Cross-Platform, Functional, Reactive GUIs
Welcome!

CARLOS AGUAYO
SOFTWARE ENGINEER MANAGER
@carlosaguayo81
Modern UIs: Characteristics

Credit Card Information

1234 5678 9012 3456

Please enter your credit card number
Overview

At this lecture, you will:

- Learn about technology choices for building modern UIs
- Gain a deeper understanding about an effective programming paradigm to help you manage the complexity of these UIs
- See an example of how Appian applies these techniques
Definitions

**Cross Platform UI**: runs on multiple Web browsers and various mobile devices

**Functional Programming**: a style of programming that emphasizes the use of functions as first-class objects and avoids mutable state and side-effects

**Reactive Programming**: a style of programming oriented around data flow and propagation of changes triggered by events (think UI interactions)
Easy to Use Means Hard to Build!

- Complex interplay between parts of UI
- User interaction driven
- Asynchronous calls to one or more servers
- Data “pushed” from server
- Incremental rendering
- Client-side storage/processing
- Offline operation

```javascript
webserver.use browserChannel {webserver, sessi
numClients++
stream = new Duplex objectMode:yes
stream_write = (chunk, encoding, callback)
console.log 's->c ', JSON.stringify(chunk)
if client.state isnt 'closed' # silently d
client.send chunk
callback()
stream_read = -> # Ignore. You can't contro
stream.headers = client.headers
stream.remoteAddress = stream.address
client.on 'message', (data) ->
  console.log 'c->s ', JSON.stringify(data)
  stream.push data
stream.on 'error', (msg) ->
  client.stop()
client.on 'close', (reason) ->
  stream.push null
  stream.emit 'close'
  numClients--
  console.log 'client went away', numClients
stream.on 'end', ->
  client.close()
# ... and give the stream to ShareJS.
share.listen stream
webserver.use '/doc', share.rest()
port = argv.p or 7007
webserver.listen port
console.log "Listening on http://localhost:"+p
```
MV* Frameworks
More Choices...

TodoMVC
Helping you select an MV* framework

Download (1.3)  View on GitHub  Blog
What is easier to understand? Example #1

\[ f(a, b) = \frac{a \times 2}{b - 1} \]

Imperative Code

```
int f(int a, int b) {
    int num = a * 2;
    int den = b - 1;
    return num / den;
}
```

Declarative Notation
What is easier to understand? Example #2a

```javascript
var article = document.createElement("article");
var paragraph = document.createElement("p");
var text = document.createTextNode("Lots of text");
paragraph.appendChild(text);
article.appendChild(paragraph);
```

**Imperative Code**
article(
    p(
        "Lots of text"
    )
)

Imperative Code. Declarative API
What is easier to understand? Example #2c

```html
<article>
  <p>
    Lots of text
  </p>
</article>

Declarative Code
article.className = "pretty";
var header = document.createElement("h2");
var title = document.createTextNode("Awesome Title");
header.appendChild(title);
article.insertBefore(header, paragraph);
paragraph.setAttribute("lang","en");

**Imperative Mutations**
What is easier to understand? Example #3b

```html
<article class="pretty">
  <h2>Awesome Title!</h2>
  <p lang="en">
    Lots of text
  </p>
</article>
```

Declarative Code
(State after Mutations)
...Our intellectual powers are rather geared to master static relations. Our powers to visualize processes evolving in time are relatively poorly developed...

---  Dijkstra [EWD215]
What to do about it?

... For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible...

--- Dijkstra [EWD215]
Declarative Programming

- A style of programming that defines computation logic without defining its detailed control flow.

- Declarative programs are context-independent. Because they only declare what the ultimate goal is, but not the intermediary steps to reach that goal.

- Declarative programming is when you write your code in such a way that it describes what you want to do, and not how you want to do it.
$f(x)$
Functional Programming

- Promotes a declarative programming style through:
  - **First-class and higher-order functions** instead of explicit control flow
    - e.g. No while or for loops

```javascript
var sum = 0;
var numbers = [1,2,3,4,5];
for(var i = 0; i < numbers.length;i++) {
    sum += numbers[i];
}

var sumFn = function(a,b) {return a + b};
var sum = [1,2,3,4,5].reduce(sumFn);
```
Functional Programming

- Promotes a declarative programming style through:
  - Referentially transparent functions (pure functions)
    - free to memoize,
    - parallelize and
    - perform other optimizations

\[ |x| - 2 = \]

<table>
<thead>
<tr>
<th>Domain</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>-3</td>
<td>0</td>
</tr>
</tbody>
</table>
Functional Programming

- Promotes a declarative programming style through:
  - Referentially transparent functions (pure functions)
    - free to memoize,
    - parallelize and
    - perform other optimizations

\[
\frac{a \times 2}{b} = \frac{\text{mul}}{\text{div}}
\]
Functional Programming

- Promotes a declarative programming style through:
  - No mutable state
    - e.g. no arrays, no variables, no side-effects
    - safe to share state
    - free to cache state
    - no concurrency problems

ONE DOES NOT SIMPLY SHARE MUTABLE STATE
FRP is about declarative programming with *time-varying values* using functional building blocks.
In FRP, *time is implicit* as the code describes a UI at any moment in time.
FRP provides declarative control flow structures for events. There are no event handlers or callbacks.
Spreadsheets: Most popular FRP systems

<table>
<thead>
<tr>
<th>Person</th>
<th>Date</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Brust</td>
<td>15-Mar</td>
<td>$10.00</td>
</tr>
<tr>
<td>Bill Ryan</td>
<td>15-Mar</td>
<td>-$100.00</td>
</tr>
<tr>
<td>Bill Ryan</td>
<td>18-Mar</td>
<td>$5.00</td>
</tr>
<tr>
<td>Carl Franklin</td>
<td>18-Mar</td>
<td>-$20.00</td>
</tr>
<tr>
<td>Frans Bouma</td>
<td>16-Mar</td>
<td>$20.00</td>
</tr>
<tr>
<td>Mark Dunn</td>
<td>15-Mar</td>
<td>$40.00</td>
</tr>
<tr>
<td>Raymond</td>
<td>15-Mar</td>
<td>-$10.00</td>
</tr>
<tr>
<td>Stephen Forre</td>
<td>19-Mar</td>
<td>-$40.00</td>
</tr>
<tr>
<td>Tobin Titus</td>
<td>17-Mar</td>
<td>-$27.39</td>
</tr>
</tbody>
</table>

**Total CashFlow:** -$122.38

Cash Chart

- Andrew Brust 15-Mar
- Bill Ryan 15-Mar
- Carl Franklin 18-Mar
- Frans Bouma 16-Mar
- Mark Dunn 15-Mar
- Raymond 15-Mar
- Stephen Forre 19-Mar
- Tobin Titus 17-Mar
FRP at Appian

New Purchase Request

Requested By *

Department *

--- Select Department ---

Due Date *

11/20/2014

Special Instructions

Shipping Address *

City *

State/Province *

Zip/Postal Code *

Items

<table>
<thead>
<tr>
<th>Description</th>
<th>Category</th>
<th>Qty</th>
<th>Unit Price</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$0.00</td>
</tr>
</tbody>
</table>

Total $0.00

+ Add Item

Cancel

Submit
FRP at Appian

\[ n \leftarrow \text{“John”.Smith} \]
\[ p \leftarrow \text{“Secret”} \]
\[ f(n, p) \]

\[ n \leftarrow g(\text{”John.Smith”}) \]

1-Way Data Flow
Appian UIs are created from pure functions that describe the state of the UI at any point in time.
Data propagation changes are automatic and implicit. There are no manual updates of the UI.
Side effects can only be triggered by user interactions as part of a *unidirectional* data flow.
Why Functional UIs?

- **Easier to test & reason about**
  - Given some data (inputs), render a UI
  - Function describes state at any given point in time
  - UI functions only depend on inputs, not hidden state, not global state. i.e. stateless UIs

- **Composable**
  - UIs are assembled by declaratively gluing functions

- **Powerful**
  - Functions encapsulate business and UI logic
  - No separate, underpower, templating language
  - Templates are functions, placeholders are parameters
Why a One-Way Data Flow?

- **No cascading effects**
  - Views do not directly modify the model. They “ask” the FRP runtime to do it.
  - No dirty checking, no infinite loops checks

- **Single channel for state changes**
  - Good for Debugging
  - Good for Logging

- **All side-effects are controlled by the framework**
  - Allows Recording
  - Allows Replaying, etc
Summary

- Start with a declarative programming model
- Use functional composition to build higher-level abstractions
- Isolate and simplify side-effects in a unidirectional data flow
Cross-Platform Applications
HTML5 Apps
HTML5 Apps: Some Options

- Kendo UI
- Bootstrap
- jQuery
- Sencha
- UI
- Kendo UI
Hybrid Apps
Hybrid Apps
Hybrid Apps: Some Options

- PhoneGap
- Worklight
- ionic
- AppBuilder
- Apache Cordova
- TRIGGER IO FORGE
Compiled to Native Apps: Some Options

titanium™

Xamarin
Appian UI Platform (SAIL)

- Shared Libraries (Java)
  - GWT
  - j2objc
  - Android SDK
- iOS SDK
- JS / HTML / CSS
- HTML
  - <html5/>
- Android
  - <android/>
- Apple
  - <apple/>
Appian UI Platform (SAIL)
Patent-pending implementation using FRP
SAIL as a Productivity Multiplier

- Code-less design of modern GUIs
- Write once; deploy everywhere
- Specify intent, not platform-specific implementation
One UI Design: Multiple Clients
Native Components & Dynamic Behaviors

“Display a checkbox that, when checked, enables a button”

UI designer states intent. SAIL generates the code.

I agree to the terms of service
SAIL: Performance

Redraw Entire UI

Resend Entire UI

Reevaluate Entire UI

Browser

Network

Appian

Partial Rendering

Partial Updates

Partial Evaluation
Relevant Technologies

- React.js / Om
  - http://facebook.github.io/react/
  - https://github.com/swannodette/om
- Elm
  - http://elm-lang.org/
- Polymer
  - https://www.polymer-project.org/
Relevant Research

- Functional Reactive Animation
  http://conal.net/papers/icfp97/icfp97.pdf

- Asynchronous Functional Reactive Programming for GUIs

- Composable, Demand-Driven Incremental Computation