Google App Engine for Java, Part 2: Building the killer app

Build your own contact-management application in App Engine

Skill Level: Introductory

Rick Hightower (richardhightower@gmail.com)
CTO
Mammatus Inc.

11 Aug 2009

The whole point of a cloud platform like Google App Engine for Java™ is in being able to imagine, build, and deploy professional-quality killer apps that scale — without breaking the bank or driving yourself insane. In this second part of his three-part introduction to Google App Engine for Java, Rick Hightower takes you beyond the ready-made examples of Part 1 with a step-by-step guide to writing and deploying a simple contact-management application using App Engine for Java.

In Part 1 of this introduction to building scalable Java applications with App Engine for Java, you learned about the Eclipse tooling and infrastructure of Google's cloud computing platform (or PAAS) for Java developers. Examples in that article were pre-fab, so that you could concentrate on App Engine for Java's integration with Eclipse and quickly practice building and deploying different types of apps — namely one built using Google Web Toolkit (GWT) and a servlet-based application. This article builds on that foundation and also prepares you for more advanced programming exercises in Part 3 of this article.

The contact-management application you'll build allows a user to store basic contact information such as name, e-mail address, and telephone number. To create this application, you'll use the Eclipse GWT project-creation wizard.

From CRUD to contact

The first step to building a new application in App Engine for Java, as you know by now, is to launch the project-creation wizard in Eclipse. Once there, you can launch the GWT project-starter wizard to create a GWT project. (Part 1 of this article presents detailed instructions for creating a GWT project in App Engine for Java.)

For this exercise, you'll start with a simple CRUD application and later add real storage. For now, use a data access object (DAO) that has a mock implementation, as shown in Listing 1:

Listing 1. Interface for ContactDAO

```
package gaej.example.contact.server;

import java.util.List;

import gaej.example.contact.client.Contact;

public interface ContactDAO {
    void addContact(Contact contact);
    void removeContact(Contact contact);
    void updateContact(Contact contact);
    List<Contact> listContacts();
}
```

`ContactDAO` adds methods for adding a contact, removing a contact, updating a contact, and returning a list of all contacts. It is a very basic CRUD interface to manage contacts. The `Contact` class is your domain object, shown in Listing 2:

Listing 2. Contact domain object (gaej.example.contact.client.Contact)

```
package gaej.example.contact.client;

import java.io.Serializable;

public class Contact implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;
    private String email;
    private String phone;

    public Contact() {

    }

    public Contact(String name, String email, String phone) {
        super();
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }


}
```

For this first version of the application, you'll work with a mock object that stores the contacts in an in-memory collection, as shown in Listing 3:

Listing 3. Mock DAO class

```
package gaej.example.contact.server;

import gaej.example.contact.client.Contact;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class ContactDAOMock implements ContactDAO {

    Map<String, Contact> map = new LinkedHashMap<String, Contact>();

    {
        map.put("rhightower@mammatus.com",
          new Contact("Rick Hightower", "rhightower@mammatus.com", "520-555-1212"));
        map.put("scott@mammatus.com",
          new Contact("Scott Fauerbach", "scott@mammatus.com", "520-555-1213"));
        map.put("bob@mammatus.com",
          new Contact("Bob Dean", "bob@mammatus.com", "520-555-1214"));

    }

    public void addContact(Contact contact) {
        String email = contact.getEmail();
        map.put(email, contact);
    }

    public List<Contact> listContacts() {
        return Collections.unmodifiableList(new ArrayList<Contact>(map.values()));
    }

    public void removeContact(Contact contact) {
        map.remove(contact.getEmail());
    }

    public void updateContact(Contact contact) {
        map.put(contact.getEmail(), contact);
    }
```

```
}
```

Creating remote services

Your objective for now is to create a GWT GUI that allows you to use the DAO. It should use all the methods on the `ContactDAO` interface. The first step is to wrap the functionality of the DAO class (future versions will actually talk to data storage on the server side so it must be on the server) in a service, as shown in Listing 4:

Listing 4. ContactServiceImpl

```
package gaej.example.contact.server;

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

import gaej.example.contact.client.Contact;
import gaej.example.contact.client.ContactService;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;

public class ContactServiceImpl extends RemoteServiceServlet implements ContactService {
    private static final long serialVersionUID = 1L;
    private ContactDAO contactDAO = new ContactDAOMock();

    public void addContact(Contact contact) {
        contactDAO.addContact(contact);
    }

    public List<Contact> listContacts() {
        List<Contact> listContacts = contactDAO.listContacts();
        return new ArrayList<Contact> (listContacts);
    }

    public void removeContact(Contact contact) {
        contactDAO.removeContact(contact);

    }

    public void updateContact(Contact contact) {
        contactDAO.updateContact(contact);
    }


}
```

Notice that the `ContactServiceImpl` implements the `RemoteServiceServlet` and then defines methods to add a contact, list contacts, remove a contact, and update a contact. It delegates all of these operations to the `ContactDAOMock`. The `ContactServiceImpl` is just a wrapper around `ContactDAO` that exposes the `ContactDAO` functionality to the GWT GUI. The `ContactServiceImpl` is mapped in the web.xml file to the URI `/contactlist/contacts` in the web.xml file, as shown in Listing 5:

Listing 5. web.xml for ContactService

```
  <servlet>
    <servlet-name>contacts</servlet-name>
    <servlet-class>gaej.example.contact.server.ContactServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>contacts</servlet-name>
    <url-pattern>/contactlist/contacts</url-pattern>
  </servlet-mapping>
```

To allow the GUI front end to access this service, you need to define both a remote service interface and an asynchronous remote service interface, as shown in Listings 6 and 7:

Listing 6. ContactService

```
package gaej.example.contact.client;

import java.util.List;

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

@RemoteServiceRelativePath("contacts")
public interface ContactService extends RemoteService {
    List<Contact> listContacts();
    void addContact(Contact contact);
    void removeContact(Contact contact);
    void updateContact(Contact contact);
}
```

Listing 7. ContactServiceAsync

```
package gaej.example.contact.client;

import java.util.List;

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

public interface ContactServiceAsync  {
    void listContacts(AsyncCallback<List <Contact>> callback);
    void addContact(Contact contact, AsyncCallback<Void> callback);
    void removeContact(Contact contact, AsyncCallback<Void> callback);
    void updateContact(Contact contact, AsyncCallback<Void> callback);
}
```

Notice that the `ContactService` implements the `RemoteService` interface and defines a `@RemoteServiceRelativePath` that specifies a relative path of "contacts." The relative path corresponds to the path you defined for the service in the web.xml file (they must match). The `ContactServiceAsync` has callback objects so that the GWT GUI can be notified of the calls from the server without blocking other client activity.

Cutting through spaghetti code

I am not a big fan of spaghetti code and I avoid writing it whenever I can. An example of spaghetti code would be a group of anonymous inner class whose methods define anonymous inner classes. These inner classes, in turn, do callbacks that call methods, which are then defined inline in an inner class. *Ugh!* Frankly, I can't read or understand code that tangled up, even when it's my own! So, to flatten things out a bit, I would suggest breaking up the GWT GUI into three pieces:

- `ContactListEntryPoint`
- `ContactServiceDelegate`
- `ContactListGUI`

The `ContactListEntryPoint` is the main entry point; it does GUI event wiring. The `ContactServiceDelegate` wraps the `ContactService` functionality and hides the inner class callback wiring. The `ContactListGUI` manages all of the GUI components and handles events from the `GUI` and the `Service`. The `ContactListGUI` uses the `ContactServiceDelegate` to make requests of the `ContactService`.

The ContactList.gwt.xml file (a resource under gaej.example.contact) specifies `ContactListEntryPoint` as the main entry point for the application using the `entry-point` element shown in Listing 8:

Listing 8. ContactList.gwt.xml

```
<entry-point class='gaej.example.contact.client.ContactListEntryPoint'/>
```

The `ContactListEntryPoint` class implements the `EntryPoint` interface from GWT (`com.google.gwt.core.client.EntryPoint`), signifying that this class will be called to initialize the GUI. `ContactListEntryPoint` doesn't do much. It creates an instance of `ContactListGUI` and an instance of `ContactServiceDelegate`, and then lets them know about each other so that they can collaborate. The `ContactListEntryPoint` then does GUI event wiring. `ContactListEntryPoint` is shown in Listing 9:

Listing 9. ContactListEntryPoint

```
package gaej.example.contact.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.user.client.ui.HTMLTable.Cell;

/**
 * Entry point classes define onModuleLoad().
 */
public class ContactListEntryPoint implements EntryPoint {
```

```
    private ContactListGUI gui;
    private ContactServiceDelegate delegate;

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {

        gui = new ContactListGUI();
        delegate = new ContactServiceDelegate();
        gui.contactService = delegate;
        delegate.gui = gui;
        gui.init();
        delegate.listContacts();
        wireGUIEvents();


    }

    private void wireGUIEvents() {
        gui.contactGrid.addClickHandler(new ClickHandler(){
            public void onClick(ClickEvent event) {
                 Cell cellForEvent = gui.contactGrid.getCellForEvent(event);
                 gui.gui_eventContactGridClicked(cellForEvent);
            }});

        gui.addButton.addClickHandler(new ClickHandler(){
            public void onClick(ClickEvent event) {
                gui.gui_eventAddButtonClicked();
            }});

        gui.updateButton.addClickHandler(new ClickHandler(){
            public void onClick(ClickEvent event) {
                gui.gui_eventUpdateButtonClicked();
            }});

        gui.addNewButton.addClickHandler(new ClickHandler(){
            public void onClick(ClickEvent event) {
                gui.gui_eventAddNewButtonClicked();

            }});

    }
}
```

Notice that the `ContactListEntryPoint` wires events for `addButton`, `updateButton`, `contactGrid`, and the `addNewButton`. It does this by registering anonymous inner classes that implement the listener interface for widget events. It's a very similar technique to event-handling in Swing. The widget events are from widgets created by the GUI (`ContactListGUI`), which I'll discuss in a bit. Notice that the GUI class has `gui_eventXXX` methods to respond to GUI events.

The `ContactListGUI` creates the GUI widgets and responds to events from them. `ContactListGUI` translates the GUI events into actions the user wants to perform on the `ContactsService`. `ContactListGUI` uses the `ContactServiceDelegate` to invoke methods on the `ContactService`. The `ContactServiceDelegate` creates an asynchronous interface to the `ContactService` and uses it to make asynchronous Ajax calls. The `ContactServiceDelegate` notifies the `ContactListGUI` of events (success or failure returns) from the service. The `ContactServiceDelegate` is shown in

Listing 10:

Listing 10. ContactServiceDelegatepackage gaej.example.contact.client;

```
import java.util.List;

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

public class ContactServiceDelegate {
    private ContactServiceAsync contactService = GWT.create(ContactService.class);
    ContactListGUI gui;

    void listContacts() {
        contactService.listContacts(new AsyncCallback<List<Contact>> () {
                    public void onFailure(Throwable caught) {
                        gui.service_eventListContactsFailed(caught);
                    }

                    public void onSuccess(List<Contact> result) {
                        gui.service_eventListRetrievedFromService(result);

                    }
        }//end of inner class
        );//end of listContacts method call.
    }

    void addContact(final Contact contact) {
        contactService.addContact(contact, new AsyncCallback<Void> () {
            public void onFailure(Throwable caught) {
                gui.service_eventAddContactFailed(caught);
            }

            public void onSuccess(Void result) {
                gui.service_eventAddContactSuccessful();
            }
        }//end of inner class
        );//end of addContact method call.
    }

    void updateContact(final Contact contact) {
        contactService.updateContact(contact, new AsyncCallback<Void> () {
            public void onFailure(Throwable caught) {
                gui.service_eventUpdateContactFailed(caught);
            }

            public void onSuccess(Void result) {
                gui.service_eventUpdateSuccessful();
            }
        }//end of inner class
        );//end of updateContact method call.
    }

    void removeContact(final Contact contact) {
        contactService.removeContact(contact, new AsyncCallback<Void> () {
            public void onFailure(Throwable caught) {
                gui.service_eventRemoveContactFailed(caught);
            }

            public void onSuccess(Void result) {
                gui.service_eventRemoveContactSuccessful();
            }
        }//end of inner class
        );//end of updateContact method call.
    }
```

```
}
```

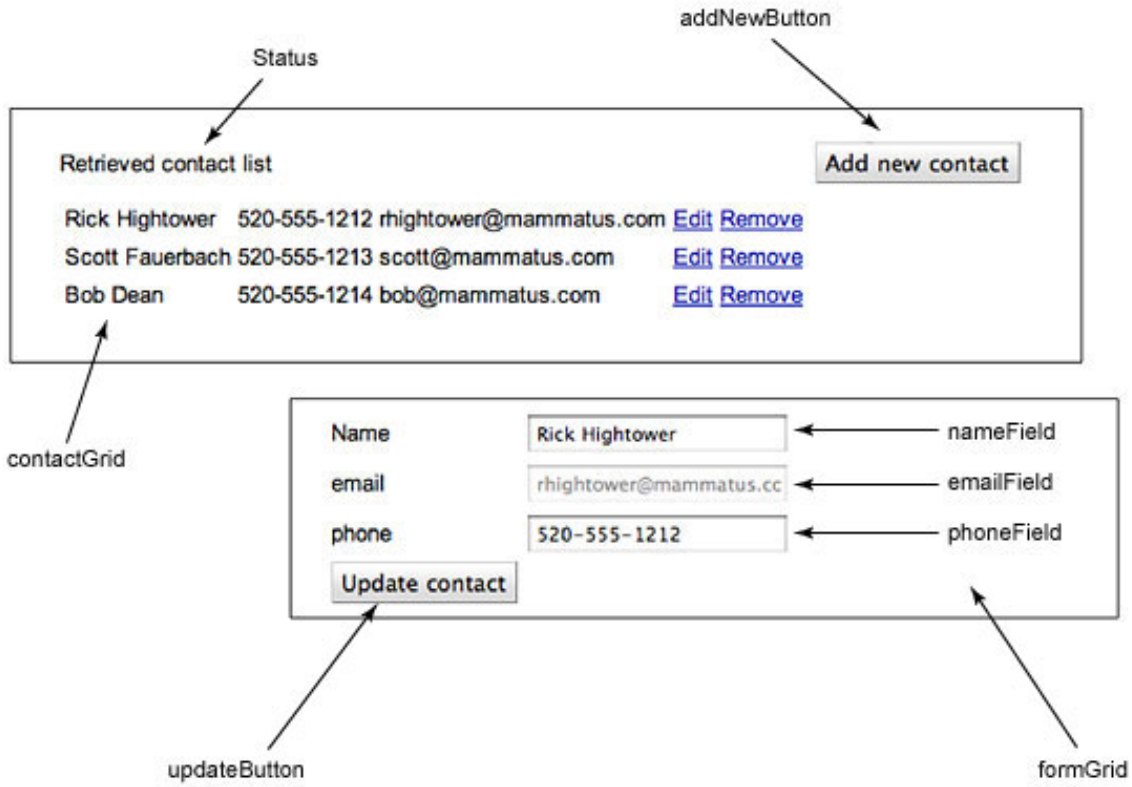
Notice that the `ContactServiceDelegate` notifies the `ContactListGUI` of service events via methods that begin in `service_eventXXX`. As previously mentioned, one of my goals when writing `ContactListGUI` was to avoid nested inner classes and create a relatively flat GUI class (one that I could easily read and work with later). The `ContactListGUI` is only 186 lines long and is fairly straightforward. The `ContactListGUI` manages nine GUI widgets and also collaborates with `ContactServiceDelegate` to manage a CRUD listing, as shown in Listing 11:

Listing 11. ContactListGUI at work

```
package gaej.example.contact.client;

import java.util.List;

import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Grid;
import com.google.gwt.user.client.ui.Hyperlink;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.HTMLTable.Cell;

public class ContactListGUI {
    /* Constants. */
    private static final String CONTACT_LISTING_ROOT_PANEL = "contactListing";
    private static final String CONTACT_FORM_ROOT_PANEL = "contactForm";
    private static final String CONTACT_STATUS_ROOT_PANEL = "contactStatus";
    private static final String CONTACT_TOOL_BAR_ROOT_PANEL = "contactToolBar";
    private static final int EDIT_LINK = 3;
    private static final int REMOVE_LINK = 4;

    /* GUI Widgets */
    protected Button addButton;
    protected Button updateButton;
    protected Button addNewButton;
    protected TextBox nameField;
    protected TextBox emailField;
    protected TextBox phoneField;
    protected Label status;
    protected Grid contactGrid;
    protected Grid formGrid;

    /* Data model */
    private List<Contact> contacts;
    private Contact currentContact;
    protected ContactServiceDelegate contactService;
```

Notice that the `ContactListGUI` keeps track of the current contact loaded in the form (`currentContact`) and the list of contacts in the listing (`contacts`). Figure 1 shows how the widgets correspond to the GUI that is created:

Figure 1. Widgets active in the contact management GUI



Listing 12 shows how `ContactListGUI` creates widgets and a contact form and then places the widgets in the form:

Listing 12. ContactListGUI creates and places widgets

```
public class ContactListGUI {
    /* Constants. */
    private static final String CONTACT_LISTING_ROOT_PANEL = "contactListing";
    private static final String CONTACT_FORM_ROOT_PANEL = "contactForm";
    private static final String CONTACT_STATUS_ROOT_PANEL = "contactStatus";
    private static final String CONTACT_TOOL_BAR_ROOT_PANEL = "contactToolBar";
    ...
    public void init() {
        addButton = new Button("Add new contact");
        addNewButton = new Button("Add new contact");
        updateButton = new Button("Update contact");
        nameField = new TextBox();
        emailField = new TextBox();
        phoneField = new TextBox();
        status = new Label();
        contactGrid = new Grid(2,5);

        buildForm();
        placeWidgets();
    }

    private void buildForm() {
        formGrid = new Grid(4,3);
```

```
        formGrid.setVisible(false);

        formGrid.setWidget(0, 0, new Label("Name"));
        formGrid.setWidget(0, 1, nameField);

        formGrid.setWidget(1, 0, new Label("email"));
        formGrid.setWidget(1, 1, emailField);

        formGrid.setWidget(2, 0, new Label("phone"));
        formGrid.setWidget(2, 1, phoneField);

        formGrid.setWidget(3, 0, updateButton);
        formGrid.setWidget(3, 1, addButton);

    }

    private void placeWidgets() {
        RootPanel.get(CONTACT_LISTING_ROOT_PANEL).add(contactGrid);
        RootPanel.get(CONTACT_FORM_ROOT_PANEL).add(formGrid);
        RootPanel.get(CONTACT_STATUS_ROOT_PANEL).add(status);
        RootPanel.get(CONTACT_TOOL_BAR_ROOT_PANEL).add(addNewButton);
    }
```

The `ContactListGUI` `init` method is called by the `ContactListEntryPoint.onModuleLoad` method. The `init` method calls the `buildForm` method to create a new form grid and populate it with fields to edit contact data. The `init` method then calls the `placeWidgets` method, which places the `contactGrid`, `formGrid`, `status`, and `addNewButton` widgets into slots defined in the HTML page that hosts this GUI app defined in Listing 13:

Listing 13. ContactList.html defines slots for widgets

```
<h1>Contact List Example</h1>

    <table align="center">
      <tr>
        <td id="contactStatus"></td> <td id="contactToolBar"></td>
      </tr>
      <tr>
        <td id="contactForm"></td>
      </tr>
      <tr>
        <td id="contactListing"></td>
      </tr>
    </table>
```

The constants (such as `CONTACT_LISTING_ROOT_PANEL="contactListing"`) correspond to IDs of elements (like `id="contactListing"`) defined in the HTML page. This allows a page designer to have more control over the layout of application widgets.

With the basic application built, let's walk through a couple of common usage scenarios.

Showing a listing on page load

When the contact management app's page first loads it calls the `ContactListEntryPoint`'s `onModuleLoad` method. `onModuleLoad` calls the `ContactServiceDelegate`'s `listContacts` method, which calls the service's `listContact` method asynchronously. When the `listContact` method returns, an anonymous inner class defined in `ContactServiceDelegate` calls the service-event handler method called `service_eventListRetrievedFromService`, shown in Listing 14:

Listing 14. Calling the listContact event handler

```
public class ContactListGUI {
  ...
  public void service_eventListRetrievedFromService(List<Contact> result) {
        status.setText("Retrieved contact list");
        this.contacts = result;
        this.contactGrid.clear();
        this.contactGrid.resizeRows(this.contacts.size());
        int row = 0;

        for (Contact contact : result) {
            this.contactGrid.setWidget(row, 0, new Label(contact.getName()));
            this.contactGrid.setWidget(row, 1, new Label (contact.getPhone()));
            this.contactGrid.setWidget(row, 2, new Label (contact.getEmail()));
            this.contactGrid.setWidget(row, EDIT_LINK, new Hyperlink("Edit", null));
            this.contactGrid.setWidget(row, REMOVE_LINK, new Hyperlink("Remove", null));
            row ++;
        }
    }
```

The `service_eventListRetrievedFromService` event-handler method stores the contact list sent by the server. Then it clears the `contactGrid` that displays the contact listing. It resizes the number of rows to match the size of the contact list returned from the server. It then iterates through the contact list, placing name, telephone, and e-mail data for each contact into the first three columns of each row. It also provides an Edit link and a Remove link for each contact, enabling users to easily remove and edit contacts.

User edits an existing contact

When a user clicks on an Edit link from the contacts listing, the `gui_eventContactGridClicked` is called, as shown in Listing 15:

Listing 15. ContactListGUI's gui_eventContactGridClicked event handler method

```
public class ContactListGUI {
  ...

    public void gui_eventContactGridClicked(Cell cellClicked) {
         int row = cellClicked.getRowIndex();
         int col = cellClicked.getCellIndex();

         Contact contact = this.contacts.get(row);
```

```
          this.status.setText("Name was " + contact.getName() + " clicked ");

          if (col==EDIT_LINK) {
              this.addNewButton.setVisible(false);
              this.updateButton.setVisible(true);
              this.addButton.setVisible(false);
              this.emailField.setReadOnly(true);
              loadForm(contact);
          } else if (col==REMOVE_LINK) {
              this.contactService.removeContact(contact);
          }
      }
     ...
      private void loadForm(Contact contact) {
          this.formGrid.setVisible(true);
          currentContact = contact;
          this.emailField.setText(contact.getEmail());
          this.phoneField.setText(contact.getPhone());
          this.nameField.setText(contact.getName());
      }
```

The `gui_eventContactGridClicked` method must determine whether Edit link or Remove link has been clicked. It does this by finding out which column was clicked. It then hides the `addNewButton` and the `addButton` and makes the `updateButton` visible. The `updateButton` displays in the `formGrid` and allows the user to send the update information back to `ContactService`. It also makes the `emailField` read only so that the user can't edit the e-mail field. Next, the `gui_eventContactGridClicked` calls `loadForm` (shown in Listing 15), which sets the `formGrid` to *visible*, sets the contact being edited, and then copies the contact properties into `emailField`, `phoneField`, and `nameField` widgets.

When the user clicks the `updateButton`, the `gui_eventUpdateButtonClicked` event-handler method is called, as shown in Listing 16. This method makes the `addNewButton` visible (so the user can add new contacts) and hides the `formGrid`. It then calls `copyFieldDateToContact`, which copies the text from the `emailField`, `phoneField`, and `nameField` widgets back into the properties of the `currentContact`. It then calls the `ContactServiceDelegate` `updateContact` method to pass the newly updated contact back to the service.

Listing 16. ContactListGUI's gui_eventUpdateButtonClicked event-handler method

```
public class ContactListGUI {
  ...

    public void gui_eventUpdateButtonClicked() {
        addNewButton.setVisible(true);
        formGrid.setVisible(false);
        copyFieldDateToContact();
        this.contactService.updateContact(currentContact);
    }
    private void copyFieldDateToContact() {
        currentContact.setEmail(emailField.getText());
        currentContact.setName(nameField.getText());
        currentContact.setPhone(phoneField.getText());
```

```
    }
```

These two scenarios should give you an idea of how the application works, as well as how it draws upon the infrastructure provided by App Engine for Java. The complete code for the `ContactListGUI` is shown in Listing 17:

Listing 17. Complete code for ContactListGUI

```
package gaej.example.contact.client;

import java.util.List;

import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Grid;
import com.google.gwt.user.client.ui.Hyperlink;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.HTMLTable.Cell;

public class ContactListGUI {
    /* Constants. */
    private static final String CONTACT_LISTING_ROOT_PANEL = "contactListing";
    private static final String CONTACT_FORM_ROOT_PANEL = "contactForm";
    private static final String CONTACT_STATUS_ROOT_PANEL = "contactStatus";
    private static final String CONTACT_TOOL_BAR_ROOT_PANEL = "contactToolBar";
    private static final int EDIT_LINK = 3;
    private static final int REMOVE_LINK = 4;

    /* GUI Widgets */
    protected Button addButton;
    protected Button updateButton;
    protected Button addNewButton;
    protected TextBox nameField;
    protected TextBox emailField;
    protected TextBox phoneField;
    protected Label status;
    protected Grid contactGrid;
    protected Grid formGrid;

    /* Data model */
    private List<Contact> contacts;
    private Contact currentContact;
    protected ContactServiceDelegate contactService;

    public void init() {
        addButton = new Button("Add new contact");
        addNewButton = new Button("Add new contact");
        updateButton = new Button("Update contact");
        nameField = new TextBox();
        emailField = new TextBox();
        phoneField = new TextBox();
        status = new Label();
        contactGrid = new Grid(2,5);

        buildForm();
        placeWidgets();
    }

    private void buildForm() {
        formGrid = new Grid(4,3);
        formGrid.setVisible(false);

        formGrid.setWidget(0, 0, new Label("Name"));
```

```
        formGrid.setWidget(0, 1, nameField);

        formGrid.setWidget(1, 0, new Label("email"));
        formGrid.setWidget(1, 1, emailField);

        formGrid.setWidget(2, 0, new Label("phone"));
        formGrid.setWidget(2, 1, phoneField);

        formGrid.setWidget(3, 0, updateButton);
        formGrid.setWidget(3, 1, addButton);

    }

    private void placeWidgets() {
        RootPanel.get(CONTACT_LISTING_ROOT_PANEL).add(contactGrid);
        RootPanel.get(CONTACT_FORM_ROOT_PANEL).add(formGrid);
        RootPanel.get(CONTACT_STATUS_ROOT_PANEL).add(status);
        RootPanel.get(CONTACT_TOOL_BAR_ROOT_PANEL).add(addNewButton);
    }

    private void loadForm(Contact contact) {
        this.formGrid.setVisible(true);
        currentContact = contact;
        this.emailField.setText(contact.getEmail());
        this.phoneField.setText(contact.getPhone());
        this.nameField.setText(contact.getName());
    }


    private void copyFieldDateToContact() {
        currentContact.setEmail(emailField.getText());
        currentContact.setName(nameField.getText());
        currentContact.setPhone(phoneField.getText());
    }

    public void gui_eventContactGridClicked(Cell cellClicked) {
         int row = cellClicked.getRowIndex();
         int col = cellClicked.getCellIndex();

         Contact contact = this.contacts.get(row);
         this.status.setText("Name was " + contact.getName() + " clicked ");

         if (col==EDIT_LINK) {
             this.addNewButton.setVisible(false);
             this.updateButton.setVisible(true);
             this.addButton.setVisible(false);
             this.emailField.setReadOnly(true);
             loadForm(contact);
         } else if (col==REMOVE_LINK) {
             this.contactService.removeContact(contact);
         }
    }


    public void gui_eventAddButtonClicked() {
        addNewButton.setVisible(true);
        formGrid.setVisible(false);
        copyFieldDateToContact();
        this.phoneField.getText();
        this.contactService.addContact(currentContact);
    }

    public void gui_eventUpdateButtonClicked() {
        addNewButton.setVisible(true);
        formGrid.setVisible(false);
        copyFieldDateToContact();
        this.contactService.updateContact(currentContact);
    }
```

```
    public void gui_eventAddNewButtonClicked() {
        this.addNewButton.setVisible(false);
        this.updateButton.setVisible(false);
        this.addButton.setVisible(true);
        this.emailField.setReadOnly(false);
        loadForm(new Contact());
    }


    public void service_eventListRetrievedFromService(List<Contact> result) {
        status.setText("Retrieved contact list");
        this.contacts = result;
        this.contactGrid.clear();
        this.contactGrid.resizeRows(this.contacts.size());
        int row = 0;

        for (Contact contact : result) {
            this.contactGrid.setWidget(row, 0, new Label(contact.getName()));
            this.contactGrid.setWidget(row, 1, new Label (contact.getPhone()));
            this.contactGrid.setWidget(row, 2, new Label (contact.getEmail()));
            this.contactGrid.setWidget(row, EDIT_LINK, new Hyperlink("Edit", null));
            this.contactGrid.setWidget(row, REMOVE_LINK, new Hyperlink("Remove", null));
            row ++;
        }
    }

    public void service_eventAddContactSuccessful() {
        status.setText("Contact was successfully added");
        this.contactService.listContacts();
    }

    public void service_eventUpdateSuccessful() {
        status.setText("Contact was successfully updated");
        this.contactService.listContacts();
    }
    public void service_eventRemoveContactSuccessful() {
        status.setText("Contact was removed");
        this.contactService.listContacts();

    }

    public void service_eventUpdateContactFailed(Throwable caught) {
        status.setText("Update contact failed");
    }

    public void service_eventAddContactFailed(Throwable caught) {
        status.setText("Unable to update contact");
    }

    public void service_eventRemoveContactFailed(Throwable caught) {
        status.setText("Remove contact failed");
    }

    public void service_eventListContactsFailed(Throwable caught) {
        status.setText("Unable to get contact list");

    }

}
```

Conclusion

This second part of a three-part introduction to Google App Engine for Java has introduced you to the process of creating a custom GWT application using the

Eclipse plugin tools for App Engine for Java. In building the simple contact-management application, you've learned how to:

- Build remote services that work asynchronously
- Organize GUI code to avoid nested inner class declarations
- Utilize GWT to implement functionality for two key use cases

Stay tuned for Part 3 of this article, where you will refine the contact-management application and add support for persisting `Contact` objects with the App Engine for Java datastore facilities.

Resources

Learn

- "Google App Engine for Java, Part 1: Rev it up!" (Rick Hightower, developerWorks, August 2009): Get started with the Eclipse plugins for App Engine for Java and learn how to quickly build simple applications that scale.
- "Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 1: The fancy front end" (Noel Rappin, developerWorks, October 2006): Launches a four-part tutorial introduction to Ajax programming with Google Web Toolkit.
- "Connecting to the cloud, Part 1: Leverage the cloud in applications" (Mark O'Neill, developerWorks, April 2009): A three-part overview of cloud computing platforms from the major vendors: Amazon, Google, Microsoft, and SalesForce.com.
- Google App Engine homepage: Learn more about App Engine.
- Google App Engine Java docs: More about Java development with App Engine for Java.
- Will it play in App Engine for Java?: Find out what standard Java APIs and frameworks are compatible with App Engine for Java.
- What's BigTable?: Read the Google research publication to find out.
- "GWT: The most important announcement at JavaOne?" (Rick Hightower, Java Developers Journal, June 2006): More about why GWT is also a step forward for Java application development.

Get products and technologies

- Google Plugin for Eclipse: Download the plugins to get started.
- Google App Engine account: You'll need one to deploy your killer app using the Google App Engine infrastructure.

Discuss

- App Engine for Java discussion group: Subscribe to this group to give and receive help and advice as you learn about App Engine for Java.
- Get involved in the My developerWorks community.

About the author

Rick Hightower

Rick Hightower serves as chief technology officer for Mammatus Inc., a training company that specializes in cloud computing, GWT, Java EE, Spring, and Hibernate development. He is coauthor of the popular book *Java Tools for Extreme Programming* and author of the first edition of *Struts Live* — the number-one download on TheServerSide.com for many years. He has also written articles and tutorials for IBM developerWorks and is on the editorial board for Java Developer's Journal, as well as a frequent contributor to the Java and Groovy topics on DZone.