SPRING & CLOUD FOUNDRY
开发者大会

springone CHINA
中国北京 - 2012年12月7日-8日

IOC + JavaScript

Jeremy Grelle, SpringSource Staff Engineer

Github / Twitter: @jeremyg484

What?

cujo.js

- Spring-like concepts, but *not* a port of Spring to Javascript
- Embraces Javascript's functional and prototypal roots
- Provides architectural tools for next-generation JavaScript applications

http://cujojs.com/

How?

Code demos: Monty Hall UI && TodoMVC

Recent project stats

- 6 "HTML Pages"
- 300+ Javascript modules
- 100+ "View modules" each of which has:
  - HTML templates
  - CSS files
  - i18n bundles
  - test harnesses

Not including 3rd party modules!

Recent project stats

Manual dependency management is just not feasible at this scale

Help!

> *Larger, more complex apps require carefully crafted rigorous architecture, patterns, and code organization.*

-- Brian Cavalier

IOC

- We know from Spring, good architectural plumbing helps to manage complexity
- Javascript is no exception
- In the browser and on the server

Can we ...?

- Build smaller modules that are easier to maintain and test
- Separate configuration and connection from application logic
- Glue it all back together

Apply IOC concepts in Javascript

- Declarative component creation
- Lifecycle management
- Configuration
- Dependency injection
- AOP

IOC for front-end Javascript

- What might it look like?
  - XML? Not very Javascript-ish
  - Annotations? No existing infrastructure, need to parse source

- Javascript is flexible: Can we work *with* the language?

Components

The first thing we need is a way to build components

AMD

Asynchronous

Module

Definition

Three main parts

- Module format
- Run-time loader
- Build-time compiler (recommended for production)

AMD

Designed with browser environments in mind

- Loads asynchronously
- No parsing or transpiling needed
- Built-in closure
- Loads other resource types via plugins

Who supports it?

- dojo 1.7+
- cujo.js
- jQuery 1.7+
- MooTools 2+
- Lodash and many, many others

define()

AMD mandates a single, standardized global function.

```
define();
```

define()

```
define(factory);
define(dependencyList, factory);
```

AMD Module variants

"Standard" AMD

```
define(['when', 'pkg/mod'], function (when, mod) {
   // use the dependencies, return your module:
  return {};
});
```

AMD Module variants

AMD-wrapped CommonJS

```
/* no deps, factory params != 0 */
define(function (require, exports, module) {
  // sync, r-value require
  var when = require('when');
  // decorate your exports object
  exports.bestUtilEver = function (stuff) {
    return when(stuff);
  };
});
```

AMD Module variants

AMD-wrapped Node

```
/* no deps, factory params != 0 */
define(function (require, exports, module) {
  // sync, r-value require
  var when = require('when');
  // declare your exports
  module.exports = function bestUtilEver (stuff) {
    return when(stuff);
  };
});
```

UMD

Universal
Module
Definition

UMD

Boilerplate to sniff environment and export module correctly

- AMD + legacy globals
- AMD + CommonJS
- AMD + Node
- AMD + Node + legacy globals ...

UMD Module variants

AMD + Node (our favorite)

```
(function (define) {

define(function (require) {

  // module code goes here

});

})(typeof define === 'function' && define.amd
    ? define
    : function (factory) {
        module.exports =  factory(require);
      } );
```

UMD: AMD + Node

app/game/controller

(code demo)

Bootstrapping an AMD app

"The *other* global"

- `curl();`
- `requirejs();`
- `require();` (global `require` is problematic!)

Bootstrapping an AMD app

run.js

(code demo)

AMD Plugins

- Same dependency mechanism
- Non-AMD resources
- text! - HTML Templates and other text
- css! and link! - stylesheets
- i18n! - localization Google Maps
- JSON data, etc.

AMD Plugins

Can do even more powerful things

- wire! - wire.js IOC container integration
- has! - has.js feature detection and conditional module loading
- cs! - loads *and transpiles* Coffeescript

Plugins

app/main

(code demo)

CommonJS Module format

- Every file is a module with its own scope
- No closure, no factory, no `define()`
- `require`, `exports`, and `module` are "free variables"

curl.js <3 CJS!

"Compile to AMD"

Node !== CommonJS

- exports === this
- exports === module.exports

WTF

I know what you're thinking

> *Which one?!?*
> *Why are there 2 (3?) module formats, and how am I supposed to know which one to pick?!?!!*

It gets worse

ES Harmony modules are coming

Problem: Harmony is an *authoring format*. It doesn't handle:

- Dependency management
- Packaging, version management
- Compiling, concatenation
- Non-harmony resources (CSS, HTML, JSONP, etc.)

Relax

Relax

Your code is safe!

AMD consumes CJS and (soon) Harmony modules

Evolution

```
// curl.js config (coming soon!)
packages: [
  { name: 'node-thing', location: 'lib/node/thing', main: './main',
    transform: ['curl/transform/cjsm11'] },
  { name: 'cs-thing', location: 'lib/cs/thing', main: './init',
    transform: ['curl/transform/coffee'] },
  { name: 'future-thing', location: 'lib/harmony/stuff',  main: './main',
    transform: ['curl/transform/es7'] },     ... ],
```

CommonJS Modules today

monty-hall-ui/cjsm (branch)

Same modules written as CommonJS Modules/1.1, but unwrapped!

(code demo)

Micro-modules

Smaller is better!

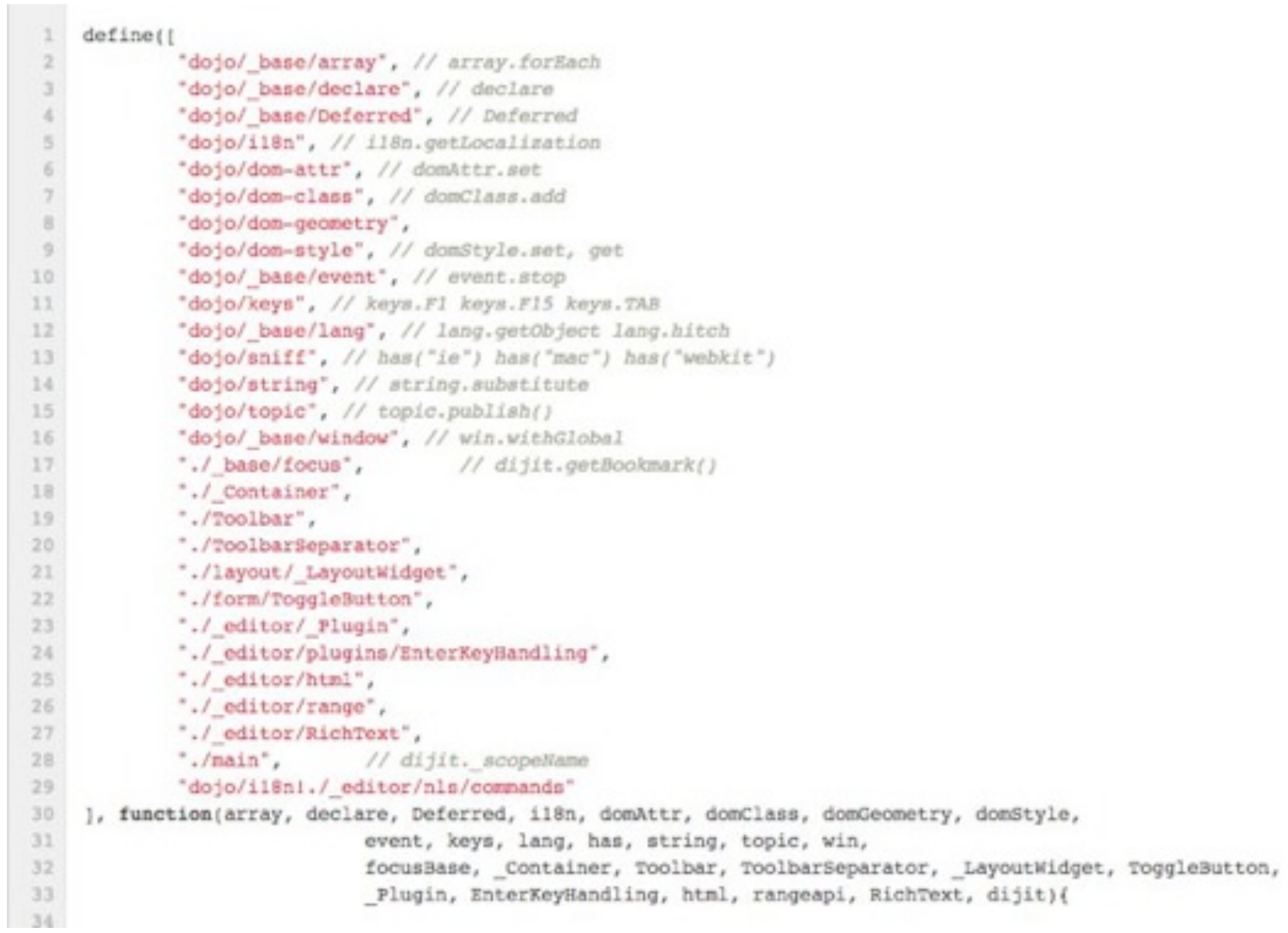
Micro-modules

Single-function modules are

- more reusable
- easier to test
- easier to discover

Hazard!

https://github.com/dojo/dijit/blob/ef9e7bf5df60a8a74f7e7a7eeaf859b9df3b0



```
define([
        "dojo/_base/array", // array.forEach
        "dojo/_base/declare", // declare
        "dojo/_base/Deferred", // Deferred
        "dojo/i18n", // i18n.getLocalization
        "dojo/dom-attr", // domAttr.set
        "dojo/dom-class", // domClass.add
        "dojo/dom-geometry",
        "dojo/dom-style", // domStyle.set, get
        "dojo/_base/event", // event.stop
        "dojo/keys", // keys.F1 keys.F15 keys.TAB
        "dojo/_base/lang", // lang.getObject lang.hitch
        "dojo/sniff", // has("ie") has("mac") has("webkit")
        "dojo/string", // string.substitute
        "dojo/topic", // topic.publish()
        "dojo/_base/window", // win.withGlobal
        "./_base/focus",        // dijit.getBookmark()
        "./_Container",
        "./Toolbar",
        "./ToolbarSeparator",
        "./layout/_LayoutWidget",
        "./form/ToggleButton",
        "./_editor/_Plugin",
        "./_editor/plugins/EnterKeyHandling",
        "./_editor/html",
        "./_editor/range",
        "./_editor/RichText",
        "./main",       // dijit._scopeName
        "dojo/i18n!./_editor/nls/commands"
], function(array, declare, Deferred, i18n, domAttr, domClass, domGeometry, domStyle,
                        event, keys, lang, has, string, topic, win,
                        focusBase, _Container, Toolbar, ToolbarSeparator, _LayoutWidget, ToggleButton,
                        _Plugin, EnterKeyHandling, html, rangeapi, RichText, dijit){
```

Hazard!

How do we avoid dependency hell when using micro-modules?

Connections

- The lines in your box-and-line diagrams
- Can be just as important as the stuff you put inside the boxes
- Unfortunately, we end up putting the lines inside the boxes

AMD

- Maintains good separation of concerns
- But more like Java `import`, which isn't necessarily right for all situations.

Example

```
define(['dojo/store/JsonRest'], function(JsonRest) {
  function Controller() {
    this.datastore = new JsonRest({ target: "mycart/items/" });
  }

  Controller.prototype = {
    addItem: function(thing) {
      return this.datastore.put(thing);
    },
    // ...
  }

  return Controller;
});
```

What's that smell?

- `this.datastore = new JsonRest(..)` is essentially a *line* inside our `Controller` *box*
- How would you unit test it?
- Could you use this with another type of data store?
- Multiple instances, each with a different type of store?
  - different target URL?

Refactor

```
define(function() { // No AMD deps!

   function Controller(datastore) {
     this.datastore = datastore;
   }

  Controller.prototype = {
    addItem: function(thing) {
      return this.datastore.put(thing);
    },
    // ...
  }

  return Controller;
});
```

Or Similarly

```
define(function() { // No AMD deps!

  // Rely on the IOC Container to beget new instances
  return {
    datastore: null,
    addItem: function(thing) {
      return this.datastore.put(thing);
    },
    // ...
  };
});
```

What did we do?

- Decoupled the concrete JsonRest implementation
- Refactored to rely on a datastore interface
  - Even though the interface is *implicit*

What did we accomplish?

- Moved the responsibility of *drawing the line* out of the Controller.
- Made Controller more flexible and easier to test

But we created a question

Who provides the datastore?

We know what to do

Dependency Injection in the Application Composition Layer

DI & Application Composition

```
define({
  controller: {
    create: 'myApp/controller',
    properties: {
      datastore: { $ref: 'datastore' }
    }
  },
  datastore: {
    create: 'dojo/store/JsonRest',
    properties: {
      target: 'things/'
    }
  }
});
```

The DOM

- Obviously, working with the DOM is a necessity in front-end Javascript
- Similar problems: lines inside the boxes

Example

```
define(['some/domLib'], function(domLib) {

   function ItemView() {
     this.domNode = domLib.byId('item-list');
   }

  ItemView.prototype = {
    render: function() {
      // Render into this.domNode
    }
  }

  return ItemView;
});
```

That same smell

- Depends on an HTML id, and a DOM selector library
- Changing the HTML could break the JS
- Have to mock the DOM selector lib

Refactor

```
define(function() { // No AMD deps!

  function ItemView(domNode) {
    this.domNode = domNode;
  }

  ItemView.prototype = {
    render: function() {
      // Render into this.domNode
    }
  }

  return ItemView;
});
```

Better

- Decouples DOM selection mechanism
- *and* HTML: Can inject a different DOM node w/o changing ItemView's source.

DOM & Application Composition

```
define({
  itemView: {
     create: {
       module: 'myApp/ItemView',
       args: { $ref: 'dom!item-list' }
     }
  },

  plugins: [
     { module: 'wire/dom' }
     // or { module: 'wire/sizzle' }
     // or { module: 'wire/dojo/dom' }
     // or { module: 'wire/jquery/dom' }
  ]
});
```

DOM Events

```
define(['some/domLib', some/domEventsLib'], function(domLib,  domEventsLib) {

  function Controller() {
    domEventsLib.on('click', domLib.byId('the-button'), this.addItem.bind(this));
  }

  Controller.prototype = {
    addItem: function(domEvent) {
      // Add the item to the cart
    }
  }

  return Controller;
});
```

That same smell, only worse!

- Depends on:
  - hardcoded event type,
  - HTML id,
  - DOM selection lib
  - DOM events lib
- More mocking

Refactor

```
define(function() { // No AMD deps!

  function Controller() {}

  Controller.prototype = {
    addItem: function(domEvent) {
      // Update the thing
    }
  }

  return Controller;
});
```

Better

- Only cares about a general event: "Now it's time to add the item to the cart"
- Different/multiple event types on multiple DOM nodes
- No hardcoded DOM selector: multiple
- Controller instances Only have to mock the `domEvent`, then call `addItem`

DOM Events & App Composition

```
itemViewRoot: { $ref: 'dom.first!.item-view'},

controller: {
  create: 'myApp/Controller',
  on: {
    itemViewRoot: {
      'click:button.add': 'addItem'
    }
  }
},

plugins: [
  { module: 'wire/on' }
  // or { module: 'wire/dojo/on' }
  // or { module: 'wire/jquery/on' },

  { module: 'wire/dom' }
]
```

JS-to-JS Connections

Can components collaborate in a more loosely coupled way than DI?

Synthetic events

- Javascript methods act like events
- "Connect" methods together
- Neither component has knowledge of the other

Example

```
Controller.prototype.addItem = function(domEvent) {...}

CartCountView.prototype.incrementCount = function() {...}
```

Using DI

```
controller: {
  create: 'myApp/cart/Controller',
  properties: {
    cartCountView: { $ref: 'cartCountView' }
  }
},

cartCountView: {
  create: 'myApp/cart/CartCountView'
}
```

Things we can improve

- Controller now dependent on CartCountView interface
- Have to mock CartCountView to unit test Controller
- What if there are other times we'd like to update the cart count?

Synthetic event connection

```
controller: {
  create: 'myApp/cart/Controller' },
  cartCountView: {
    create: 'myApp/cart/CartCountView',
    connect: {
      'controller.addItem': 'incrementCount'
    }
}
```

Better

- Application Composition layer makes the connection
- Controller no longer dependent on CartCountView
- Neither component needs to be re-unit tested when making this connection
  - Nor if the connection is removed later
  - Only need to re-run functional tests
- Could *completely remove* CartCountView simply by cutting it out of the Application Composition spec

Still not perfect

What if `addItem` throws or fails in some way?

AOP Connections

```
controller: {
  create: 'myApp/cart/Controller' },

  cartCountView: {
    create: 'myApp/cart/CartCountView',
    afterReturning: {
      'controller.addItem': 'incrementCount'
    }
}
```

Closer

- Only increment count on success
- What about failures?

AOP Connections

```
controller: {
  create: 'myApp/cart/Controller',
  afterReturning: {
    'addItem': 'cartCountView.incrementCount'
  },
  afterThrowing: {
    'addItem': 'someOtherComponent.showError'
  }
},

cartCountView: {
  create: 'myApp/cart/CartCountView'
},

someOtherComponent: // ...
```

Better! But not quite there

- More decoupled, testable, refactorable
- Still a level of coupling we can remove

Coupled parameters

```
function Controller() {}

Controller.prototype = {
  addItem: function(domEvent) {
    // How to find the item data, in order to add it?
  }
}
```

Coupled parameters

- Controller receives a `domEvent`, but must locate the associated data to update
- Need DOM traversal, and understand the DOM structure
  - data id or hash key hiding in a DOM attribute?
- Have to mock for unit testing

Coupled parameters

Controller only really cares about the item

Refactor

```
function Controller() {}

Controller.prototype = {
  addItem: function(item) {
    // Just add it
  }
}
```

Transform connections

Connections that can, um, *transform* data!

Transform function

```
define(function() {

  // Encapsulate the work of finding the item
  return function findItemFromEvent(domEvent) {
    // Find the item, then
    return item;
  }

});
```

App Composition

```
itemList: { $ref: 'dom.first!.item-list'},

findItem: { module: 'myApp/data/findItemFromEvent' }

controller: {
  create: 'myApp/Controller',
  on: {
    itemList: {
      'click:button.add': 'findItem | addItem'
    }
  }
}
```

Ahhh, at last

- Controller is easier to unit test
- Algorithm for finding the thing
    - can also be unit tested separately and more easily
    - can be changed separately from Controller
    - can be reused in other parts of the app

Awesome, we're done, right?

Not quite ...

What about asynchrony?

- Occurs most often at component and system boundaries
- Hence, connections often need to be asynchronous
  - Canonical example: XHR

Example

```
Controller.prototype.addItem = function(item, callback) {...}

CartCountView.prototype.incrementCount = function() {...}
```

Example

```
controller: {
          create: 'myApp/cart/Controller',
          afterReturning: {
                     'addItem': 'cartCountView.incrementCount'
          },
          afterThrowing: {
                     'addItem': 'someOtherComponent.showError'
          }
},

cartCountView: {
          create: 'myApp/cart/CartCountView'
},

someOtherComponent: // ...
```

Uh oh

- Moved the function result from the return value to the parameter list
- Since addItem can't *return* anything, `afterReturning` doesn't work!
- And how do we provide the callback?

Brief, asynchronous detour

- Javascript is designed around a single-threaded event loop
- Browser DOM events and network I/O are async
- SSJS platforms (Node, RingoJS, etc) are built around async I/O
- AMD module loading is async--the A in AMD!

Callbacks

The typical solution is callbacks, aka "Continuation Passing"

Example

```
// You wish!
var content = xhr('GET', '/stuff');
```

Add callback and error handler

```
xhr('GET', '/stuff',
          function(content) {
                     // do stuff
          },

          function(error) {
                     // handle error
          }
);
```

Callback infestation

```
// It's turtles all the way *up*
function getStuff(handleContent, handleError) {
          xhr('GET', '/stuff',
                     function(content) {
          // transform content somehow, then
          // (what happens if this throws?)
                                handleContent(content);
                     },
      function(error) {
          // Maybe parse error, then
          // (what happens if THIS throws?!?)
                                handleError(error);
                     }
          );
}
```

Async is messy

- Code quickly becomes deeply nested and harder to reason about
- Familiar programming idioms don't work
  - It's upside-down: Values and errors flow *down* the stack now rather than up.
  - Functions are no longer easily composable: `g(f(x))` doesn't work anymore
  - try/catch/finally, *or something reasonably similar* is impossible
- Callback and errback parameters must be added to every function signature that might eventually lead to an asynchronous operation
- Coordinating *multiple* async tasks is a pain

Promises

- Synchronization construct
- Not a new idea
- Similar to `java.util.concurrent.Future`
- Placeholder for a result or error that will materialize later.

Example

Return a promise, into which the content, or an error, will materialize.

```
function getStuff() {
    var promise = xhr('GET', '/stuff');
    return promise;
}
```

Promises

- Restore call-and-return semantics
  - Move function results back to the return value
  - Remove callback function signature pollution

- Provide an async analog to exception propagation It's right-side up

More about Promises

http://en.wikipedia.org/wiki/Futures_and_promises

http://blog.briancavalier.com/async-programming-part-1-its-messy

http://blog.briancavalier.com/async-programming-part-2-promises

http://github.com/cujojs/when/wiki

http://wiki.commonjs.org/wiki/Promises/A

Promises

- Several proposed standards
- Promises/A *defacto* standard
  - cujo.js: when.js
  - Dojo: dojo/Deferred
  - jQuery: $.Deferred (well, close enough)
  - Q
  - soon YUI, Ember

IOC + Promises

- Promises/A is an *integration standard* for asynchrony
- IOC is about gluing components together so they can collaborate
- Sounds like a match!

Refactor to return a promise

```
Controller.prototype.addItem = function(item) {
    // Asynchronously add the item, then
    return promise;
}
```

Promise-aware AOP

```
controller: {
          create: 'myApp/cart/Controller',
          afterResolving: {
                     'addItem': 'cartCountView.incrementCount'
          },
          afterRejecting: {
                     'addItem': 'someOtherComponent.showError'
          }
},

cartCountView: {
          create: 'myApp/cart/CartCountView'
},

someOtherComponent: // ...
```

Win

- Count will only be incremented after the item has been added successfully!
- If adding fails, show the error

Async without async

- Promise-aware AOP for async connections
- AMD loaders manage async module loading *and* dependency graph resolution.
- Promise-aware IOC container:
    - Integrate with AMD loader to load modules used in application composition specs.
    - Async component creation: constructor or plain function can return a promise
    - Async DI: component references are injected as promises for the components resolve
    - Component startup/shutdown methods can return a promise

Connections

- Implement application logic in components
- Connect components non-invasively via Application Composition
  - DI, events (DOM and JS), AOP, Promise-aware AOP

- Adapt APIs by transforming data along connections Enjoy the easier testing and refactoring :)

Organize!

Components, components, components

> *the "file tree on the left" actually became useful!*

-- Brian Cavalier

Organize!

Divide your app into *feature areas*

What are the *things* you talk about when you talk about the app?

Organize!

app/

(code demo)

Case: View-component

View-components consist of

- HTML(5) template (keep it simple!)
- CSS file (structural bits of OOCSS/SMACCS)
- i18n file(s)
- javascript controller (optional)
- test harness (also for design)
- any assets necessary for rendering
- any view-specific data transforms, validations, etc.
- wire spec (optional)

Case: View-component

app/instructions

(code demo)

Testing visual components

How?????

Testing visual components

Double-duty test harnesses

- Fixture for creating HTML and designing CSS
- Harness for user-driven tests
- Harness for unit tests

Testing visual components

Double-duty test harnesses

(code demo)

Unit tests

- Smaller is better
- Fewer dependencies means fewer mocks!

Unit tests

(code demo)

cujo.js

- AMD Modules - curl & cram
- IOC & Application Composition - wire
- Promises/A - when
- AOP - meld
- ES5 - poly
- Data binding - cola (alpha)

Alternatives

- AMD Modules - RequireJS, Dojo, lsjs, BravoJS
- IOC - AngularJS
- Promises - Q, Dojo
- AOP - Dojo
- ES5 - es5shim
- Data binding - Backbone, and everyone else

cujo.js

Get it at http://cujojs.com

Discussions, Announcements, Questions, etc.

https://groups.google.com/d/forum/cujojs

Questions?

Cloud Foundry 启动营

在www.cloudfoundry.com注册账号并成功上传应用程序，

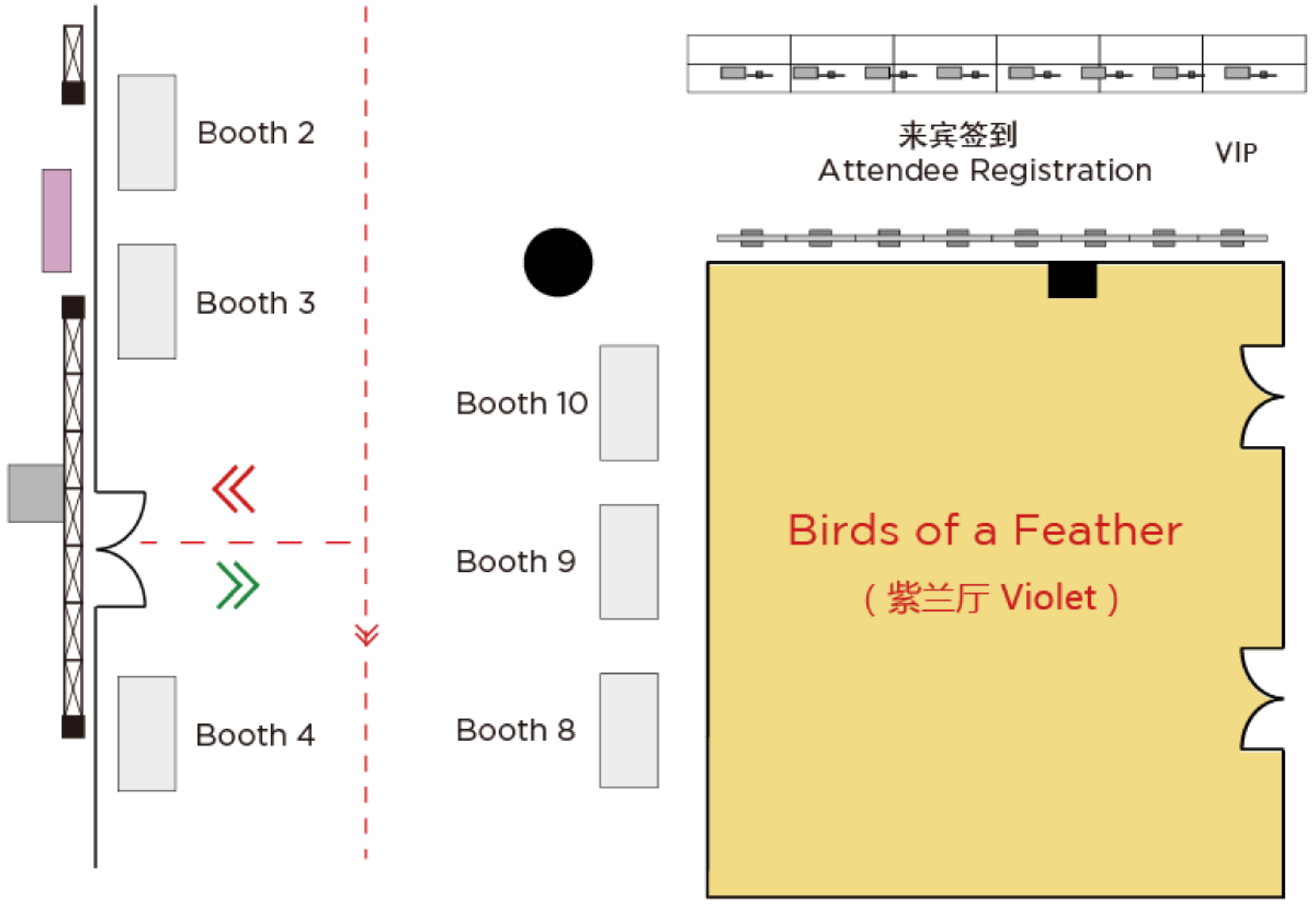
即可于12月8日中午后凭账号ID和应用URL到签到处换取Cloud Foundry主题卫衣一件。

iPhone5 等你拿

第二天大会结束前，请不要提前离开，将填写完整的意见反馈表投到签到处的抽奖箱内，即可参与“iPhone5”抽奖活动。

Birds of a Feather 专家面对面

所有讲师都会在课程结束后，到紫兰厅与来宾讨论课程上的问题


Booth 2
Booth 3
Booth 4
Booth 10
Booth 9
Booth 8
来宾签到
Attendee Registration
VIP
Birds of a Feather
（紫兰厅 Violet）