Google
Developers

# Demystifying MVP and EventBus in GWT

Erik Kuefler
Software Engineer, Google Wallet

## Agenda

**Background:** GWT, UiBinder, Gin

**Questions to answer:**

- What is MVP?
- What are some strategies for implementing it?
- When should I consider alternatives?
- How should I test my application?
- How can I combine multiple pieces of my application?
- How should I get those pieces talking to each other?

## Before we begin...

First: **what are our goals?**

Testability

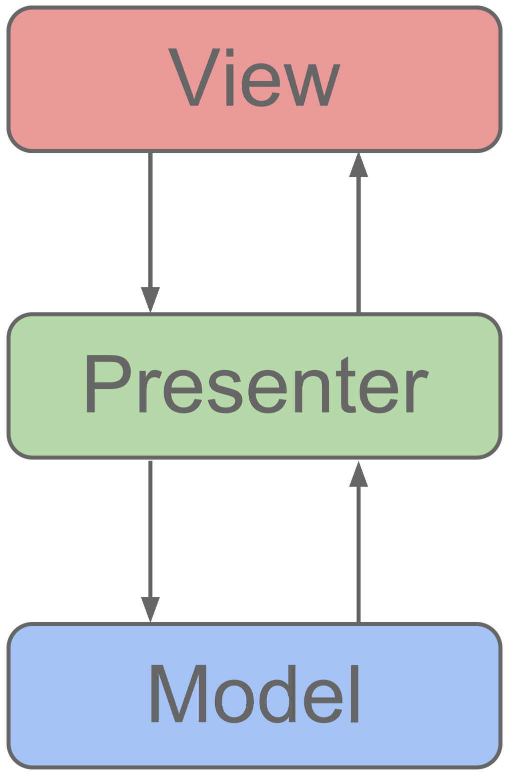
*~100% of application logic can be tested using plain JVM TestCases*

Maintainability

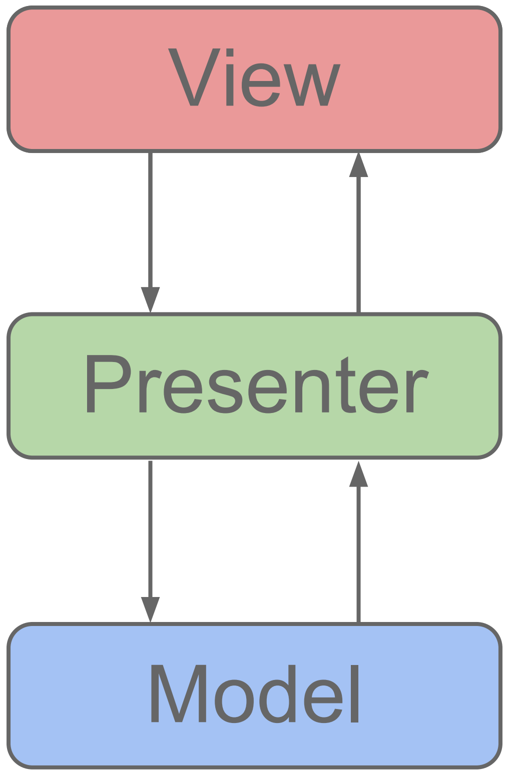
*Simple changes are easy, complex changes are possible*

## MVP and its alternatives

## What is MVP?



## What is MVP?



Key question: **how much code goes in the view?**

## Rich Views

Option 1: **Lots of code**

*Views should contain all of the application's display logic*

**Pros**

- Ensures no DOM code in the presenter

**Cons**

- Pretty vague - what is "display logic"?
- Code in the view is hard to test
- We should be using UiBinder to define static layouts anyways

Java

```java
void setContact(Contact c) {
  // Build children
  Label name = new Label(
      c.getFirstName() + " " + c.getLastName());
  Label phone = new Label();
  for (PhoneNumber phone : c.getPhoneNumbers()) {
    if (phone.isDefault()) {
      phone.setText(formatter.format(phone));
    }
  }
  // Style root widget
  if (c.isFavorite()) {
   addStyle(style.favorite());
  }
  // Add children to root
  add(name);
  add(phone);
  // Attach handlers
  phone.addClickHandler(new ClickHandler() {
    void onClick(ClickEvent e) {
      presenter.onPhoneNumberClicked();
    }
  }
}
```

## Simple Views

Option 2: **Little code**

*Views should be a thin wrapper around the widgets in a ui.xml file*

**Pros**

- Still keeps DOM code out of the presenter
- Keeps more code in the presenter, making it easier to test

**Cons**

- Complexity tends to slowly creep up over time
- Tedious to maintain

ui.xml

```
<g:HTMLPanel ui:field="root">
  <g:Label ui:field="name"/>:  <g:Label ui:field="phone"/>
</g:HTMLPanel>
```

Java

```
@UiField Widget root;
@UiField HasText name;
@UiField HasText phone;

void setName(String name) {
  this.name.setText(name);
}

void setPhoneNumber(String phone) {
  this.phoneNumber.setText(phone);
}

void setIsFavorite(boolean isFavorite) {
  this.root.setStyle(style.favorite(), isFavorite);
}

@UiHandler("phoneNumber")
void onPhoneNumberClicked(ClickEvent e) {
  presenter.onPhoneNumberClicked();
}
```

# Eliminating the View

ui.xml

```
<g:HTMLPanel ui:field="root">
  <g:Label ui:field="name"/>:  <g:Label ui:field="phone"/>
</g:HTMLPanel>
```

Option 3: **Zero code**

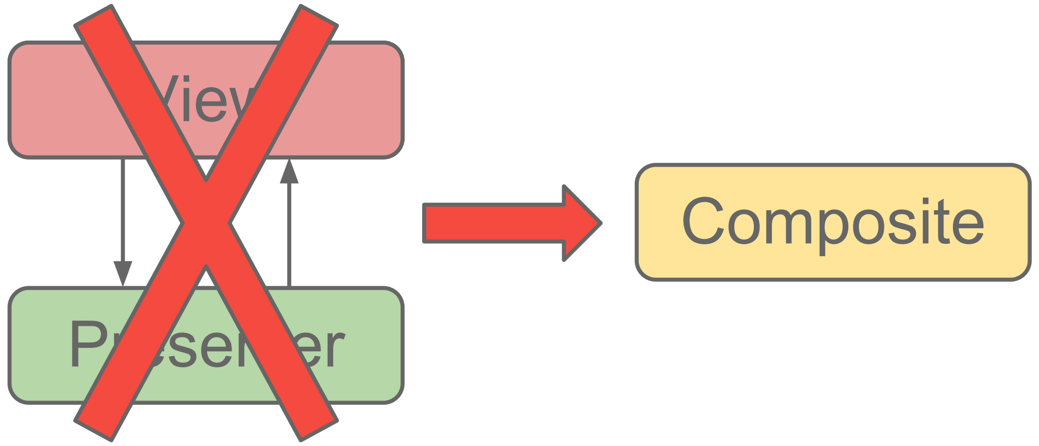
*For fairly static UIs, there's no reason to have a wrapper class at all. Combine the view and presenter into a single class.*

**Pros**

- No ambiguity over what goes where
- No place for code to hide from unit tests
- No boilerplate classes

**Cons**

- Less opportunity to encapsulate DOM manipulation



## Full Example: Rich View

ContactPresenter.java

```java
class ContactPresenter {
  interface ContactDisplay {
    void setContact(Contact contact);
  }

  @Inject ContactsDisplay view;
  private List<Contact> contacts;

 void loadData(ServerResponse response) {
    contacts = extractContacts(response);
  }
 void selectContact(String id) {
    for (Contact contact : contacts) {
      if (contact.getId().equals(id)) {
        view.setContact(contact);
        return;
      }
    }
  }

  void onPictureClicked() {
    showPictureEditor();
  }
}
```

Java

ContactView.java

```java
class ContactView extends Composite
    implements ContactDisplay {
  @UiField Widget root;
  @UiField Style style;
  @UiField HasText name;
  private final ContactPresenter presenter;

  @Inject void ContactView(ContactPresenter p) {
    this.presenter = p;
    uiBinder.createAndBindUi(this);
  }

  void setContact(Contact contact) {
    name.setText(contact.getFirstName() + " " +
                        contact.getLastName());
    if (contact.isFavorite()) {
      root.addStyleName(style.favorite());
    }
  }

  @UiHandler("picture")
  void onPictureClicked(ClickEvent e) {
    presenter.onPictureClicked();
  }
}
```

Java

ContactView.ui.xml

```xml
<g:HTMLPanel ui:field="root">
  <g:Image ui:field="picture">
  <g:Label ui:field="name"/>
</g:HTMLPanel>
```

ui.xml

Fairly maintainable - not much repetition ✓

Testability suffers since the view has logic ✗

# Full Example: Rich View

ContactPresenter.java

```java
class ContactPresenter {
  interface ContactDisplay {
    void setContact(Contact contact);
  }

  @Inject ContactsDisplay view;
  private List<Contact> contacts;

 void loadData(ServerResponse response) {
   contacts = extractContacts(response);
 }
 void selectContact(String id) {
   for (Contact contact : contacts) {
     if (contact.getId().equals(id)) {
       view.setContact(contact);
       return;
     }
   }
 }

  void onPictureClicked() {
    showPictureEditor();
  }
}
```

Java

ContactView.java

```java
class ContactView extends Composite
    implements ContactDisplay {
  @UiField Widget root;
  @UiField Style style;
  @UiField HasText name;
  private final ContactPresenter presenter;

  @Inject void ContactView(ContactPresenter p) {
    this.presenter = p;
    uiBinder.createAndBindUi(this);
  }

  void setContact(Contact contact) {
    name.setText(contact.getFirstName() + " " +
                            contact.getLastName());
    if (contact.isFavorite()) {
     root.addStyleName(style.favorite());
    }
  }

  @UiHandler("picture")
  void onPictureClicked(ClickEvent e) {
    presenter.onPictureClicked();
  }
}
```

Java

ContactView.ui.xml

```xml
<g:HTMLPanel ui:field="root">
 <g:Image ui:field="picture">
 <g:Label ui:field="name"/>
</g:HTMLPanel>
```

ui.xml

Fairly maintainable - not much repetition

✓

Testability suffers since the view has logic

✗

# Full Example: Simple View

ContactPresenter.java

Java

```
class ContactPresenter {
     rface ContactDisplay {
    void       me(Stri          ne);
    void add              e();
  }

  @Inject ContactsDisplay view;
  private List<Contact> contacts;

 void loadData(ServerResponse response) {
   contacts = extractContacts(response);
 }
 void selectContact(String id) {
   for (Contact contact : contacts) {
     if (contact.getId().equals(id)) {
       view.setName(contact.getFirstName() +
           " " + contact.getLastName());
       if (contact.isFavorite()) {
         root.addFavoriteStyleStyleName();
       }
       return;
     }
   }
 } ...
```

ContactView.java

Java

```
class ContactView extends Composite
    implements ContactDisplay {
  @UiField Widget root;
  @UiField Style style;
  @UiField HasText name;
  private final ContactPresenter presenter;

  @Inject    id ContactView(ContactPre    nter p) {
    this.pr      ter = p;
    uiBinder       teAndBindUi(this);
  }

  void setName(S       name)
    this.name.setTe      me
  }

  void addFavoriteSt
    root.addStyleNa     style      rite());
  }

  @UiHandl       cture")
  void onP       eClicked(ClickEvent e
    prese       onPictureClicked();
  }
}
```

ContactView.ui.xml

```
<g:HTMLPanel ui:field="root">
  <g:Image ui:field="picture">
  <g:Label ui:field="name"/>
</g:HTMLPanel>
```

ui.xml

Tedious to maintain due to lots of boilerplate

✗

Fairly testable since view has little logic

✓

# Full Example: Static View

ContactsComposite.java

Java

```
class ContactComposite extends Composite {
 @UiField Widget root;
 @UiField Style style;
 @UiField HasText name;
 private List<Contact> contacts;

 void loadData(ServerResponse response) {
   contacts = extractContacts(response);
 }
 void selectContact(String id) {
   for (Contact contact : contacts) {
     if (contact.getId().equals(id)) {
       name.setText(contact.getFirstName() +
          " " + contact.getLastName());
       if (contact.isFavorite()) {
         root.addStyleName(style.favorite());
       }
       return;
     }
   }
 }

 @UiHandler("picture") void onPictureClicked(ClickEvent e) {
   showPictureEditor();
 }
```

ContactView.ui.xml

ui.xml

```
<g:HTMLPanel ui:field="root">
  <g:Image ui:field="picture">
  <g:Label ui:field="name"/>
</g:HTMLPanel>
```

Easy to maintain since boilerplate is minimized ✓

Fully testable since the view can't have any logic ✓

## Tests Without GWTTestCase

If we can mock views, why not just mock widgets directly?

You can do this manually...

Java
```
@RunWith(MockitoTestRunner.class)
public class ContactCompositeTest {
  private ContactComposite contact;

  @Before public void setUp() {
    GWTMockUtilities.disarm();
    contact = new ContactsComposite() {
      protected void initWidget() { /* disarm for test */ }
    };
    contact.root = mock(Widget.class);
    contact.name = mock(Label.class);
  }

  @Test public void shouldSetName() {
    contact.setContact(new Contact("Fred", "Smith"));
    verify(contact.name).setText("Fred Smith");
  }
}
```

... or automate it with a library like GwtMockito

Java
```
@RunWith(GwtMockitoTestRunner.class)
public class ContactCompositeTest {
  private ContactComposite contact;

  @Before public void setUp() {
    contact = new ContactComposite();
  }

  @Test public void shouldSetName() {
    contact.setContact(new Contact("Fred", "Smith"));
    verify(contact.name).setText("Fred Smith");
  }
}
```

https://github.com/google/gwtmockito

## What's the Downside?

We've significantly reduced boilerplate, but it comes with some cost

- Sometimes display logic really should be separate
  - True of complex applications like games, rarely of more form-based applications
  - Widgets can be a good way to separate out graphical subcomponents
  - Often a good idea to start with a single class and factor out a view as needed

- Less flexibility in swapping view implementations
  - But this is usually more trouble than it is worth
  - Replacing ui.xml files can get you part of the way there
  - If you start with a single implementation, factoring out an interface is easy

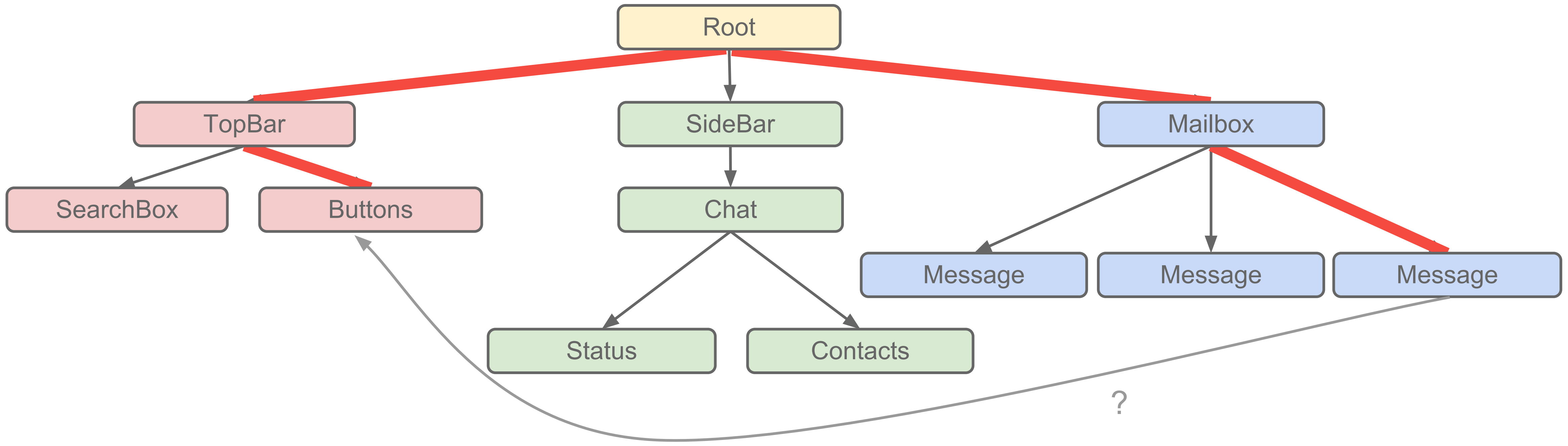
Rule of thumb: start with the simplest solution that can work, add complexity only when needed

## Composing components and communicating among them

# Composition and Communication

Visually, we can compose components in a tree mimicking the DOM

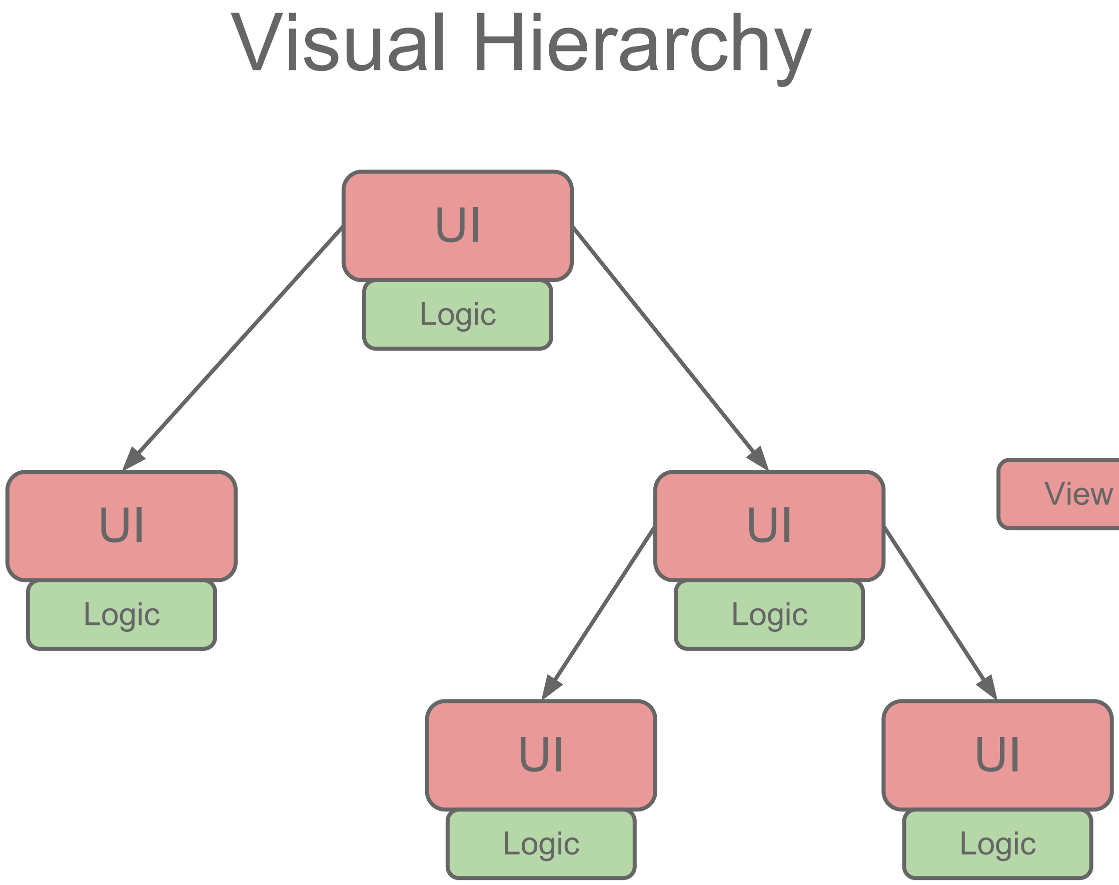
But we don't want to be tied to the DOM for communication

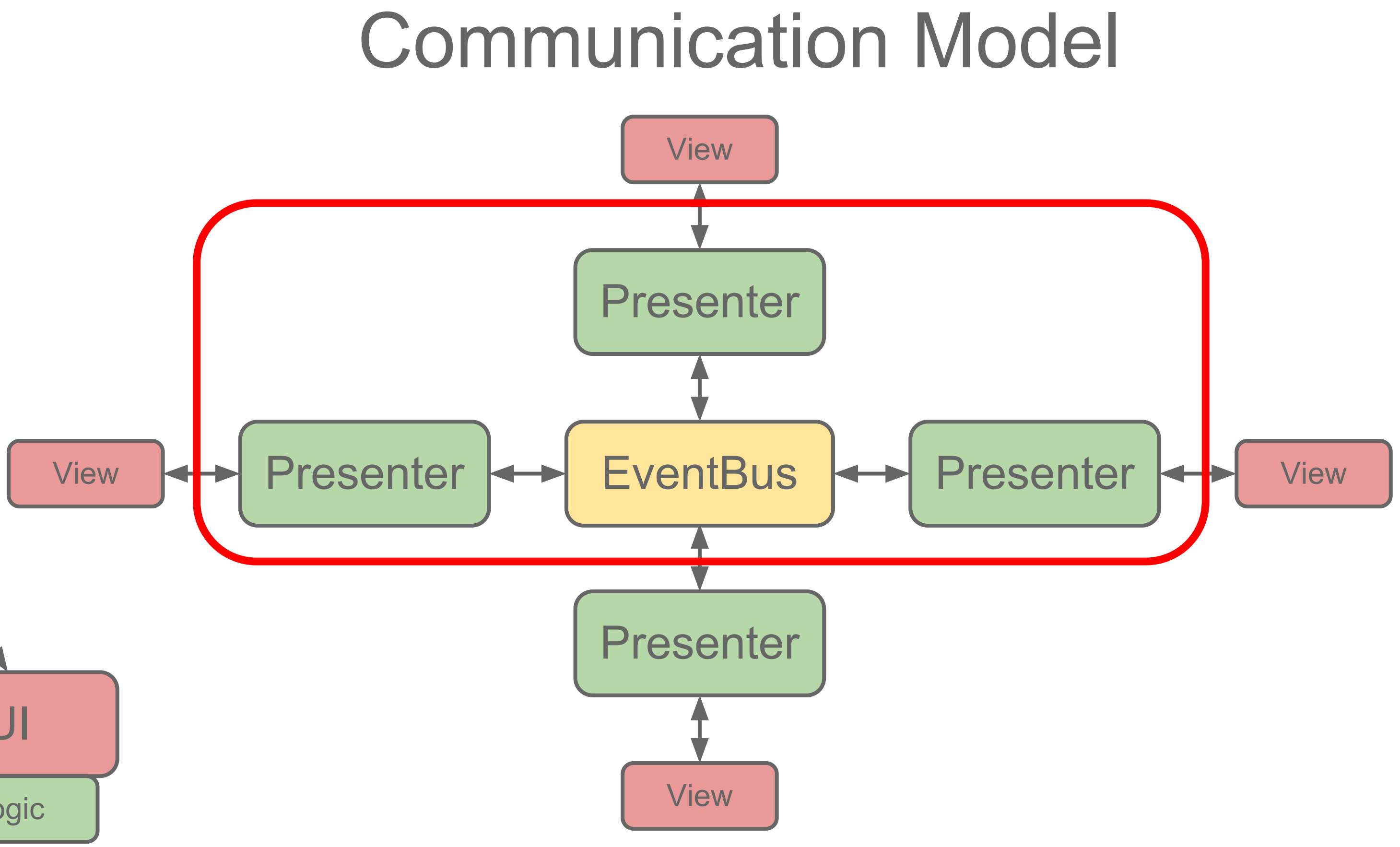


We need a way for distant components to talk to one another without knowing about each other

# Visual Hierarchy vs. Communication Model

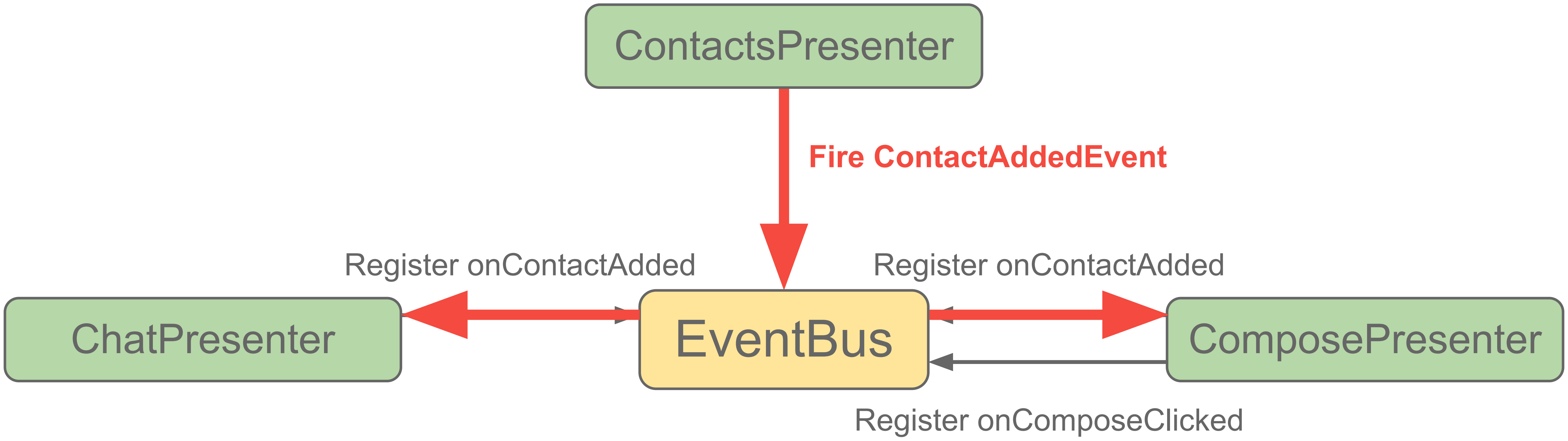
EventBus allows presenters to communicate without knowing about one another





## Decoupling Presenters with EventBus

The event bus is all about separating structure from communication



# Implementing EventBus

There's no magic involved. Here's a fully-functional implementation:

Java

```
public class EventBus {
  public interface EventHandler<T> {
    void handleEvent(T event);
  }

  private final Map<EventType<?>, List<EventHandler<?>>> handlerMap = Maps.newHashMap();

  public <T> void addHandler(EventType<T> type, EventHandler<T> handler) {
    if (!handlerMap.containsKey(type)) handlerMap.put(type, Lists.newLinkedList());
    handlerMap.get(type).add(handler);
  }

  public void fireEvent(Event event) {
    for (EventHandler<?> handler : handlerMap.get(event.getType())) {
      handler.handleEvent(event);
    }
  }
}
```

## Using EventBus

To use the event bus, first define your events...

You can do this manually...

Java

```java
public class ContactsLoadedEvent extends GwtEvent {
  public final List<Contact> contacts;
  public ContactsLoadedEvent(List<Contact> c) { contacts = c; }

  public static final Type<ContactsLoadedEvent> TYPE =
      new Type<ContactsLoadedEvent>();

  @Override
  protected void dispatch(ContactsLoadedHandler handler) {
    handler.onContactsLoaded(this);
  }
  @Override
  public GwtEvent.Type<ContactsLoadedEvent> getAssociatedType() {
    return TYPE;
  }
}
public interface ContactsLoadedEvent extends EventHandler {
  void onContactsLoaded(ContactsLoadedEvent event);
}
```

... or automate it with a library like EventBinder

Java

```java
public class ContactsLoadedEvent extends GenericEvent {
  public final List<Contact> contacts;
  public ContactsLoadedEvent(List<Contact> c) { contacts = c; }
}
```

https://github.com/google/gwteventbinder

## Using EventBus

... then register handlers for them ...

Manually...

```java
MyPresenter(EventBus eventBus) {
  eventBus.addHandler(new ContactsLoadedHandler() {
    void onContactsLoaded(ContactsLoadedEvent e) {
      onContactsLoaded(e);
    }
  }, ContactsLoadedEvent.TYPE);
  eventBus.addHandler(new ContactSavedHandler() {
    void onContactsLoaded(ContactSavedEvent e) {
      onContactSaved(e);
    }
  }, ContactSavedEvent.TYPE);
}

void onContactsLoaded(ContactsLoadedEvent event) {
  // Do stuff
}

void onContactSaved(ContactsSavedEvent event) {
  // Do stuff
}
```

... or via EventBinder

```java
interface MyEventBinder extends EventBinder<MyPresenter> {}

MyPresenter(EventBus eventBus, MyEventBinder binder) {
  binder.bindEventHandlers(this, eventBus);
}

@EventHandler
void onContactsLoaded(ContactsLoadedEvent event) {
  // Do stuff
}

@EventHandler
void onContactSaved(ContactSavedEvent event) {
  // Do stuff
}
```

https://github.com/google/gwteventbinder

## Using EventBus

... and fire them somewhere else

Java

```
eventBus.fireEvent(new ContactsLoadedEvent(
    Lists.newArrayList(
        new Contact("John",  "Doe"),
        new Contact("Jane",  "Doe")));
```

## Comparing Methods to Events

| Methods | vs | Events |
|---|---|---|
| ● Send messages between classes<br>● Can carry arbitrary arguments | | ● Send messages between classes<br>● Can carry arbitrary arguments |
| ● Have a single, known receiver<br>● Have a defined return value | | ● Have any number of unknown receivers<br>● Don't have return values |
| *Good for low-level commands between tightly-coupled components* | | *Good for high-level notifications between loosely-coupled components* |

## Best Practices for Using Events

**Events are *notifications*, not commands**

| LoadContactsFromServerEvent | ContactManagerOpenedEvent |
|---|---|
| ✗ | ✓ |

In practice, events are fired only for **user input** and **server responses**

## Putting it All Together

# Putting it All Together

Java

```java
public class ContactComposite extends Composite {
  @UiField(provided=true) Widget card;
  @UiField HasText name;
  private String contactId;

  @Inject ContactsComposite(
     EventBus eventBus, MyUiBinder uiBinder, InfoCardComposite card) {
    this.card = card;
    initWidget(uiBinder.createAndBindUi(this));
    eventBinder.bindEventHandlers(eventBus, this);
  }
  @UiHandler("picture") void onPictureClicked(ClickEvent event) {
    eventBus.fireEvent(new ContactPictureClickedEvent(contactId));
  }
  @EventHandler void onContactLoaded(ContactLoadedEvent event) {
    name.setText(event.getFirstName() + " " + event.getLastName());
    contactId = event.getId();
  }
  @EventHandler void onPictureLoaded(PictureLoadedEvent event) {
    picture.setUrl(event.getUrl());
  }
}
```

## Putting it All Together

Java

```
public class ContactComposite extends Composite {
  @UiField(provided=true) Widget card;
  @UiField HasText name;
  private String contactId;

  @Inject ContactsComposite(
     EventBus eventBus, MyUiBinder uiBinder, InfoCardComposite card) {
    this.card = card;
    initWidget(uiBinder.createAndBindUi(this));
    eventBinder.bindEventHandlers(eventBus, this);
  }
  @UiHandler("picture") void onPictureClicked(ClickEvent event) {
    eventBus.fireEvent(new ContactPictureClickedEvent(contactId));
  }
  @EventHandler void onContactLoaded(ContactLoadedEvent event) {
    name.setText(event.getFirstName() + " " + event.getLastName());
    contactId = event.getId();
  }
  @EventHandler void onPictureLoaded(PictureLoadedEvent event) {
    picture.setUrl(event.getUrl());
  }
}
```

Things to note:

- All Java code lives in one widget
- Layout is defined by the ui.xml file
- Children are injected, but never referenced after being placed in @UiFields

## Putting it All Together

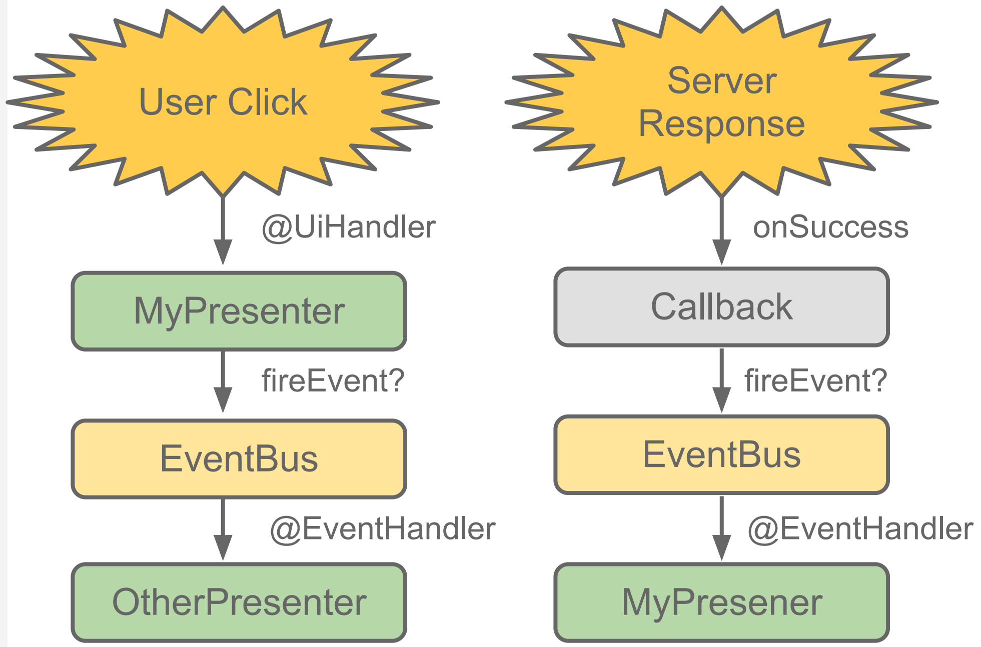
Java

```
public class ContactComposite extends Composite {
  @UiField(provided=true) Widget card;
  @UiField HasText name;
  private String contactId;

  @Inject ContactsComposite(
     EventBus eventBus, MyUiBinder uiBinder, InfoCardComposite card) {
    this.card = card;
    initWidget(uiBinder.createAndBindUi(this));
    eventBinder.bindEventHandlers(eventBus, this);
  }
  @UiHandler("picture") void onPictureClicked(ClickEvent event) {
    eventBus.fireEvent(new ContactPictureClickedEvent(contactId));
  }
  @EventHandler void onContactLoaded(ContactLoadedEvent event) {
    name.setText(event.getFirstName() + " " + event.getLastName());
    contactId = event.getId();
  }
  @EventHandler void onPictureLoaded(PictureLoadedEvent event) {
    picture.setUrl(event.getUrl());
  }
}
```

Things to note:

- No public methods
- All non-private methods are @UiHandlers or @EventHandlers
- @UiHandlers aren't required to fire events



## Summary

- For relatively static UIs, simple composites can be better than MVP
- Use an EventBus to decouple pieces of your application
- Define events that represent notifications, not commands
- Whatever you do, start simple and add complexity only when you need it

## Q & A

ekuefler@google.com
https://github.com/ekuefler
+Erik Kuefler

EventBinder: https://github.com/google/gwteventbinder
GwtMockito: https://github.com/google/gwtmockito

Google Developers