作成時に提供します。

住所グリッドは再利用可能な住所表現で、ここには何らかの機能を追加することができます。ここでは Places アプリケーションに、イベント・ハンドラーを利用して住所グリッドの機能を実装します。

イベントの処理

Places アプリケーションには 2 つのイベント・ハンドラーがあります。一方のイベント・ハンドラーは、ユーザーが住所リスト・ボックスから住所を選択すると、その住所を住所グリッドに取り込みます。もう一方は、ユーザーが住所グリッドの Show ボタンをクリックするとウィンドウを作成するイベント・ハンドラーです。リスト 9 に、この両方のイベント・ハンドラーを記載します。このリストは、リスト 1 の更新バージョンです。

リスト 9. Places.java、テイク 3

```
package com.clarity.client;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ChangeEvent;
import com.google.gwt.event.dom.client.ChangeHandler;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.HorizontalSplitPanel;
import com.google.gwt.user.client.ui.ListBox;
import com.google.gwt.user.client.ui.RootPanel;

public class Places implements EntryPoint {
  final ListBox addresses = new ListBox();
  final HorizontalSplitPanel hsp = new HorizontalSplitPanel();
  final ArrayList<Address> addressList = new ArrayList<Address>();
  final AddressGrid addressGrid = new AddressGrid("Show", new ShowPlaceHandler());

  public void onModuleLoad() {
    hsp.add(addresses);
    hsp.add(addressGrid);
    hsp.setSplitPosition("175px");

    getAddresses();

    addresses.addChangeHandler(new ChangeHandler() {
      public void onChange(ChangeEvent e) {
        addressGrid.setAddress(addressList.get(addresses.getSelectedIndex()));
      }
    });

    RootPanel.get().add(hsp);
  }

  public void getAddresses() {
    AddressServiceAsync as = (AddressServiceAsync) GWT
        .create(AddressService.class);

    as.getAddresses(new AsyncCallback<List<Address>>() {
      public void onFailure(Throwable caught) {
        GWT.log("Can't access database", caught);
      }
```

```
      public void onSuccess(List<Address> result) {
        Iterator<Address> it = result.iterator();

        while (it.hasNext()) {
          Address address = it.next();
          addresses.addItem(address.getAddress());
          addressList.add(address);
        }
        addresses.setVisibleItemCount(result.size());
        addressGrid.setAddress(addressList.get(0));
      }
    });
  }

  private class ShowPlaceHandler implements ClickHandler {
    private String[] urls;

    public void onClick(ClickEvent event) {
      final WeatherServiceAsync ws = (WeatherServiceAsync)
        GWT.create(WeatherService.class);

      final MapServiceAsync ms = (MapServiceAsync) GWT.create(MapService.class);
      final Address address = addressGrid.getAddress();

      addressGrid.getButton().setEnabled(false);

      ms.getMap(address.getAddress(), address.getCity(), address.getState(),
        new AsyncCallback<String[]>() {
          public void onFailure(Throwable arg0) {
            Window.alert(arg0.getMessage());
          }
          public void onSuccess(final String[] urls) {
            addresses.setEnabled(true);
            ws.getWeatherForZip(address.getZip(), true,
              new AsyncCallback<String>() {
              public void onFailure(Throwable arg0) {
                Window.alert(arg0.getMessage());
                done();
              }

              public void onSuccess(String weatherHTML) {
                PlaceDialog dialog = new PlaceDialog(addressGrid.getAddress(),
                    urls, weatherHTML);
                dialog.setPopupPosition(200, 200);
                dialog.show();
                done();
              }
            });
          }
        });
    }
    private void done() {
      addressGrid.getButton().setEnabled(true);
    }
  }
}
```

リスト 9 に記載した変更後の Places アプリケーションでは、変更ハンドラーが住所リスト・ボックスに追加されています。この変更ハンドラーが、ユーザーがリスト・ボックスから選択した住所を反映させるように住所グリッドを更新します。

また、このアプリケーションは `AddressGrid` のインスタンスを、テキストとグリッドのボタン用のクリック・ハンドラーでインスタンス化します。クリック・ハンドラーが実装される

`ShowPlaceHandler` クラスは、RPC を Yahoo! 地図および Yahoo! 天気情報の Web サービスにネストします。両方の RPC が完了すると、イベント・ハンドラーは、選択された住所の地図とその地域の天気 (予報を含む) が含まれるダイアログを作成します。`PlaceDialog` のインスタンスであるこのダイアログについては、第 2 回で説明します。

リスト 9 ではリスナーではなく、ハンドラーを使用している点に注目してください。GWT は当初、Swing でのイベント処理用にお馴染みのリスナー・パターンを実装していました。リスナー・インターフェースを実装する手段として、すべてのインターフェースのメソッドを実装する必要がないように空のメソッドを実装するアダプター・クラスを使用することもあったと思いますが、GWT 1.6 以降ではリスナーに代わってハンドラーが使用されています。ハンドラーはリスナーと非常によく似ていますが、ハンドラーが定義するメソッドは 1 つだけであること、そしてハンドラーには常に 1 つのイベント・オブジェクトが渡されるという点が異なります。単一のイベント・タイプのみをリッスンするハンドラーは、リスナーよりも粒度が細かくなります。さらに、リスナーにはイベントを起動したウィジェットが渡される一方、ハンドラーは、イベント固有の情報 (イベントを起動したウィジェットも含む) が含まれるイベント・オブジェクトから、より多くの情報を取得します。

この時点で、Places アプリケーションにはかなりの機能が備わりました。そこで必要となるのは、その機能を試すためのテストです。次のセクションでは、GWT で Ajax 呼び出しをテストする方法を説明します。

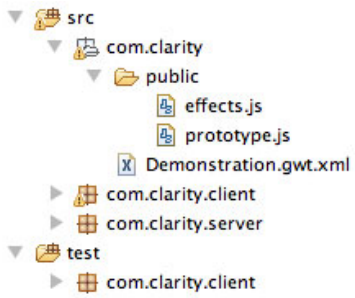
Ajax テスト

GWT ではアプリケーションを容易にテストできるように、JUnit との統合を行います。テストに取り掛かるのに最も簡単な方法は、GWT の `junitCreator` スクリプトを実行して、アプリケーション用のスケルトン・テストを作成することです。Places アプリケーションの場合には、以下のスクリプトを実行してスケルトン・テストを作成します。

```
junitCreator -junit /Developer/Java/Tools/junit-4.5/junit-4.5.jar
   -module com.clarity.Places
   -eclipse Places  com.clarity.client.PlacesTest
```

`junitCreator` スクリプトが作成するテスト・ディレクトリーとスケルトン・テストは図 5 のとおりです。

図 5. テスト用ディレクトリー構造



注意する点として、私はテスト・クラスをアプリケーションのユーザー・インターフェースの実装と同じパッケージに作成しましたが、ディレクトリーは変えています。こうすることによっ

て、テスト・クラスが UI クラスのメンバー変数にアクセスしやすくなると同時に、テスト・コードを UI コードから分離しておくことができます。テスト・クラスを UI と同じパッケージの別のディレクトリーに配置することが、一般的な GWT の慣例となっています。

GWT がスケルトン・テスト・クラスを作成したら、今度は実装を提供する番です。アプリケーションのインスタンスを作成し、プログラムによってウィジェットを操作して、その結果を検証します。リスト 10 に、Places アプリケーションがデータベースから住所を取得できることをテストする方法を示します。

リスト 10. RPC をテストする

```
package com.clarity.client;

import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.Timer;
import com.google.gwt.user.client.ui.ListBox;

public class PlacesTest extends GWTTestCase {

  public String getModuleName() {
    return "com.clarity.Places";
  }

  public void testGetAddresses() {
    Places demo = new Places();
    demo.getAddresses();
    final ListBox addresses = demo.addresses;

    new Timer() {
      public void run() {
        assert (addresses.getItemCount() == 6);
        assert (demo.addressList.size() == 6);
        finishTest();
      }
    }.schedule(10000);

    delayTestFinish(20000);
  }
}
```

リスト 10 ではタイマーを使用する一般的な GWT の慣例を、`finishTest()` および `delayFinishTest()` の呼び出しと併せて実装し、サーバーに対する非同期呼び出しをテストしています。`testGetAddresses()` メソッドで行われる処理は以下のとおりです。

1. Places アプリケーションのインスタンスを作成します。
2. アプリケーションの `getAddresses()` メソッドを呼び出します。
3. アプリケーションによって住所リスト・ボックスに 6 つの住所が取り込まれたことをアサートします。
4. アプリケーションによってアプリケーションの住所リストに 6 つの住所が取り込まれたことをアサートします。

`getAddresses()` の呼び出しは非同期であるため、この呼び出しが完了するのを待ってからでないと、住所リスト・ボックス内の項目数をチェックすることはできません。そこで、作成されると 10 秒間実行されるタイマーで項目数のチェックは行っています。10 秒あれば、アプリケーションが RPC を完了するには十分です。`delayTestFinish()` の呼び出しは GWT に対し、`finishTest()`

が呼び出されるまで 20 秒間待機するように指示します。20 秒以内に `finishTest()` が呼び出されなければ、テストはエラーによってタイムアウトになります。

次回の予告

ドラッグ・アンド・ドロップ、ウィンドウとダイアログ、そしてビューポートのような対話型ウィジェットなど、快適な機能が完備したデスクトップ風のアプリケーションを作成するには、GWT が最適です。Places アプリケーションは単純ながらも、GWT を使ってこのようなアプリケーションを実現できることを明らかにしています。今回の記事では、RPC とデータベース・アクセス、複合ウィジェットの実装、イベント処理、そして Ajax テストといった GWT の基本をおさえました。次の第 2 回では、イベントの受信、GWT モジュールの実装、イベント・プレビューの使用といった高度な GWT の機能について学びます。

ダウンロード

内容	ファイル名	サイズ
Source code for the places application	j-gwtfu-part1-code	690KB

著者について

David Geary

著者、講演者、コンサルタントとして活躍する David Geary は、Clarity Training, Inc. の社長です。彼はこの会社で、開発者に JSF および GWT (Google Web Toolkit) を使用した Web アプリケーションの実装を指導しています。JSTL 1.0 および JSF 1.0/2.0 Expert Group のメンバー、そして Sun の Web 開発者認定試験の共同制作者としての経験を持つ彼は、Apache Struts や Apache Shale などのオープンソース・プロジェクトにも貢献しています。彼の著書『グラフィック Java2 〈Vol.2〉 Swing 編』は Java 関連の本のなかでは史上に残るベスト・セラーの 1 つで、『Core JSF』(Cay Horstman との共著) は JSF 関連のベストセラー本となっています。コンファレンスやユーザー・グループで定期的に講演を行っている他、2003年以来 NFJS ツアーの常連で、Java University では講座を受け持っています。彼は JavaOne ロック・スターに 2 回選ばれました。