canoo

› your provider for business web solutions ›

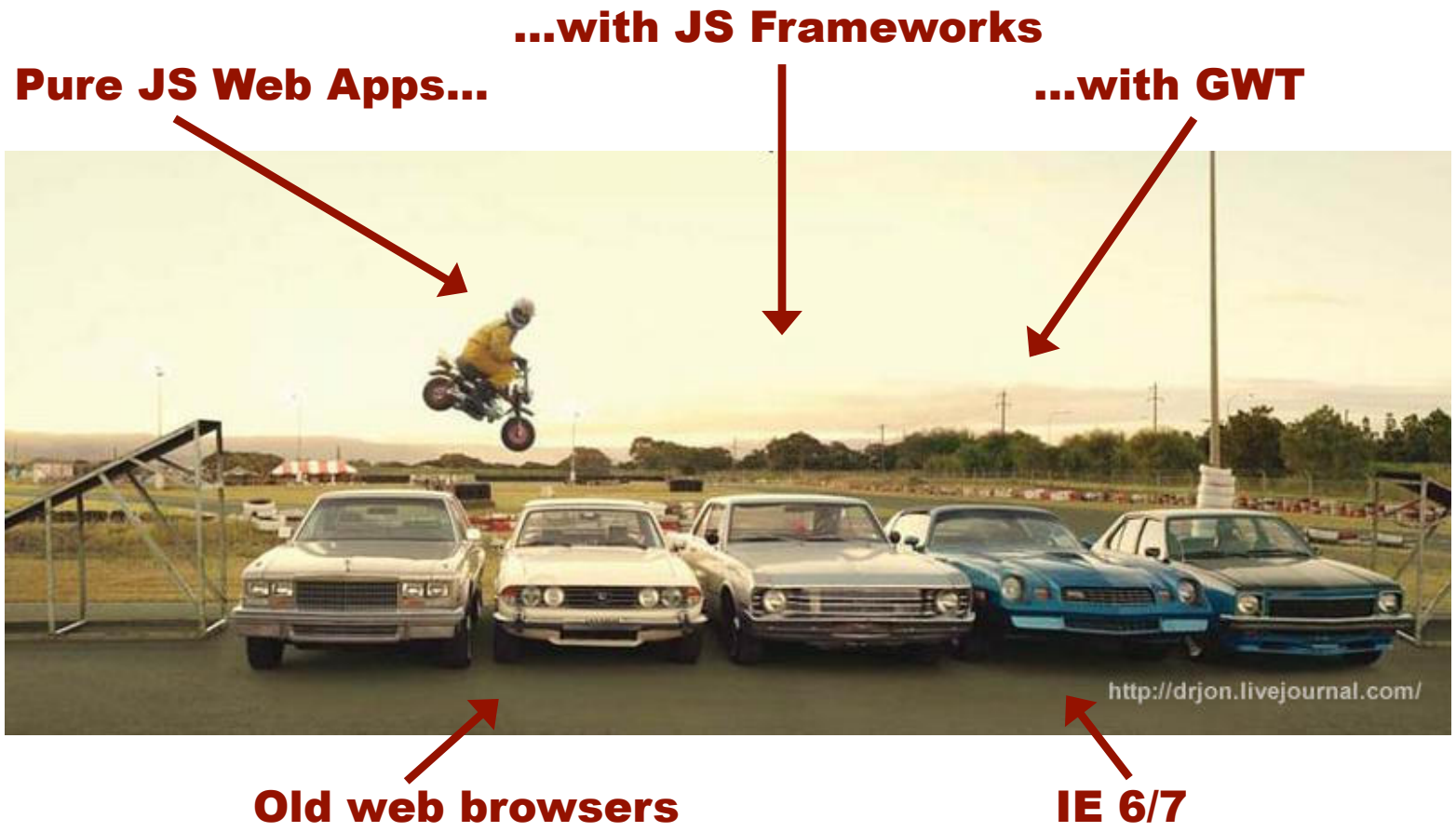
Cooking with GWT: Recipes for the perfect dinner

Alberto Mijares
alberto.mijares@canoo.com
Basel, Switzerland

Introduction

- Who am I?
- What do I do?
- What is GWT?
- What does GWT try to solve?
- What does GWT provide?
- What to expect from this presentation?
- What is in the slides?

Introduction (II)



Recipe 1: how to present your dishes.

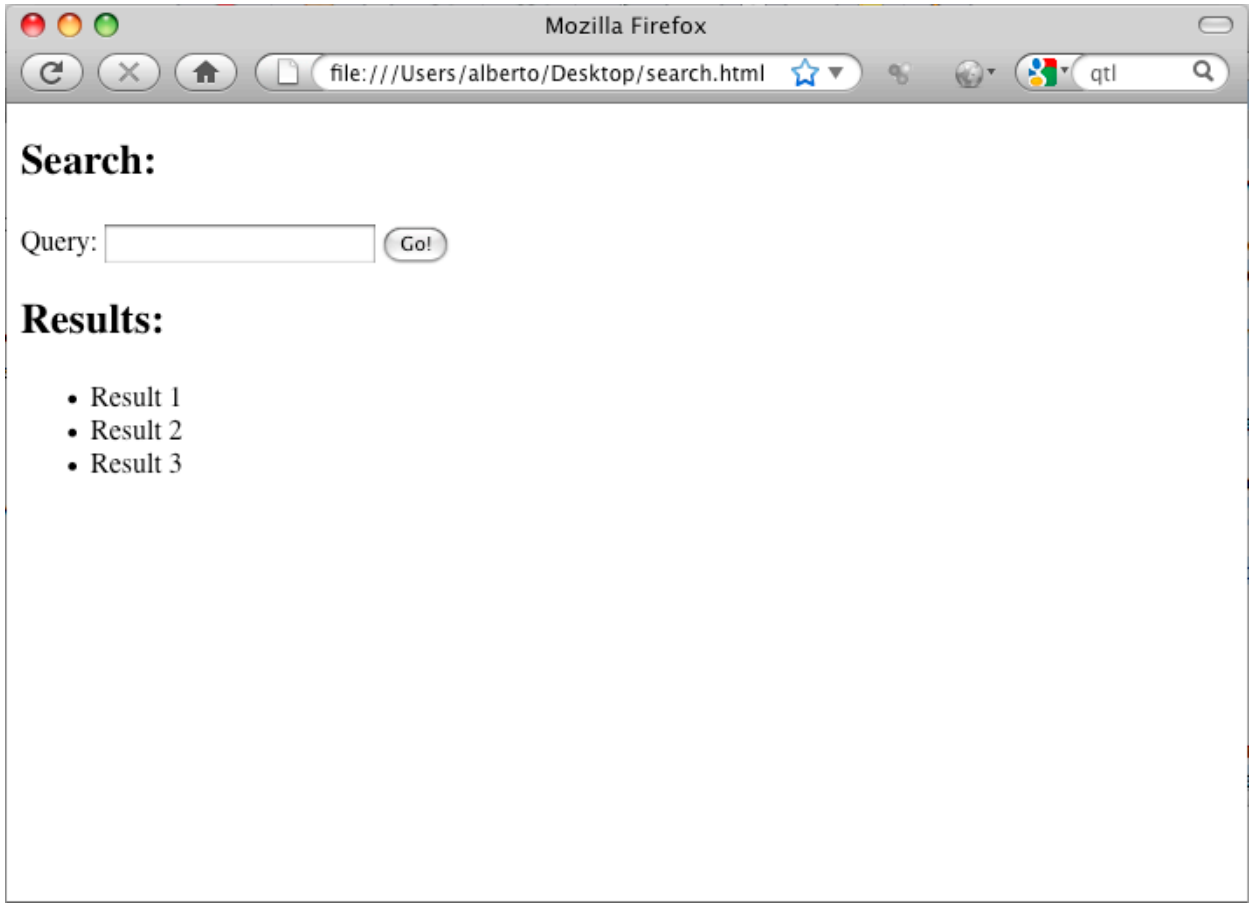
Photo: © 2007 Real Estate Chocolate

“Go declarative, less Java code in the UI”

Because many developers are familiar with Swing, it can happen that they build an UI in GWT using pure Java. It is perfectly possible, but because you are building an Ajax application (HTML behind!) it is not a good idea:

- The code for building widgets is difficult to refactor and makes the Java classes more complex and difficult to understand.
- It couples the different responsibilities and different levels of your UI (more on this later).
- Produces too complex DOM structures that arise all kind of issues in different browsers.
- Avoids leveraging knowledge and skills from UI specialists.
- Either delays feedback on the UI, or repeats work done with HTML prototypes.

A little search application



Target HTML interface

```
<html>
<body>
<div id="searchBox">
    <h2>Search:</h2>
    <form action="">
        <label for="queryInput">Query:</label>
        <input id="queryInput" type="text"/>
        <input type="submit" value="Go!"/>
    </form>
</div>
<div id="searchResults">
    <h2>Results:</h2>
    <ul id="results">
        <li>Result 1</li>
        <li>Result 2</li>
        <li>Result 3</li>
    </ul>
</div>
</body>
</html>
```

GWT “Swing-like” approach

```
public void onModuleLoad() {
    Widget searchBox = createSearchBox("Query:", "Go!");
    Panel searchResults = createSearchResults();
    VerticalPanel container = new VerticalPanel();
    container.add(new Label("Search:")); container.add(searchBox);
    container.add(new Label("Results:")); container.add(searchResults);
    RootPanel.get().add(container);
}
private Widget createSearchBox(String inputLabel, String buttonLabel) {
    HorizontalPanel searchBox = new HorizontalPanel();
    searchBox.add(new Label(inputLabel)); searchBox.add(new TextBox());
    searchBox.add(new Button(buttonLabel));
    return searchBox;
}
private Panel createSearchResults() {
    VerticalPanel searchResults = new VerticalPanel();
    for (int i = 1; i <= 3; i++) {
        searchResults.add(new Label("Result " + i));
    }
    return searchResults;
}
```

GWT “Swing-like” approach

```
public void onModuleLoad() {
    Widget searchBox = createSearchBox("Query:", "Go!");
    Panel searchResults = createSearchResults();
    VerticalPanel container = new VerticalPanel();
    container.add(new Label("Search:")); container.add(searchBox);
    container.add(new Label("Results:")); container.add(searchResults);
    RootPanel.get().add(container);
}
private Widget createSearchBox(String inputLabel, String buttonLabel) {
    HorizontalPanel searchBox = new HorizontalPanel();
    searchBox.add(new Label(inputLabel)); searchBox.add(new TextBox());
    searchBox.add(new Button(buttonLabel));
    return searchBox;
}
private Panel createSearchResults() {
    VerticalPanel searchResults = new VerticalPanel();
    for (int i = 1; i <= 3; i++) {
        searchResults.add(new Label("Result " + i));
    }
    return searchResults;
}
```

GWT “Swing-like” approach: resulting DOM

```
<div>
   <div class="gwt-Label">Search:</div>
   <table cellspacing="0" cellpadding="0"> <= We have a table here!!
      <tbody>
         <tr>
            <td align="left" style="vertical-align: top;">
               <div class="gwt-Label">Query:</div></td>
            <td align="left" style="vertical-align: top;">
               <input type="text" tabindex="0" class="gwt-TextBox"/></td>
            <td align="left" style="vertical-align: top;">
               <button type="button" tabindex="0" class="gwt-Button">Go!
               </button></td>
         </tr>
      </tbody>
   </table>
   <div class="gwt-Label">Results:</div>
   <table>                                    <= … and also here!!
   …
   </table>
</div>
```

Using GWT “UI Binder”: SearchComponent.java

```
public class SearchGwt implements EntryPoint {
    public void onModuleLoad() {
        SearchComponent searchComponent = new SearchComponent();
        RootPanel.get().add(searchComponent);
    }
}

…

public class SearchComponent extends Composite {
    interface Binder extends UiBinder<Widget, SearchComponent> {}

    private static Binder uiBinder = GWT.create(Binder.class);

    public SearchComponent() {
        initWidget(uiBinder.createAndBindUi(this));
    }
}
```

Using GWT “UI Binder”: SearchComponent.ui.xml

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
             xmlns:g='urn:import:com.google.gwt.user.client.ui'>
    <g:HTMLPanel>
        <div id="searchBox">
            <h2>Search:</h2>
            <form action="">
                <label for="queryInput">Query:</label>
                <input id="queryInput" type="text"/>
                <input type="submit" value="Go!"/>
            </form>
        </div>
        <div id="searchResults">
            <h2>Results:</h2>
            <ul id="results">
               <li>Result 1</li>
               <li>Result 2</li>
               <li>Result 3</li>
            </ul>
        </div>
    </g:HTMLPanel>
</ui:UiBinder>
```

Using GWT “UI Binder”: SearchComponent.ui.xml

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
             xmlns:g='urn:import:com.google.gwt.user.client.ui'>
    <g:HTMLPanel>
        <div id="searchBox">
            <h2>Search:</h2>
            <form action="">
                <label for="queryInput">Query:</label>
                <input id="queryInput" type="text"/>
                <input type="submit" value="Go!"/>
            </form>
        </div>
        <div id="searchResults">
            <h2>Results:</h2>
            <ul id="results">
               <li>Result 1</li>
               <li>Result 2</li>
               <li>Result 3</li>
            </ul>
        </div>
    </g:HTMLPanel>
</ui:UiBinder>
```

Using GWT “UI Binder”: SearchComponent.ui.xml (II)

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
             xmlns:g='urn:import:com.google.gwt.user.client.ui'>
    <g:HTMLPanel>
        <div id="searchBox">
            <h2>Search:</h2>
            <label for="queryInput">Query:</label>
            <g:TextBox ui:field="queryInput"/>
            <g:Button ui:field="searchButton" text="Go!"/>
        </div>
        <div id="searchResults">
            <h2>Results:</h2>
            <ul ui:field="results"/>
        </div>
    </g:HTMLPanel>
</ui:UiBinder>
```

Using GWT “UI Binder”: SearchComponent.java (II)

```
public class SearchComponent extends Composite {
    … (Binder interface definition)
    @UiField
    TextBox queryInput;
    @UiField
    Button searchButton;
    @UiField
    UListElement results;
    … (Constructor)

    @UiHandler("searchButton")
    void buttonClick(ClickEvent event) {
        for (int i = 1; i <= 3; i++) {
            results.appendChild(createResultsItem("Result " + i));
        }
    }
    private Element createResultsItem(String value) {
        LIElement result = Document.get().createLIElement();
        result.setInnerHTML(value);
        return result;
    }
}
```

Recipe 1:

“For the presentation, avoid ingredients with a strong presence”.

When building the UI, use as less Java code as possible.

Summary:

Things to Avoid in GWT when building a UI:

- Using panels that generate tables (VerticalPanel, HorizontalPanel, …), use “Layout” panels instead.
- Deep nesting panels: generates very complicated and deep DOM structures that produce layout issues.
- Building your UI with Java code: pay attention to methods like “Panel.add()”!
- Developing your own widgets extending widget classes: you better use composition or get ready for changes!

Things I do not like from UI Binder:

- It is XML based: XHTML would be better, cleaner and easier to work with (Wicket-like approach).
- It mixes two things: UI components composition and HTML binding.
- There are some unresolved inheritance issues when you try to extend GWT widgets and use them with “UIBinder”.

Recipe 2: how to balance your menus.

Photo: © Deni

“Separate components in levels”

High Level UI Components:

- They have low granularity (5 +/- 2) and are normally identifiable with top level UI containers: one for each "big" part of your UI or one for each service or one for each “main” domain entity. Normally depends on your approach: “Feature driven”, “Service driven”, “Domain driven”...
- They are normally responsible for invoking backend services informing the other “High Level Components” of the request state. When the response has been delivered, they set the response payload in form of “model” in internal UI Low Level Components (widgets) to present the data.
- They know nothing about presentation and how to handle data. They mediate between general layout, other UI High Level Components and the application services.

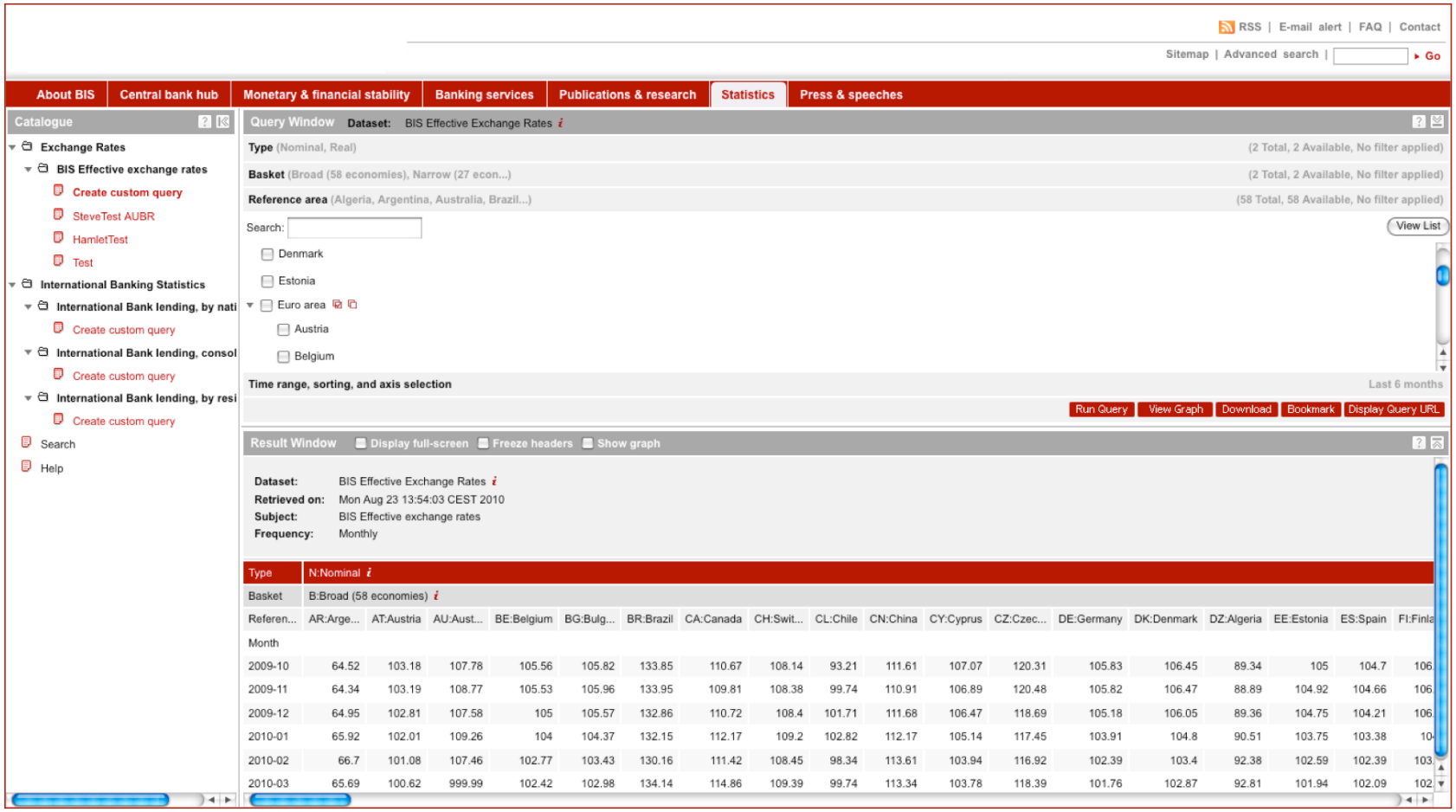
Low Level UI Components:

- They have also low granularity in the context of their container (a “High Level Component”).
- They know the data as a model, how to present it and how to work with it. They are mostly widgets.
- They know nothing about services and other elements outside the boundaries of their container.

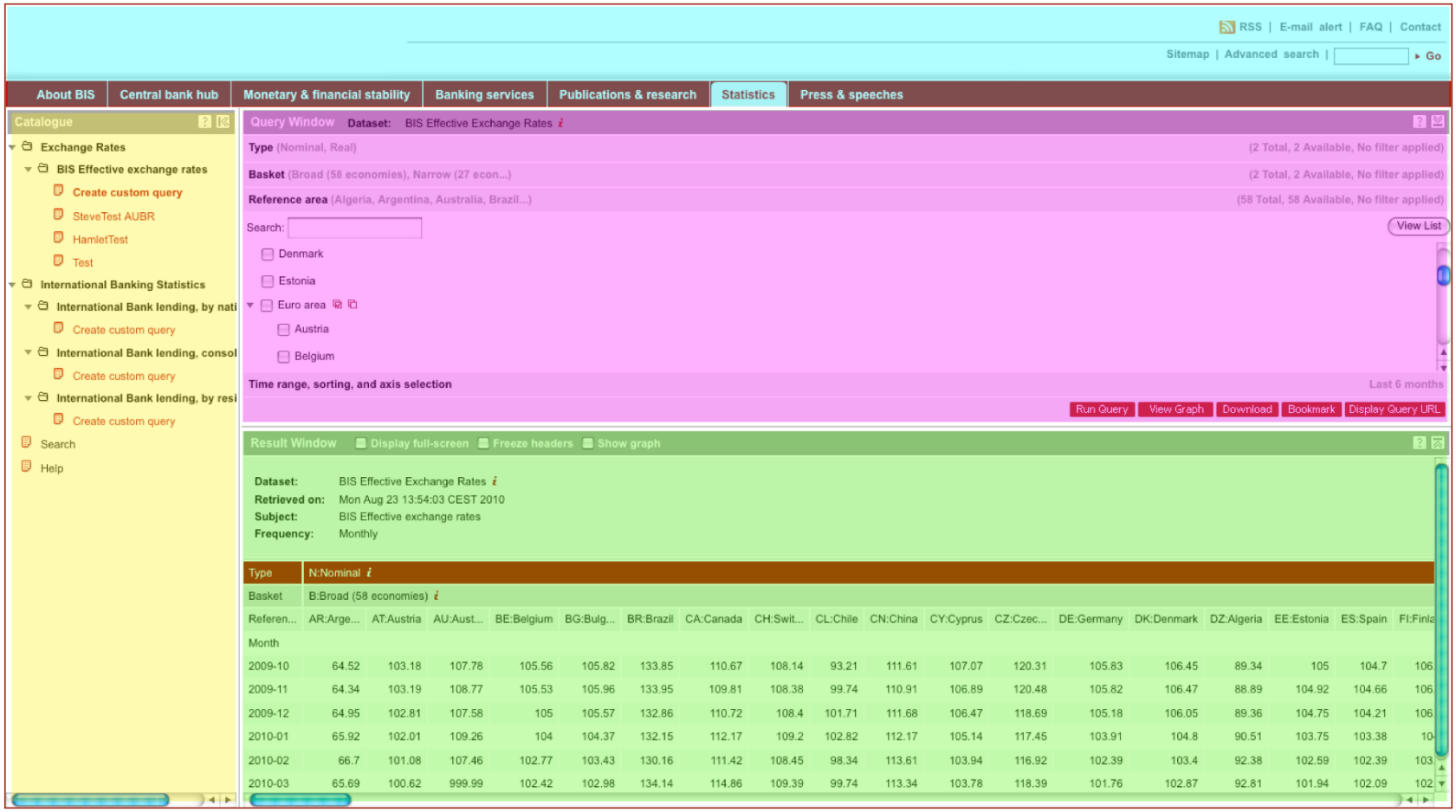
General Layout Manager:

- Controls the layout of the High Level UI Components and knows nothing about services, data or Low Level UI Components.
- Receives messages from the user through widgets integrated directly in itself or from the High Level UI Components holding widgets that ask for layout changes.

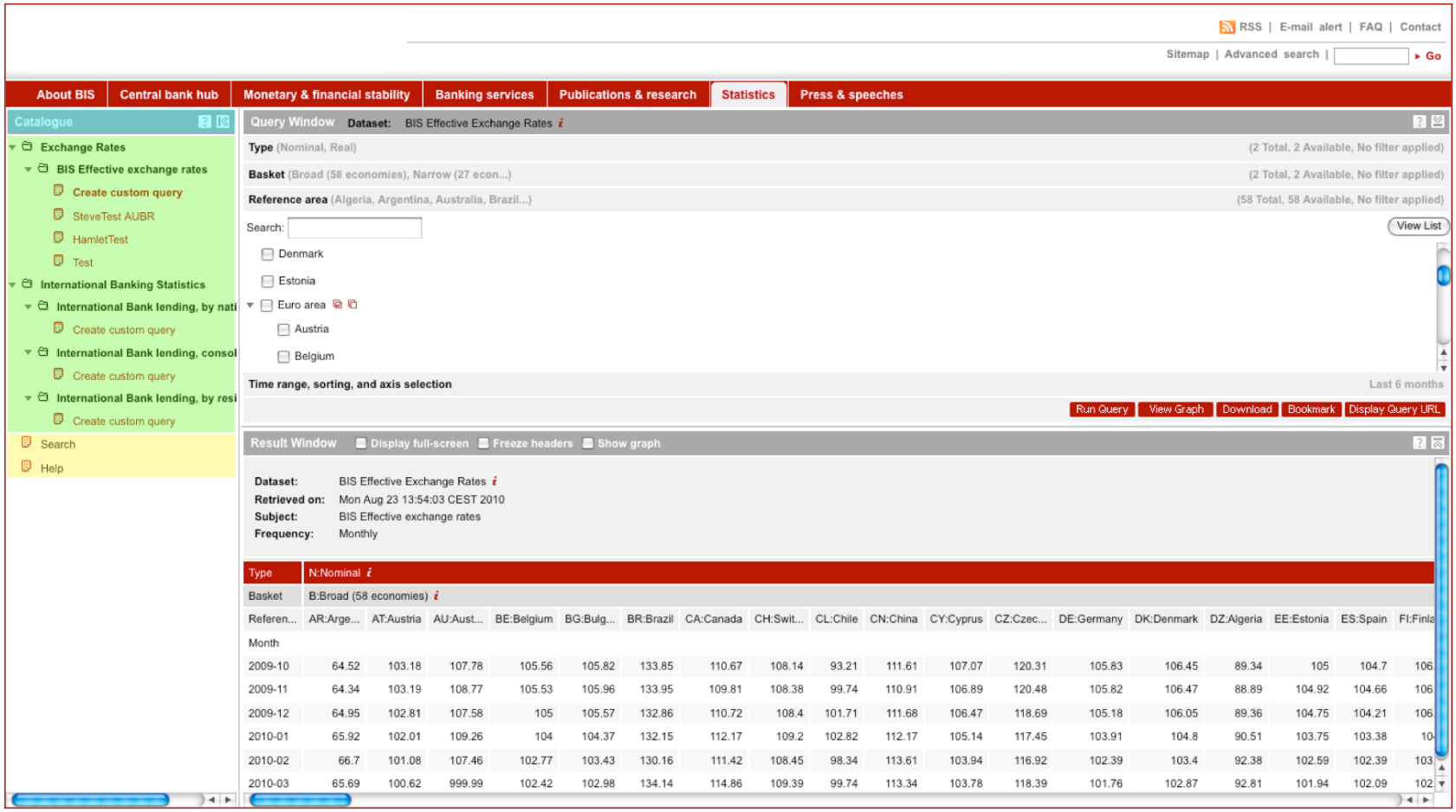
“Separate components in levels”



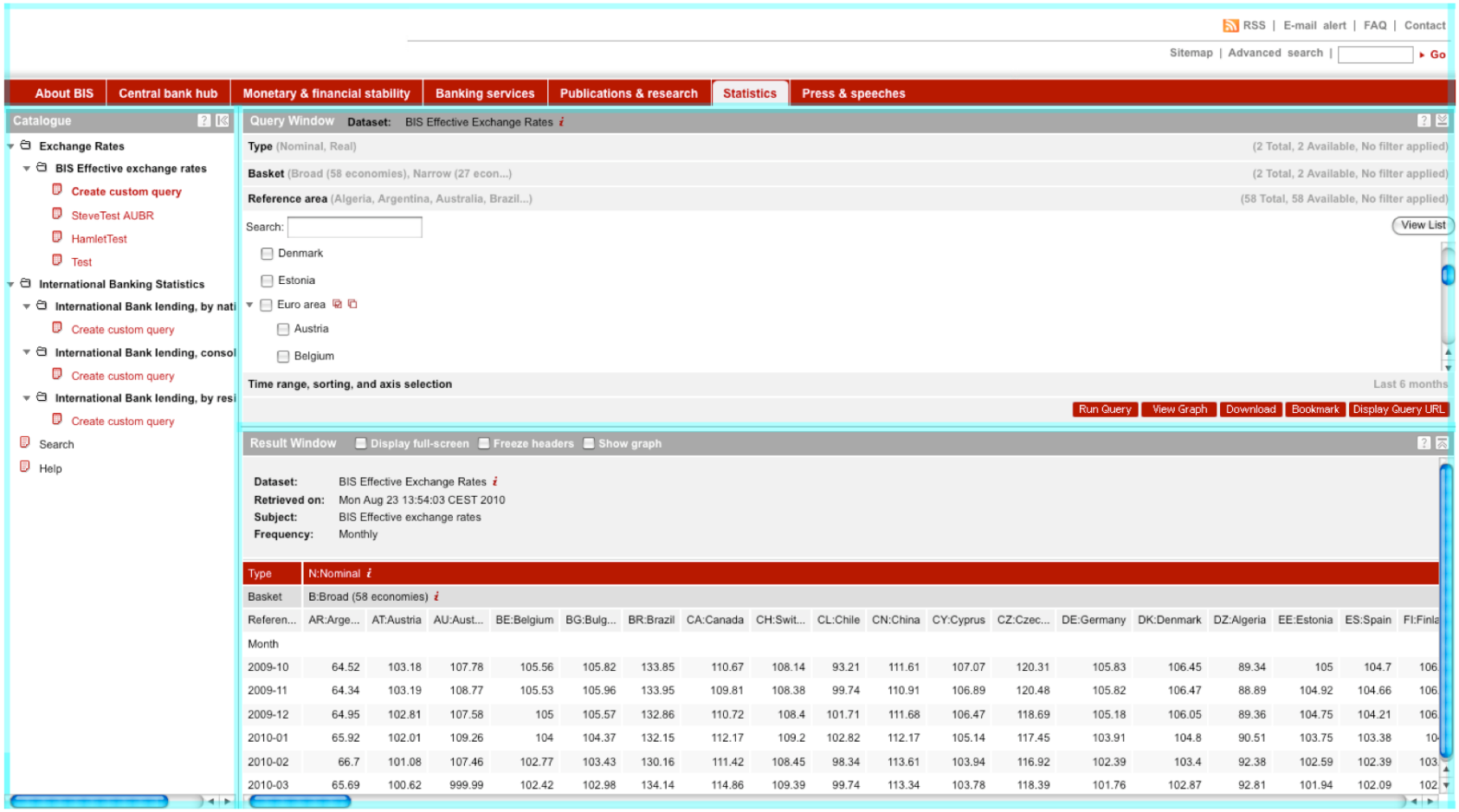
“High Level UI Components”



“Low Level UI Components”



“Layout Manager”



Recipe 2:

“If you have to cook vegetables, fish and meat, separate the flavors in different courses”.

You cannot really reduce the complexity.

Summary:

Things to Avoid in GWT when designing a UI:

- Do not rely only in widgets and event listeners. You need objects with different responsibilities and a clear concept to decide things like:
  - Where to store fetched data.
  - From where to call the backend services.
  - Which are the actions that change the “application state”.
  - How to react to application state changes, etc.
- Layout in Web applications (across different browsers) is too difficult without constraints. Having a general layout manager (that manages the placeholders) and isolated widgets (that try to get as much space as they can) divides the problem making it a little bit easier.

Why to use an approach like this:

- If you do not have boundaries, you are not aware that you are crossing them.
- The best way to handle complexity is by isolating parts and focusing on them (decomposition).
- Then, use different names with different rules and use them to assign different responsibilities.
- Because, if the application is simple enough to not need such an architecture, but it makes something useful, then new features will come and they will make the application too complicated to be maintainable without an architecture.

Recipe 3: how to cater for events.

Photo: © www.catermeisterei.net

“Differentiate the application events”

Avoid using listeners for “application level” events:

- When event listeners cross the different boundaries of your UI (High Level Components / Low Level Components) they couple everything. They can easily become difficult to maintain and refactor and it is difficult to distinguish to which level they belong (DOM events, widget events, application events...).
- Event listeners require properly registration, but JavaScript in old browsers is not fast or robust enough to deal with lots of JavaScript objects with ease (memory leaks and bad performance). For example: working with immutable objects requires unregistering the listeners when new immutable objects are created to substitute the old ones (better in GWT 2.x by using handlers instead of listeners).

What to use instead:

- For application level events, create an application “event bus” (use “HandlerManager” in GWT 2.x) and use a “publish / subscribe” scheme, allowing the High Level Components to register themselves with the events they are interested in. Only High Level Components handle application level events.
- Use a “payload” for the events that is based in your application’s domain model (Ex: “List<User>”).
- When the component responsible for it, makes a service call (Ex: “UserService.getAll()”), send an event to inform the components interested on this event that the request has been placed.
- When the result of the service call arrives, inform the interested components with another event passing the response of the service (Ex: “List<User>”).
- If the service call fails, use a flag or field on the previous event to inform about the request result.

“Differentiate the Application Events” (I)

```
HandlerManager eventBus = new HandlerManager(this);
ajaxSpinnerComponent.registerEvents(eventBus);
userListWindowComponent.registerEvents(eventBus);

…

public void registerEvents(HandlerManager eventBus) { // AjaxSpinner

  eventBus.addHandler(UserListRequestedEvent.TYPE,
    new UserListRequestedEventHandler() {
      public void onRequestStarted(UserListRequestedEvent event) {
        startSpinning();
      }
    });

  eventBus.addHandler(UserListRequestFinishedEvent.TYPE,
    new UserListRequestFinishedEventHandler() {
      public void onRequestFinished(UserListRequestFinishedEvent ev){
         stopSpinning();
      }
    });

}
```

“Differentiate the Application Events” (II)

```
public void registerEvents(HandlerManager eventBus) { // UserListWindow

  eventBus.addHandler(UserListRequestedEvent.TYPE,
    new UserListRequestedEventHandler() {
      public void onRequestStarted(UserListRequestedEvent event) {
        clearUsers();
        showMessage("Fetching users from server.");
      }
    });

  eventBus.addHandler(UserListRequestFinishedEvent.TYPE,
    new UserListRequestFinishedEventHandler() {
      public void onRequestFinished(UserListRequestFinishedEvent event){
        if (event.requestHasFailed()) {
          showMessage("No users could be fetched.");
        } else {
          showUsers(event.getUsers());
        }
      }
    });

}
```

“Differentiate the Application Events” (III)

```
new Button("Fetch users", new ClickHandler() {

  public void onClick(ClickEvent event) {

    eventBus.fireEvent(new UserListRequestedEvent());
    userService.getAll(new AsyncCallback<List<User>>() {

      public void onFailure(Throwable caught) {
        eventBus.fireEvent(new UserListRequestFinishedEvent(caught));
      }

      public void onSuccess(List<User> users) {
        eventBus.fireEvent(new UserListRequestFinishedEvent(users));
      }

    });

  }

});
```

“Differentiate the Application Events” (IV)

```
public class UserListRequestFinishedEvent extends
             GwtEvent<UserListRequestFinishedEventHandler> {

 …

 @Override
 protected void dispatch(UserListRequestFinishedEventHandler handler) {
   handler.onRequestFinished(this);
 }

}
```

Recipe 3:

“Not all events have the same relevance”.

An event bus enables communication without coupling.

Summary:

Not all events in your application have the same relevance. Even when every interaction can change the application state, there are changes that are more important than others:

- The user interacts with only one widget at a time.
- When not manageable within the widget, the widget communicates with the container (High Level UI Component) using a normal widget event listener (“onclick”).
- The High Level Component is who decides if it is an internal event or an external one (requires service calls or other components interaction or changes in the main layout, etc…).
- If it is an internal change, then no external action is needed. The component can itself handle this.
- If it is an external change, then it requires the execution of an application action and communicating the other components that something that could affect them is happening.

Recipe 4: what to do with calorie-rich meals.

Photo: © Dunkin Donuts

“Lots of data in old browsers?”

Showing lots of data using GWT is not difficult to implement (Ex: thousands of cells in a table) . The problem appears when you have to interact with the data and make it perform well in old browsers. Initially, the data widgets in GWT where not designed to use “lazy loading” and they were not even “model-oriented” (you had to set every value in every cell or internal widget).

Looping in old JavaScript engines is “sloooow” plus working with so many widgets as needed makes it even worse (poor garbage collection, memory leaks, …).

What to do:

- Render the result in HTML in the server and send it back to the browser.
- If the HTML is really big, optimize it before sending (minimize CSS styles and avoid superficial markup).
- Use "innerHTML" to integrate the response in the UI. It works as a DOM modification batch command and is a simple JS method well supported in old browsers.
- To make parts of the result interactive, register DOM handlers “at the right level”. This is normally only one handler at the HTML container where you called "innerHTML”.
- Be sure that, in the HTML generated, you can easily reach the data you need to invoke new services or application functionalities (make easy to call back from HTML into your GWT application).
- Design granular services, but if you know you are going to need some data for what is going to happen “immediately”, then grab as much as you can in one HTTP request. Reduce roundtrips wrapping granular services in bigger ones. If there is “static data” embed it in the host page (it can be tricky!).

“Server-side rendering + native event previewing”

```
service.getBigResult(new AsyncCallback<String>() {
   public void onSuccess(String htmlResult) {
      HTML html = new HTML();
      html.sinkEvents(Event.ONMOUSEOVER | Event.ONMOUSEOUT |
                      Event.ONCLICK);
      Event.addNativePreviewHandler(createNativeEventHandler());
      html.setHTML(htmlResult);
   }
});

private Event.NativePreviewHandler createNativeEventHandler() {
   return new Event.NativePreviewHandler() {

      public void onPreviewNativeEvent(Event.NativePreviewEvent event) {
         NativeEvent nativeEvent = event.getNativeEvent();
         Element target = nativeEvent.getEventTarget().cast();
         if (youAreInterestedOnTheEvent(event) || onTheTarget(target)) {
            // Do Something interesting with the element
         }
      }
   };
}
```

Recipe 4:

“Saccharin never tastes like sugar”.

Server-side rendering is sometimes necessary.

Summary

The browser knows no widgets, only a DOM.

Differentiating elements allows to assign responsibilities.

Avoid components coupling because of communication.

When something works 80%, it is said to be “not working”.

Thank you & *“Bon Appetit”*! :)