The Google Web Toolkit (GWT): The Model-View-Presenter (MVP) Architecture – General Approach

(GWT 2.5 Version)

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/gwt.html

© 2013 Marty Hall & Yaakov Chaikin

Topics in This Section

- **Motivation**
- **Advantages/Disadvantages**
- **Main components of MVP (MVPA?)**
  - Model
  - View
  - Presenter
  - AppController
- **Need for EventBus**
- **Testing in MVP**

MVP Overview

Motivation

- **Large scale app development needs a strategy**
  - Multiple developers working simultaneously
  - Working on the same code base
  - When components do too much, trouble is not far behind
  - No strategy = spaghetti code = maintenance nightmare
    - In any GUI-based development, the nightmare will start *even before maintenance* stage
- **MVP is a design pattern**
  - Like any design pattern, it tries to accomplish what we all learned in CS101:
    - DO ONE THING AND DO IT WELL

Motivation (Continued)

- **Compartmentalize areas of project responsibility**
  - There are other approaches, like MVC, but MVP suits GWT development best
- **MVP has two major goals**
  - Decouple development to allow multiple developers to work simultaneously
    - Separates functionality into components that logically make sense
  - Allow testing of core components without a browser
    - Write fast JRE-based unit tests, not slow GWTTestCase-based tests

Advantages

- **Once you understand the pattern, everything in the app makes sense and falls into place**
  - Cookie-cutter approach: these are the standard interfaces I need to implement, etc.
  - Heavier reliance on interface-based design makes development easier and less error prone
- **Easier to develop multiple screens simultaneously**
- **Easier to test core functionality**
  - Mock out the rest with easymock library

Disadvantages

- **Big learning curve**
  - Have to change your way of thinking
  - Hard to get used to at first
  - Without clear understanding what a component should do, and most importantly what it *should not* do, mistakes creep in and some rework is needed
- **Not easy to set up at first**
  - But, once it is set up, it's a breeze to use
    - Assuming you understand the components in play
- **Quick & easy things previously done in one class now require multiple classes**
  - Only *seems* like a bad thing, it's actually a good thing
  - But, if it's a quick prototype app, it's not worth it

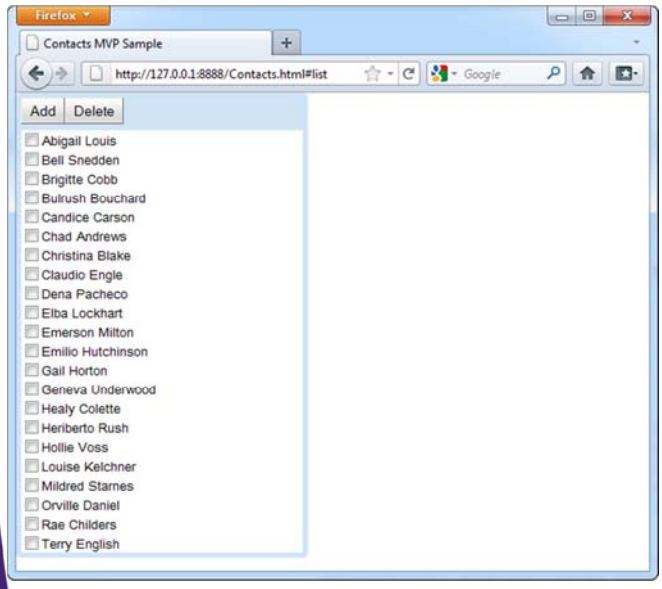
Main Components of MVP (MVPA?)

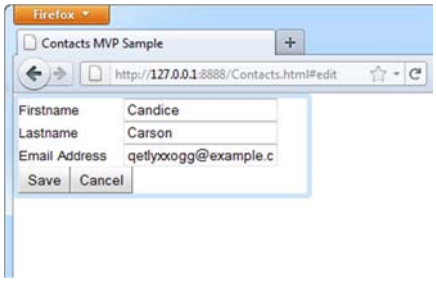
- **Model**
  - Responsible for holding raw data
  - No business logic allowed (just like any model really)
- **View**
  - Responsible for displaying data
  - No business logic allowed
  - Implements a Display interface
    - Usually defined by the presenter, but not always
- **Presenter**
  - Responsible for getting the data, driving the view, listening for GUI events, implements business logic
- **AppController**
  - Uber presenter: the starting place and the main Presenter
  - Responsible for registering app-wide event handlers, e.g., history events
  - Wiring up all components together

Example

Example Setup

- **Using Google's Contacts app**





Example: Model

Model

- **Business object**
- **In our example:**
  - Contact – representation of a contact within our app
    - This is the server-side object with full data
  - ContactDetails
    - Lightweight client-side representation of a Contact
      - Fewer fields (just for the list)
      - Maintains the 'id' – that's what we'll use to get the full contact info
      - Naming of Contact vs. ContactDetails is a bit backwards. Should be Client and LightweightClient.
  - In real apps, it's common to *only* send lightweight objects to the GUI
    - Reduces amount of data over the wire
    - Makes the retrieval faster

Example: Model (Contact.java)

```
package com.google.gwt.sample.contacts.shared;
...
public class Contact implements Serializable {
  private String id, firstName, lastName, emailAddress;

  ... // Constructors

  public ContactDetails getLightWeightContact() {
    return new ContactDetails(id, getFullName());
  }

  public String getId() { return id; }
  public void setId(String id) { this.id = id; }
  ... // Other getters and setters
  public String getFullName() {
   return firstName + " " + lastName;
  }
}
```

ContactDetails is a lightweight object that will be sent to the client. It's all we need for our view.

Example: Model (ContactDetails.java)

```
package com.google.gwt.sample.contacts.shared;
...
public class ContactDetails implements Serializable {
  private String id;
  private String displayName;

  public ContactDetails() {
    new ContactDetails("0", "");
  }

  public ContactDetails(String id, String displayName) {
    this.id = id;
    this.displayName = displayName;
  }

  public String getId() { return id; }
  public void setId(String id) { this.id = id; }
  ...
}
```

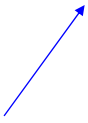

Example: Presenter

Presenter

- **Contains all the logic**
  - Implements a very simple Presenter interface
  - Attaches its view to whatever widget is passed to it
  - Drives the view via presenter defined Display interface
  - Does NOT know what widgets the view consists of
    - All it knows is that the view has components that HasClickHandlers, HasValue<String>, etc.
  - Responsible for listening to events on those view components
  - Responsible for navigation to other view via history management
    - History.newItem("viewId");

Example: Presenter.java

```
package com.google.gwt.sample.contacts.client.presenter;

import com.google.gwt.user.client.ui.HasWidgets;

public interface Presenter {
  public void go(final HasWidgets container);
}
```



Container that you attach this Presenter to. If you have a single Presenter for entire app, then this might be the result of RootPanel.get(). But, if AppController sets up a split screen with a Presenter for each part, this might be one of the parts (e.g., a FlowPanel).

Gives the presenter a chance to take the view it's responsible for and attach it to the container with Display.asWidget() call. See later slides.

Example: (Contacts List) ContactsPresenter.java

```
package com.google.gwt.sample.contacts.client.presenter;
...
public class ContactsPresenter implements Presenter {
  private List<ContactDetails> contactDetails;

  public interface Display {
    HasClickHandlers getAddButton();
    HasClickHandlers getDeleteButton();
    HasClickHandlers getList();
    void setData(List<String> data);
    int getClickedRow(ClickEvent event);
    List<Integer> getSelectedRows();
    Widget asWidget();
  }
```

Presenter's cached data.

The view that is driven by this presenter must implement this interface.

Key idea: the Display (view) lists the functionality that it will have, but not the concrete widgets that will be used to attain this functionality.

To actually attach the view to some container, we need its representation as a Widget, not just Display.

Example: (Contacts List) ContactsPresenter.java (cont.)

```
  private final ContactsServiceAsync rpcService;
  private final HandlerManager eventBus;
  private final Display display;

  public ContactsPresenter(ContactsServiceAsync rpcService,
                           HandlerManager eventBus,
                           Display view) {
    this.rpcService = rpcService;
    this.eventBus = eventBus;
    this.display = view;
  }
```

"Central nervous system" of the application. See later slides.

Example: (Contacts List) ContactsPresenter.java (cont.)

```
...
public void go(final HasWidgets container) {
  bind();
  container.clear();
  container.add(display.asWidget());
  fetchContactDetails();
}
...
```

Attach whatever handlers are needed for this view via bind.
Clear the container of whatever it is was there before.
Attach this presenter's display (as a Widget) to the container.

Kick off some action to take care of business logic of this presenter. In this case, fetch contacts (from server) and populate the display's list.

Example: (Contacts List) ContactsPresenter.java (cont.)

```
public void bind() {
  display.getAddButton().addClickHandler(
  new ClickHandler() {
    public void onClick(ClickEvent event) {
      eventBus.fireEvent(new AddContactEvent());
    }
  });

  display.getDeleteButton().addClickHandler(
  new ClickHandler() {
    public void onClick(ClickEvent event) {
      deleteSelectedContacts();
    }
  });
```

Example: (Contacts List) ContactsPresenter.java (cont.)

```
  display.getList().addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
      int selectedRow = display.getClickedRow(event);
      if (selectedRow >= 0) {
        String id = contactDetails
                          .get(selectedRow).getId();
        eventBus.fireEvent(new EditContactEvent(id));
      }
    }
  });
}
```

Presenter-cached list of displayed contacts in the *same order as they were added to the display.*

Example: (Contacts List) ContactsPresenter.java (cont.)

```
private void fetchContactDetails() {
  rpcService.getContactDetails(
  new AsyncCallback<ArrayList<ContactDetails>>() {
   public void onSuccess(ArrayList<ContactDetails> result) {
        contactDetails = result;
        sortContactDetails();
        List<String> data = new ArrayList<String>();
        for (int i = 0; i < result.size(); ++i) {
          data.add(contactDetails.get(i).getDisplayName());
        }
        display.setData(data);
    }

    public void onFailure(Throwable caught) {
      Window.alert("Error fetching contact details");
    }
  });
}
```

Note that the display doesn't even get the ContainDetails object. It just gets Strings! This is on purpose. The dumber the view the better. More reuse of it later, more lightweight processing for the widget!

Example: (Contacts List) ContactsPresenter.java (cont.)

```
private void deleteSelectedContacts() {
  List<Integer> selectedRows = display.getSelectedRows();
  ArrayList<String> ids = new ArrayList<String>();
  for (int i = 0; i < selectedRows.size(); ++i) {
    ids.add(contactDetails.get(selectedRows.get(i)).getId());
  }

  rpcService.deleteContacts(ids,
   new AsyncCallback<ArrayList<ContactDetails>>() {
     public void onSuccess(ArrayList<ContactDetails> result) {
       contactDetails = result;
       List<String> data = new ArrayList<String>();
       for (int i = 0; i < result.size(); ++i) {
         data.add(contactDetails.get(i).getDisplayName());
       }
       display.setData(data);
     }
     ...
```

Presenter-cached data structure, representing the model data for the display.

Example: View

Example: ContactsView.java

```
public class ContactsView extends Composite implements
                                                ContactsPresenter.Display {
  private final Button addButton;
  ...
  public ContactsView() {
   ...
  }
  ...
  public HasClickHandlers getAddButton() {
    return addButton;
  }
  ...
  public void setData(List<String> data) {
    contactsTable.removeAllRows();
    for (int i = 0; i < data.size(); ++i) {
      contactsTable.setWidget(i, 0, new CheckBox());
      contactsTable.setText(i, 1, data.get(i));
    }
  }

  public Widget asWidget() {
    return this;
  }
}
```


Example: AppController

AppController

- **Uber presenter**
- **Handles logic not specific to any particular presenter, i.e., application-wide logic**
  - Application-wide events like history management, transitions between views, etc.
- **Most often attaches itself directly to the RootLayoutPanel**
  - The only presenter EntryPoint class ever sees

Example: (EntryPoint Class) Contacts.java

```
package com.google.gwt.sample.contacts.client;
...
public class Contacts implements EntryPoint {
  public void onModuleLoad() {
    ContactsServiceAsync rpcService =
                    GWT.create(ContactsService.class);
    HandlerManager eventBus = new HandlerManager(null);
    AppController appViewer =
                    new AppController(rpcService, eventBus);
    appViewer.go(RootPanel.get());
  }
}
```

Example: AppController.java

```
package com.google.gwt.sample.contacts.client;
...
public class AppController implements Presenter,
                               ValueChangeHandler<String> {
  private final HandlerManager eventBus;
  private final ContactsServiceAsync rpcService;
  private HasWidgets container;

  public AppController(ContactsServiceAsync rpcService,
                       HandlerManager eventBus) {
    this.eventBus = eventBus;
    this.rpcService = rpcService;
    bind();
  }
  ...
```


Example: AppController.java (Continued)

```
  ...
  public void go(final HasWidgets container) {
    this.container = container;
    if ("".equals(History.getToken())) {
      History.newItem("list");
    }
    else {
      History.fireCurrentHistoryState();
    }
  }
```

Example: AppController.java (Continued)

```
private void bind() {
 History.addValueChangeHandler(this);
 eventBus.addHandler(AddContactEvent.TYPE,
                     new AddContactEventHandler() {
    public void onAddContact(AddContactEvent event) {
      doAddNewContact();
    }
 });

 eventBus.addHandler(EditContactEvent.TYPE,
                     new EditContactEventHandler() {
    public void onEditContact(EditContactEvent event) {
       doEditContact(event.getId());
    }
 });

 eventBus.addHandler(ContactUpdatedEvent.TYPE,
                     new ContactUpdatedEventHandler() {...
```

Registration of app-wide history management handler and other app-wide handlers.

Example: AppController.java (cont.)

```
public void onValueChange(ValueChangeEvent<String> event) {
  String token = event.getValue();

  if (token != null) {
    Presenter presenter = null;

    if (token.equals("list")) {
      presenter = new ContactsPresenter(rpcService, eventBus,
                                        new ContactsView());
    } else if (token.equals("add")) {
      presenter = new EditContactPresenter(rpcService, eventBus,
                                           new EditContactView());
    } else if (token.equals("edit")) {
      presenter = new EditContactPresenter(rpcService, eventBus,
                                           new EditContactView());
    }

    if (presenter != null) {
      presenter.go(container);
    } ...
```

Example: EditContactEventHandler.java

```
package com.google.gwt.sample.contacts.client.event;

import com.google.gwt.event.shared.EventHandler;

public interface EditContactEventHandler extends EventHandler {
  void onEditContact(EditContactEvent event);
}
```


Example: EditContactEvent.java

```
public class EditContactEvent extends
                              GwtEvent<EditContactEventHandler> {
  public static Type<EditContactEventHandler> TYPE =
                              new Type<EditContactEventHandler>();
  private final String id;
  public String getId() { return id; }
  public EditContactEvent(String id) {
    this.id = id;
  }

  @Override
  public Type<EditContactEventHandler> getAssociatedType() {
    return TYPE;
  }

  @Override
  protected void dispatch(EditContactEventHandler handler) {
    handler.onEditContact(this);
  }
}
```

Example: ContactsView.java

```
public class ContactsView extends Composite implements
                                                ContactsPresenter.Display {
  private final Button addButton;
  ...
  public ContactsView() {
   ...
  }
  ...
  public HasClickHandlers getAddButton() {
    return addButton;
  }
  ...
  public void setData(List<String> data) {
    contactsTable.removeAllRows();
    for (int i = 0; i < data.size(); ++i) {
      contactsTable.setWidget(i, 0, new CheckBox());
      contactsTable.setText(i, 1, data.get(i));
    }
  }

  public Widget asWidget() {
    return this;
  }
}
```


Example: ContactsService.java

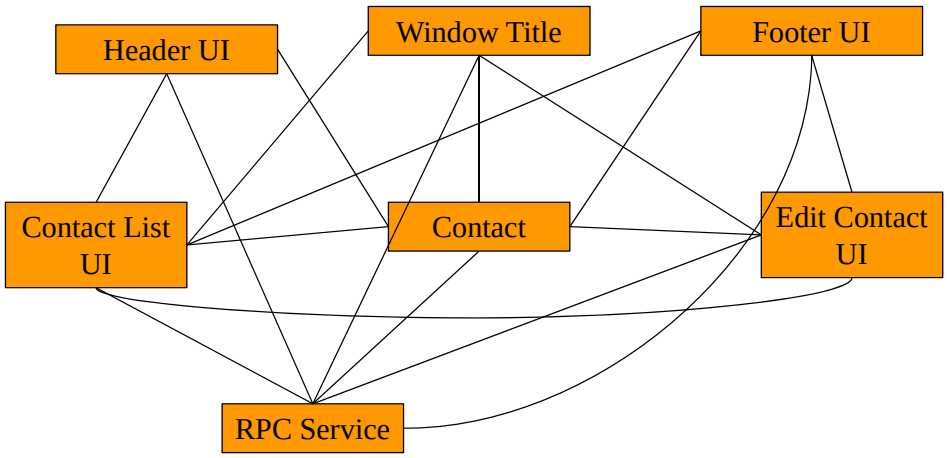
```
package com.google.gwt.sample.contacts.client;
...

@RemoteServiceRelativePath("contactsService")
public interface ContactsService extends RemoteService {

  Contact addContact(Contact contact);
  Boolean deleteContact(String id);
  ArrayList<ContactDetails> deleteContacts(
                                          ArrayList<String> ids);
  ArrayList<ContactDetails> getContactDetails();
  Contact getContact(String id);
  Contact updateContact(Contact contact);
}
```

EventBus (HandlerManager)

Need for EventBus

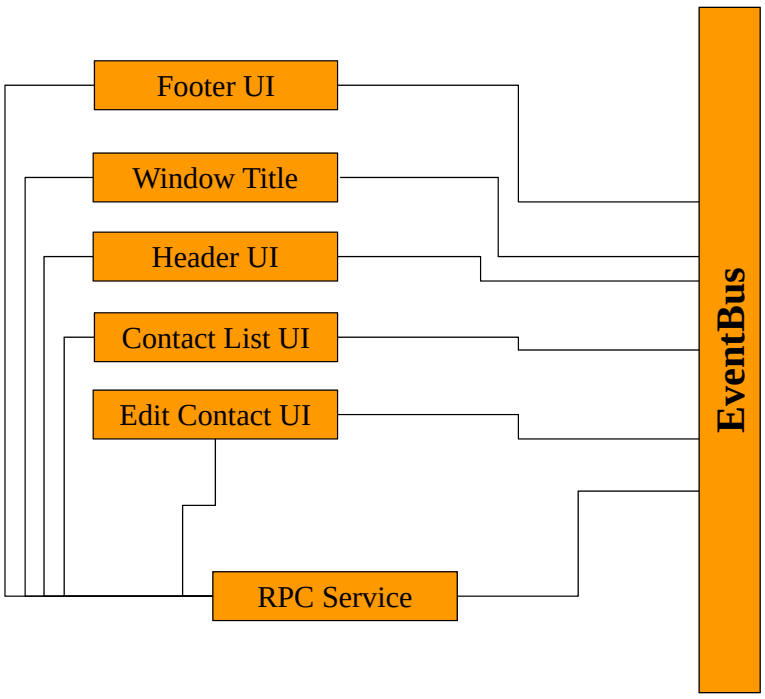
- **Many events occur in the system**
- **Application wide events**
  - History management
- **Component to component events**
  - One component needs to communicate some information to another component
- **Example 1**
  - Someone clicks a link with target “edit”
  - Presenter fires event “Someone wants Edit screen”
- **Example 2**
  - Presenter completes server-side call to delete item
  - Presenter fires event “Contact was deleted”
    - Components that care about that should do something about it

Without EventBus: Coupled Spaghetti



With EventBus: Loose Coupling

This is NOT to imply that it is never appropriate for one component to call another one directly.



Testing MVP Apps

Testing

- **Presenter is where main functionality lives**
  - So, we (mostly) test the presenter using regular JUnit tests, mocking out the rest of the components
  - For tests that absolutely need browser environment to be tested properly, we use GWTTestCase class
  - The rest is tested using some framework like Selenium
- **There is a lot to be said for factoring out most of the behavior even out of the presenter**
  - Not even mocking would be required then
    - I.e., use a lot of utilities, etc.
- **Tests go into the test source tree in Eclipse**

Testing Contacts Sorting From ContactsPresenter

```
public void sortContactDetails() {
  for (int i = 0; i < contactDetails.size(); ++i) {
    for (int j = 0; j < contactDetails.size() - 1; ++j) {
      if (contactDetails.get(j).getDisplayName()
              .compareToIgnoreCase(contactDetails.get(j + 1)
                                   .getDisplayName()) >= 0) {
        ContactDetails tmp = contactDetails.get(j);
        contactDetails.set(j, contactDetails.get(j + 1));
        contactDetails.set(j + 1, tmp);
      }
    }
  }
}
```


Example: JRE Test ExampleJRETest.java

```
public class ExampleJRETest extends TestCase {
  private ContactsPresenter contactsPresenter;
  private ContactsServiceAsync mockRpcService;
  private HandlerManager eventBus;
  private ContactsPresenter.Display mockDisplay;

  protected void setUp() {
    mockRpcService =
                  createStrictMock(ContactsServiceAsync.class);
    eventBus = new HandlerManager(null);
    mockDisplay =
              createStrictMock(ContactsPresenter.Display.class);
    contactsPresenter = new ContactsPresenter(mockRpcService,
                                              eventBus,
                                              mockDisplay);
  }
```

Example: JRE Test ExampleJRETest.java

```
public void testContactSort() {
    ArrayList<ContactDetails> contactDetails =
                                    new ArrayList<ContactDetails>();
    contactDetails.add(new ContactDetails("0", "c_contact"));
    contactDetails.add(new ContactDetails("1", "b_contact"));
    contactDetails.add(new ContactDetails("2", "a_contact"));
    contactsPresenter.setContactDetails(contactDetails);
    contactsPresenter.sortContactDetails();

    assertTrue(contactsPresenter.getContactDetail(0)
                        .getDisplayName().equals("a_contact"));
    assertTrue(contactsPresenter.getContactDetail(1)
                        .getDisplayName().equals("b_contact"));
    assertTrue(contactsPresenter.getContactDetail(2)
                        .getDisplayName().equals("c_contact"));
  }
}
```


Example: GWT Test ExampleGWTTest.java

```
public class ExampleGWTTest extends GWTTestCase {
  private ContactsPresenter contactsPresenter;
  private ContactsServiceAsync rpcService;
  private HandlerManager eventBus;
  private ContactsPresenter.Display display;

  public String getModuleName() {
    return "com.google.gwt.sample.contacts.Contacts";
  }

  public void gwtSetUp() {
    rpcService = GWT.create(ContactsService.class);
    eventBus = new HandlerManager(null);
    display = new ContactsView();
    contactsPresenter = new ContactsPresenter(rpcService,
                                              eventBus, display);
  }
```

Example: GWT Test ExampleGWTTest.java

```
  public void testContactSort(){
    ArrayList<ContactDetails> contactDetails =
                                  new ArrayList<ContactDetails>();
    contactDetails.add(new ContactDetails("0", "c_contact"));
    contactDetails.add(new ContactDetails("1", "b_contact"));
    contactDetails.add(new ContactDetails("2", "a_contact"));
    contactsPresenter.setContactDetails(contactDetails);
    contactsPresenter.sortContactDetails();

    assertTrue(contactsPresenter.getContactDetail(0)
                        .getDisplayName().equals("a_contact"));
    assertTrue(contactsPresenter.getContactDetail(1)
                        .getDisplayName().equals("b_contact"));
    assertTrue(contactsPresenter.getContactDetail(2)
                        .getDisplayName().equals("c_contact"));
  }
}
```


Wrap-Up

Summary

- **Big learning curve, but worth it for large/complex apps**
- **Model**
  - Responsible for holding raw data
  - No business logic (just like any model)
- **View**
  - Responsible for displaying data
  - No business logic allowed
  - Implements a Display interface
    - Usually defined by the presenter, but not always
- **Presenter**
  - Responsible for getting the data, driving the view, listening for GUI events, implements business logic
- **AppController**
  - Uber presenter
  - Responsible for registering app-wide event handlers, e.g., history events
  - Wires all components together

Questions?