“Build Your First Mobile Application... in the Cloud... in 45 minutes.”

DIY Guide

Jay Marshall, Senior Systems Engineer
vFabric Cloud Application Platform
Welcome to the DIY guide for “Build Your First Mobile Application… in the Cloud… In 45 minutes”. This guide includes step-by-step instructions on how to do everything we did in the vFabric Blog, including how to sign up for your own Cloud Foundry account and how to download, install, and configure the SpringSource Tool Suite. By the time you are finished with this guide, you will be able to build and run your very first mobile application, in the cloud, on your very own device.

This document is meant to be a step-by-step guide focused around what we did in the blog video. It is not a formal introduction to any specific technology or associated terminologies. It also assumes you watched the video and consumed the talking points (i.e. when I reference specific technologies or themes they will hopefully be familiar).

**NOTE: As mentioned within the blog, this is not intended to be a best-practices guide on mobile development. The goal was to introduce developers, infrastructure teams, managers, and anyone else with mobile curiosities to a web-based approach to mobile development.**

This guide will be broken down into the following sections:

- Part One – Register with Cloud Foundry ........................................................................................................3
- Part Two – Download and Install the SpringSource Tool Suite (STS) .......................................................... 6
- Part Three – Build your first Grails Application ............................................................................................... 10
- Part Four – Build your first mobile application ................................................................................................ 26
- Part Five – Wrap-up .......................................................................................................................................... 49
Part One – Register with Cloud Foundry

1. Visit www.cloudfoundry.com and click “Register”.

2. Type in your email address and press “Request Invite”. That’s it! You will get a confirmation email with the details of your account (including initial password).
3. Once you log into your Cloud Foundry account, you will be given an option to “create a domain name”. This will be the subdomain on “cloudfoundry.com” where your application(s) will run. During the blog video session we used “challenges.cloudfoundry.com” for the standard database-backed web application, and “mobileapp.cloudfoundry.com” for mobile. You can use any name you want here (assuming, of course, no one else has taken it yet! For the purposes of this guide, you will want to create TWO subdomains: one for the “standard web application” and one for the “mobile app”. Since everyone’s domains will be different, for the purposes of this document we will refer to the standard web app as <WEB_APP>.cloudfoundry.com and the mobile app as <MOBILE_APP>.cloudfoundry.com.

Note: Although the screen states “cloudfoundry.me” and “Micro Cloud Foundry” on the site, the choices that you make here will also apply to “cloudfoundry.com” (i.e. if you enter “myapplication” in the box below and deploy it to the cloud, it will run in your browser at myapplication.cloudfoundry.com”).
That's it! You now have your account setup at Cloud Foundry with your own subdomains prepared to deliver your applications to. No middleware to setup, no database configuration, no hardware, no anything!
Part Two – Download and Install the SpringSource Tool Suite (STS)

1. Go to springsource.org and click on “Get Started”
2. On the subsequent page, click “Get Toolkit”:

3. Select the appropriate version of the SpringSource Tool Suite and download it to your computer.

4. Double-click the installer and leave default installation options in place. It may require a Java runtime if you do not have one, but will do all of it automatically for you.

5. At this point, STS is installed! Simply click (Mac) or double-click (PC) the SpringSource icon to launch your new workspace so we can write some code. (note: on a Mac, if you are having trouble locating your launch icon you can go to /YOU/springsource/sts-3.0.0.RELEASE to launch STS or drag the icon to your dock).

6. You will be presented with a “launch” window that prompts you for a workspace path. This is the root directory where STS will store all of your files for you. Enter a location and click “OK” to continue.

**NOTE:** I would suggest NOT clicking the “default” checkbox so that you can easily change your workspace location in the future, if necessary.
7. You will be presented with a blank STS opening page. We now need to add the necessary extensions for Grails and Cloud Foundry in order to use these features within STS.

**NOTE:** you will only need to do this the first time you launch STS. Once it’s setup you won’t have to do it again.

Click “Install Extension”.

8. From here, simply select the following options:
a. Under "Languages and Frameworks" - select "Grails (current production release).
b. Under "Language and Framework Tooling" select "Grails Support".
c. Under "Servers and Clouds" select "Cloud Foundry Integration for Eclipse".

9. When prompted, finish the wizard “as is” (i.e. allow unsigned content, etc.). You will be prompted to restart STS.

That’s it! Your SpringSource Tool Suite is now configured and ready for us to build our first application in Grails and Cloud Foundry!
Part Three – Build your first Grails Application

We now have our development tools and runtime environment configured and ready for use. As discussed in the blog, the first thing we want to do is build a standard web application backed by a database. We want to do this to emulate the common enterprise need to expose existing applications and data to our mobile devices. We will first build the application using the Grails development framework, and we will then expose the database data via a RESTful web service (also done via Grails).

1. Go ahead and close the “Welcome” tab by clicking the “X” to the right of the tab. This will take you to a blank workspace (note: we will not be covering terminology and details of the tools and frameworks used).

2. In the lower-left hand corner you will notice a window called “Servers”. Right-click anywhere in the whitespace of that window and select “New->Server”.

![Servers window with New Server option](image)
3. This will popup the new server wizard. Select “Cloud Foundry” for your server and click “Next”. 
4. Enter your email and password for your Cloud Foundry account. I strongly suggest that you use the “Validate Account” button when first adding a server. It is only one extra click, and it verifies you did not “fat finger” your account credentials. Otherwise, you will have to troubleshoot it later. Click “Finish” once you’ve entered your credentials and you are done. At this point you have successfully added a Cloud Foundry server to your development environment and should see it in the Servers window.

5. Double-click on the new Cloud Foundry server instance and it will open a panel in the main view. Once open, click on the “Applications” tab on the bottom. This shows a list of current applications and services (at this point it will be empty). We want to add a database service to our Cloud Foundry instance, so click on the “Add Service” icon to the right of the “services” text.

6. Name your database service whatever you want (we used “MyBlogDatabase” in the video), select “MySQL database service” as the type, and then “Finish” to create the service.
**NOTE:** This is how easy it is to get database instance setup and configured for your application! It should now show up under your list of “Services”.

7. Now we can start creating our application! On the left-hand side of the IDE under “Package Explorer”, right-click anywhere in the whitespace and select “New->Grails Project”. Give your project a name and press “Finish.”
**Note:** STS may ask you if you want to use the “Grails perspective”. This is simply a customized layout of the IDE that is intended to make Grails development easier. I would suggest clicking the checkbox that says “Remember my decision” and say “Yes”

8. STS will now setup your IDE so that you can begin writing code. In the lower right-hand side of the screen you will notice commands firing and a status bar. STS will create a whole structure for you that consists of domains, controllers, views, etc. Once again, we will not be going into the technical details of each of these in this document. But realize that this is an organizational structure to keep your development activities in check.

9. We want to start by creating a new domain object. The domain is what will be used for data persistence and is where we will define our data model. Expand your project by clicking the little arrow next to your project name in the project explorer. Right-click “domain” and select “Grails Tools->Create Domain Class…” In the subsequent popup, simply enter “Challenge” in the field provided and press “Finish”. We will create an application like we did in the blog to allow users to select their biggest IT challenges from a list and then report off of that data from a mobile device.
10. At this point Grails will create our Challenge domain object for us and bring it up in the editor. This will look very Java-like to those that have seen or programmed in Java. But this is where it gets fun. We want to enter firstName, lastName, and bigChallenge as String fields in our domain object. These will be the fields that store the data our users enter, and also the fields that get persisted to the database. In the line after the curly brace in class Challenge… And before the “static constraints” line… enter the following:

String firstName
String lastName
String bigChallenge

It’s that simple to declare your fields and expose them to you application and database. But we also want to make sure that we add validations so that users are required to enter values (i.e. they are not allowed to leave any fields blank). So we simply add a “blank” validation rule in the constraints section:

firstName(blank:false)
lASTName(blank:false)
bigChallenge(blank:false)
At this point your Groovy class should look like follows:

```groovy
package mywebapplication

class Challenge {
    String firstName
    String lastName
    String bigChallenge

    static constraints = {
        firstName(blank:false)
        lastName(blank:false)
        bigChallenge(blank:false)
    }
}
```

Now you can save your file by either hitting the “Save” icon in the upper-left part of the workspace, or simply hitting command or control S (Mac/PC).

11. Next we want to tell our Grails application where our database is located. Luckily, Grails is smart enough to know we are going to want to talk to a database. And it is also smart enough to know that it may very well be a MySQL database. So all of the plumbing is already in place for us to edit a few lines that were created for us and we will be ready to go.

12. First, under the package explorer, expand the “conf” folder under your project and open BuildConfig.groovy. Around line 37 (you can see the line number in the bottom-right portion of STS) you should see a commented out line of code like the following:

```groovy
// runtime 'mysql:mysql-connector-java:5.1.20'
```

Simply uncomment that line of code (remove the forward slashes). That tells Grails that we have a dependency on MySQL.

13. Save the file and close.
14. Open the file “DataSource.groovy”, also inside the “conf” folder. Change the “driverClassName” under “dataSource” to “com.mysql.jdbc.Driver”. This tells Grails which JDBC driver to use.

15. Set the “username” value of “datasource” to blank. This value is not important, as we will be deploying the application to Cloud Foundry and telling the application which database to bind to.

16. Under “environments”, completely remove the “development” and “test” entries (watch the curly braces so you don’t accidentally delete the wrong code!).

17. Under environments/production/dataSource, change the url to “jdbc:mysql://localhost/db”. This will tell the application to look locally for the database, which will once again be provided automatically by Cloud Foundry.
Your file should look like the following:

```java
DataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = ""
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = false
    cache.region.factory_class = 'net.sf.ehcache.hibernate.EhCacheRegionFactory'
}
// environment specific settings
environments {
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:mysql://localhost/db"
            pooled = true
            properties {
                maxActive = -1
                minEvictableIdleTimeMillis=1800000
                timeBetweenEvictionRunsMillis=1800000
                numTestsPerEvictionRun=3
                testOnBorrow=true
                testWhileIdle=true
                testOnReturn=true
                validationQuery="SELECT 1"
            }
        }
    }
}
```

18. Save and close the file.
19. We now need to tell Grails to generate our application for us. Open the Grails Command Wizard by right-clicking the Challenge domain object and selecting “Grails Tools->Grails Command Wizard…”

20. Scroll down and select “generate-all” and click “Next”. 
21. At this point we need to select the domain object we are going to generate off of. Simply click the “Browse” button and type in “Challenge”. This should show you the domain object you created earlier. Select it and press “OK”. When control returns to the wizard simply press “Finish” and Grails will generate your application for you.

22. When Grails is finished, you should have the “ChallengeController.groovy” class in front of you. This is the code that Grails generated for you based on the information provided. Once again, if you are new to Groovy or to Java it may look a bit odd at first. But it is like any other technology or any other language. Once you get the hang of it, it becomes normal.

23. While we are here, we are going to go ahead and expose a RESTful web service out of our application so that we can return JSON data to our mobile client in the next phase of our development. You will notice that each of the methods in your controller class start with “def” and are enclosed in curly braces. To expose a REST service called “mobilefeed” that returns all of our data as JSON, it’s as simple as adding the following code before the “def index()” method:

```groovy
def mobilefeed(){
    render Challenge.list() as JSON
}
```

You will get an error because Grails doesn’t know what to do with that “JSON”-thing you put in there. All
you have to do is add a reference to the proper JSON class and it will fix itself. Simply right-click on the little lightbulb where the error is and select grails.converters.JSON from the list and the error will go away.

24. Save and close

That’s it! Grails takes care of all of the heavy lifting for your web service!

25. Let’s go ahead and add the dropdown code for the data entry piece. Open up the "_form.gsp" file inside views->challenge. This should look like pretty common HTML code with the exception of some custom tags. Look for the following tag:

```html
<g:textField name="bigChallenge" required=""
value="${challengesInstance?.bigChallenge}" />
```

and replace the entire thing with the following:

```html
<g:select name="bigChallenge" from="["Big Data", 'Cloud', 'Mobile', 'Social', 'polyglot', 'What problems?']"
value="${challengesInstance?.bigChallenge}" noSelection="'Please select'" />
```

This will give the user the ability to choose from a list of options for their big challenge!

**Save** and close the file.
Now let’s go run our app!

26. Simply grab the main project folder for your project (the one you created in step 7 when you created your Grails project), drag and drop it right onto your Cloud Foundry server, and it should launch the application deployment wizard:

![Application deployment wizard](image)

27. Enter the `<WEB_APP>` domain name that you created in Cloud Foundry for your standard web application and press “Next”. This will present the launch deployment details screen:

![Launch deployment details](image)

My suggestion is that you accept the defaults here unless you truly think your application requires more memory. Press “Next”.

22
28. Here is where you select which services you would like to bind to your application. Since we know we want to use our MySQL database service, go ahead and select it and press “Finish”.

![Services selection](image)

Grails will now build and package your application, upload it to your Cloud Foundry account, and install and start the application for you automatically. The entire process should take a minute or two tops (depending on your internet connection speed). STS will stop with status messages and the following text will indicate that your application has installed and is ready for consumption:

**INFO: Server startup in 7772 ms**

At this point you can open a browser and point it to `<WEB_APP>.cloudfoundry.com` and your application should be running!
Go ahead and navigate to “ChallengeController” and play around. Add some data. Go back and look at what the data lists look like. If you point your browser to:

<WEB_APP>.cloudfoundry.com/challenge/mobilefeed you will see the JSON data that will be consumed by your mobile application in the next session.

As a side note… I would just like to point out that we have done the equivalent of installing and setting up an application server, database server, complete CRUD application development over one table, all domain wiring, and full deployment running in a browser anywhere in the world in next to no time.
Housecleaning: Once you have verified your application is working and that you do not need to modify your code anymore, feel free to close out any open editors (e.g. “ChallengeController”, “Challenge”) to get a clean slate. Over in the “Project Explorer” tab, you will notice a “collapse all” icon (shown below). Click it to collapse the entire Grails project. This will start the next session with a clean slate.
Part Four – Build your first Mobile Application

So we’ve registered for Cloud Foundry and we’ve used the SpringSource Tool Suite. And we have now written an application using the Grails framework, and even exposed our first REST service. Now it’s time to go mobile with JavaScript. We will be using the same SpringSource Tool Suite to develop our application, and we will be deploying to Cloud Foundry:

1. First, we will need to download the Sencha Touch JavaScript framework. This is the framework that is going to give us the native looking and feeling user experience on our mobile devices. Visit [http://www.sencha.com/products/touch/download/](http://www.sencha.com/products/touch/download/). The Open Source Version will work for our purposes. They also have some nice tools to help visually build your apps. But for now we are just gonna code!

2. There are lots of great tutorials, examples, and follow along guides at the Sencha site, on YouTube, etc. But we are going to boil it down to super basics and just grab the files we need and start. If you unzip the package you just downloaded, there will be a JavaScript file called “sencha-touch-all.js” right in the root of the distribution. There is also a CSS file called “sencha-touch.css” in the “/resources/css” subfolder. Locate these two files and be prepared to access these momentarily.

**NOTE:** As I have mentioned many times in the blog and in this document, the primary goal here is to get you down and dirty quickly writing code and seeing results. This will not be formal education on the Sencha Touch framework.

3. The standard web application that we built is intended to represent an existing application within our organization. As such, it would most likely have its own codebase, it’s own project, etc. So we are going to emulate that within STS by starting a brand new project. In the last application, we created a new Grails project. Since we know we want to use web technologies to build this one, let’s go ahead and right-click in the whitespace in the package explorer and select “New->Project...” and when the window pops up, scroll down to “Web->Dynamic Web Project” and click “Next”.

26
4. This will bring up a wizard to configure the project you are about to create. You can simply enter the name of your project (e.g. “MyMobileApp”), verify that it has “Cloud Foundry” specified as it’s runtime, and click “Finish”.

You have now created your mobile workspace.

5. This type of project defaults to a JEE perspective (remember, we used a “Grails” perspective last time). It can be a little bit difficult on the eyes, so let’s simplify things a little bit by opening up STS’s default “Spring” perspective. Towards the upper-right hand corner of the workspace click the “Spring” icon (shown below) to go back to the default “Spring” perspective.
6. Now it’s time to use those Sencha files. Expand your mobile project folder, and also the WebContent folder inside. Grab the “sencha-touch-all.js” and “sencha-touch.css” files and drop them right in the “WebContent” folder (shown below).

![Image of Package Explorer with Sencha files]

7. We also want to create our own JavaScript source file called “app.js”. This will be the file where we actually write our application. In STS, right-click on the “WebContent” folder and select “New->File”.

![Image of STS with New File option selected]
When the dialogue box pops up, simply call your file “app.js” and click “Finish”.

**NOTE:** *The file will initially be completely empty. Once again, this is where all of our code is going to go.*

8. Now we can create our home page. If your “WebContent” folder has an “index.html” file, that is the one that will get loaded automatically by Cloud Foundry when your application is launched. So let’s create our “index.html”. We’ll try this with no screenshots. Right-click on the “WebContent” folder and select “New->Other”. In the subsequent popup select “Web->HTML File” and press “Next”. Name your file “index.html” and press “Finish”. This will create a shell of an HTML document.

9. This is going to be the simplest HTML document you have ever seen! As mentioned above, all of our code is going into the “app.js” JavaScript file. So literally all this HTML document is going to do is link to the CSS and JavaScript files that we need to reference. Add the following links in the HTML document before the closing of the <head> tag (i.e. before </head>):

```html
<link type="text/css" href="sencha-touch.css" rel="stylesheet"/>
<script type="text/javascript" src="sencha-touch-all.js"></script>
<script type="text/javascript" src="app.js"></script>
```

So when we launch our application on Cloud Foundry, it will automatically load the index.html page, which will in turn automatically reference those files. The only other thing we want to do is change the title of the HTML. This will be the application name that will show up on your phone (note: you are typically limited to 10-to-12 characters on the actual physical device itself, so try to pick a short name):

```html
<title>Challenges</title>
```

10. Save and close index.html

11. Now it is time to start writing our mobile application in JavaScript. Go ahead and open the “app.js” file we created. Although the goal of this document is not to “teach” any particular language, I do want to point out that coding in JavaScript (and Sencha Touch in particular) can turn some folks sideways due to its unorthodox scripting nature compared to other more traditional languages. Developing with the Sencha Touch framework tends to consist of a lot of nesting that can appear to get out of control very quickly. But we will start by illustrating some basic approaches to building out the first parts of the application, and then keep adding code and hopefully make it easier to develop an approach to writing this kind of code.

Let’s start by simply creating an instance a Sencha Touch application in our “app.js” document:

```javascript
Ext.application({});
```

As simple as that looks, I want to illustrate the fact that this is where it all begins. A simple function call to “application()” from the core “Ext” class of the JavaScript framework, terminated with a semicolon. The curly braces are where we will pass any configuration options we want the application to have.
(separated by a comma). These configuration options will tell the application what we want it to do. This is a very important concept and something that will be recurring throughout developing our app. We will constantly be nesting configuration options inside of function calls. And some of those will get nested inside of other function calls. Etc etc etc.

Now obviously we have not passed anything into “application()” yet, so it won’t be a very exciting application. 😊 But I felt this was a very important place to start to illustrate a starting point.

So let’s start passing our parameters inside our curly braces. We are going to give our application a “name”, and we are also going to define a “launch” config option, which will actually be its own function. This is the function that gets called automatically by the framework when the application is first launched. If you put your cursor between the two curly braces and press “Enter”, it will automatically put the proper spaces in and you can start typing. Our config options will vary based on what type they are. For name, we can simply add a name value pair as follows:

   name: 'MyMobileApp',

Add a comma, press enter, and our config option “launch” will actually be its own function with its own set of curly braces so that we can pass it it’s own config options:

   launch: function(){
  }

So your entire “app.js” codebase should now look like this:

Ext.application({
   name: 'MyMobileApp',
   launch: function(){
  }
});

Your application will now call the function() above and execute whatever is inside. So let’s make it do something. As you saw with “Ext.application()” above, the Sencha framework has utility classes that do pretty much everything for you. A vast majority of learning Sencha is learning which classes do what, what config options they accept, and whether those config options accept other parameters, other classes, etc. What we want to do is actually add a “Panel” to our application. This is the panel that will house whatever it is we want to display to our users (remember, right now we just have a blank application!).

So we will start the same way we started our application. We will use the “Ext” core class again and start with the basics. Inside of the curly braces in our function, add the following:

   Ext.create();
Pretty simple, eh? Well... it’s not going to look so simple in a minute, and that’s why I wanted to call attention to it. Notice this does not have the curly braces that our application() call did. That’s because we were only passing the application() method a list of config options. With our call to create we want to pass it two things: what we want it to create and a list of configuration objects for what we are creating. Now let’s pass “Ext.Panel” to let it know to create a panel, and then let’s tell it do be fullscreen:

```javascript
Ext.create('Ext.Panel',{  
    fullscreen: true  
});
```

You can start to see this “nesting” concept I am talking about. So let’s finish our first round of coding by adding a little bit of HTML that we can save and run on our phone! (Hint: don’t forget to put a comma in after your “fullscreen” config option before adding the next config option)

```javascript
Ext.create('Ext.Panel',{  
    fullscreen: true,  
    html: 'Hello Cloud Foundry!'  
});
```

A couple quick things to point out: Notice that the config option fullscreen has a value of true, and it is not surrounded in single quotes. But the config option for html has a value of 'Hello Cloud Foundry!' and is surrounded in single quotes. Boolean values and numbers do not require the single quotes. Strings do. And it is single quotes, NOT double-quotes (another “gotcha” for many programmers). Just these couple quick tips could save you TONS of time debugging your code.

Save “app.js”.

At this time your entire “app.js” class should look like this:

```javascript
Ext.application({  
    name: 'MyMobileApp',  
    launch: function(){  
        Ext.create('Ext.Panel',{  
            fullscreen: true,  
            html: 'Hello Cloud Foundry!'  
        });  
    }  
});
```

It should be VERY obvious why I started with just that one line. The “nesting” nature of JavaScript (and Sencha specifically) is usually the biggest obstacle to using these types of frameworks. But once you get the hang of it, you will learn that the biggest challenge is learning which Ext classes you want to use, what method calls to make, and what config options they require.
10. So let’s run our app! We want to run this in a completely different instance of Cloud Foundry so that we can emulate a mobile application running somewhere completely separate from our standard internal web application.

Right-click on your “index.html” file and select “Run As->Run On Server…”
This will launch a wizard to assist deploying the application. Leave the “Choose and Existing Server” option selected and select your “VMware Cloud Foundry” server that you defined previously. I would suggest clicking the “Always use this server when running this project” checkbox at the bottom of the wizard.

![Run On Server](image)

**NOTE:** Do not be confused when I say that we need to run this in a separate “instance” of Cloud Foundry, and then I have you select the same “server” as the Grails application. In both cases you are targeting your account at CloudFoundry. But you will install this on `<MOBILE_APP>.cloudfoundry.com`, which is running completely separate from `<WEB_APP>.cloudfoundry.com`.

Click “Finish”.

You will be presented with the same Cloud Foundry options you were given in the earlier Grails application. This time, enter your `<MOBILE_APP>` subdomain you created at Cloud Foundry and click “Next”. **DO NOT SELECT ANY ADDITIONAL SERVICES THIS TIME.** Remember, we want to use our Grails REST service to feed data to our mobile application.
That’s it! Your very first mobile application is ready to view. Go to your browser on your mobile phone and go to your <MOBILE_APP>.cloudfoundry.com domain you created on Cloud Foundry and check out your app.

Congratulations! You have written your first mobile application!

We took our time writing our “Hello Cloud Foundry!” application so that we could illustrate the nested-nature of this environment. If any of what we just did was confusing, don’t be alarmed. It will feel more natural as you go. I would suggest that you completely wipe out the entire contents of “app.js” and start over again at Step 11 just to get more comfortable. The rest of this guide will not have the same step-by-step feel, so it is important to have a decent grasp of the concepts above.

11. Instead of a simple “Hello Cloud Foundry” app, let’s go ahead and display a list of data. We are not going to worry about talking to our Grails app yet. Let’s just focus on getting a list of data on the screen that looks, feels, and acts like a real mobile application.

To do this, instead of simply creating blank panel using “Ext.Panel”, we want to display a listing of data using “Ext.List”. You can start by simply swapping those two values out on your current application. We are also not going to use the html tag anymore, as we want to dynamically create the list based on a data source. Replace the html option with the following:

```
itemTpl: '{firstName}'
```

itemTpl is a config property for controlling list layouts in Sencha. And we are going to display a list of first names in our application (okay, so it’s nothing fancy, but it’s a start!). The curly braces here are essentially identifying the property name that the itemTpl should be looking for in our data source (more later).

Now we need to give our list some data. As mentioned before, we are going to start with hardcoded data in the JavaScript file. We are going to use the config property “data” to define our data. But unlike our other properties, we are going to nest our data elements inside a set of brackets. Let’s start by simply defining the property before we start nesting the data.
Enter the following after the itemTpl property (don’t forget to add a comma!):

```javascript
    data:[
    ]
```

Think about what we need to do next. We need to define each data element that we are going to display in the list, right? And we know from our itemTpl attribute that it is going to be looking for a property called firstName. For each “record” we need to encapsulate the name/value pairs in a set of curly braces. So each entry would look like this:

```javascript
    {firstName:'Jay'}
```

And we know that we need to separate each record with commas. So let’s go ahead and add five first names to our data config. Save the “app.js” file.

That’s it! Your complete code should look like this:

```javascript
Ext.application({
    name: 'MyMobileApp',
    launch: function(){
        Ext.create('Ext.List',{ 
            Ext.create('Ext.List',{
                fullscreen: true,
                itemTpl: '{firstName}',
                data:[
                    {firstName:'Jay'},
                    {firstName:'Andy'},
                    {firstName:'Pat'},
                    {firstName:'Susan'},
                    {firstName:'Mary'}
                ]
            }));
        }
    });
});
```
We now need to refresh our application out on Cloud Foundry. Follow these instructions to update and restart your application:

- Double-click on the “Servers” tab in the lower left-hand corner of the screen, this should expand and take over the entire workspace.
- Click the arrow to the left of the “VMware Cloud Foundry” server. This should expand and show you your list of applications.
- Click on your mobile application
- Right-click and select “Update and Restart”
- Wait until the workspace indicates the restart has completed
- Double-click the “Servers” tab again to restore your workspace to its normal state.

NOTE: From this point forward, anytime I say UPDATE AND RESTART your application, repeat the six steps above.

Go back to your mobile device and refresh your browser. Your application should now have the data you coded above.

It is important to point out the look-and-feel of the application. If you swipe the screen up and down, you will notice that it has that “floating” feel that mobile applications do. If you touch one of the rows, it responds and highlights with a slight gradient the way mobile applications do. As simple of an example as this is, this illustrates the “mobile UX” concepts discussed during the session.

12. We are now going to wire the data coming out of our Grails application we built initially to our mobile application. Instead of building out the additional modules piece-by-piece like we’ve done so far, we will instead share the blocks of code that we are going to add to the “app.js” file, and then explain what we did.

First, we need to add what’s called a “Model” to our application. We know that we have data coming out of our Grails application. We know that it’s in JSON format. And we know the URL that it is coming out of. But that is a separate application living on a separate application server that may have been built by a different team of developers and is being exposed by a completely different group of people. What is my point? We need to define the data that is coming out of the REST service we are calling for our mobile application so that we know what data we have. That is the purpose of a “model” in this context.

Here is our model code (place it right after our “launch: function” line and before the code to create the list):

```javascript
// define our model
Ext.define('Challenge',{  
    extend:'Ext.data.Model',  
    config:{  
        fields:[  
            {name:'firstName',type:'string'},  
            {name:'lastName',type:'string'},  
            {name:'bigChallenge',type:'string'}  
        ]  
    }  
});
```
This should at least be starting to look familiar. We started with Ext.define(); just like we did Ext.application(); and Ext.create(). ‘Challenge’ is the name of the model. That is important as we will need that later when we are populating data. And you can see the configuration is very similar to what we did when creating our Ext.List. The major difference is when we start adding fields. That is very similar to our data config property of our list.

The two forward-slashes before “define our model” makes that a comment and gets ignored by the browser.

So once again, much of this is about syntax and just learning how Sencha (or any other JavaScript framework) expects to be configured.

13. Now let’s add something called a “store”. For the purposes of this document, a store is simply telling Sencha where to get the data, and what model to load the data to. So this is where we will go ahead and tell our application to go to our Grails application to get the data, and then we will reference our “Challenge” model object above when loading data for display. Go ahead and write the following code immediately after our model code we just wrote:

```
// create our store
Ext.create('Ext.data.Store',{ 
    model: 'Challenge', 
    proxy: { 
        type: 'jsonp', 
        url: 'http://<WEB_APP>.cloudfoundry.com/challenge/mobilefeed', 
        callbackKey: 'myMobileKey'
    }, 
    autoLoad: true
});
```

I want to take this opportunity to point out that I would start all coding like this with the following line:

```
Ext.create();
```

…and then go into the parens and add ‘Ext.data.Store’, the comma, the open curly brace, and then hit “Enter”. That would result in the following code:

```
Ext.create('Ext.data.Store',{ 
});
```

I wanted to revisit this one more time because the SpringSource Tool Suite will assist you when writing JavaScript code to keep things nice and formatted. And that is many time the biggest battle when trying to write this code. By being patient and taking the coding step-by-step and watching your braces, commas, etc. it can make this much much easier and practically code itself.
Also notice that the ‘Challenge’ model created above is now being referenced by our store and the "autoLoad" feature is basically going to grab all of the data upfront and have it ready to go for us (which actually could be a bad thing, depending on the use case!). The “proxy” is what basically tells Sencha how to get us our data. The “url” is probably the most obvious part, but we need to also understand the “type” and the “callbackKey”.

This whole blog session is intended to be a 10,000’ overview of these technologies. But one area I consistently see people get hung up on is JSONP and the “same origin policy” concept as it pertains to JavaScript. As such, I thought it was worth the time to introduce the concepts so that hopefully you can breeze through it when (yes, when) you hit this issue in your mobile travels.

A Google search will get you plenty deep into these concepts, but the part we need to understand for the purposes of this document is that most web browsers do not allow JavaScript to make “cross-domain” calls to servers other than the one the JavaScript originated from. This is a security feature. In other words, “xyz.cloudfoundry.com” is not technically allowed to make requests from “abc.cloudfoundry.com” (note: even when the domain name itself is the same, the difference in subdomain would still cause the calls to fail). So in our “proxy” config above, the “callbackKey” value is sent to the server (in this case our Grails app) and padded (the “p” in jsonp) to the JSON response, hence verifying the legitimacy of the caller.

Once again, this is a brief overview but an important enough concept that I wanted to add it to the discussion because it WILL bite you in any data-driven mobile application.

In fact, I thought it was so important that I am now going to make you go back to your Grails application and add the padding to your ChallengeController.

In the upper right-hand corner, change your perspective back to “Grails”. Go back into your original Grails project and navigate to the “controllers” and find your “ChallengeController”. Double-click to open the editor and find your “mobilefeed()” method. Replace the “render” portion with the following code:

```
render"${params.myMobileKey}(${Challenge.list()} as JSON})"
```

Save ChallengeController.

It may look a bit daunting, but look at what is in there: It is getting the “myMobileKey” out of the parameters, which is what we referenced as the “callbackKey” in our “app.js”. And you can also see our Challenge.list() as JSON just as it was before. The rest is just syntax.

The point of this exercise was to introduce you to the concept of padding your JSON response so that your JavaScript code can access your data when it’s on another server. Although this document and session are very high-level, the concepts are critical and will hopefully help you in the future.

**UPDATE AND RESTART** the GRAILS application (i.e. repeat the steps from above for restarting your application, but make sure you select the Grails application and not your mobile application). While you are waiting for the restart, go ahead and collapse your workspace and go back to the “Spring” perspective (upper right-hand corner of the workspace)
Finally, we need to rewire our Ext.List that we hardcoded last time to reference our new store! Navigate back to your “app.js” file and add a {lastName} to the first name element you already defined so that we will actually be displaying both the first and last names on our list. Now, replace the entire “data” config with a new config parameter called “store” and define it as “challengeStore”. Your new, incredibly simplified code for your list should look like this:

```javascript
// create our list
Ext.create('Ext.List',{
    fullscreen: true,
    itemTpl: '{firstName} {lastName}',
    store: challengeStore
});
```

So something may really stand out to you. It makes sense that we eliminated the “data” config since we are not hardcoding the data anymore. And it makes sense that we would use a “store”, since we just spent a page-and-a-half defining what a store is. 😊 But what is this “challengeStore” business?

We are referencing a variable we have not created yet. Here is all we have to do: simply add “`var challengeStore =`” to the beginning of the line where we create the store.

That line of code should now look like the following:

```javascript
var challengeStore = Ext.create('Ext.data.Store',{
```

This now gives us the ability to reference that store as a variable in our list! Your entire “app.js” file should now look like the following:
Ext.application({
    name: 'MyMobileApp',
    launch: function(){
        // define our model
        Ext.define('Challenge',{ 
            extend: 'Ext.data.Model',
            config:{
                fields:[
                    {name:'firstName',type:'string'},
                    {name:'lastName',type:'string'},
                    {name:'bigChallenge',type:'string'}
                ]
            }
        });

        // create our store
        var challengeStore = Ext.create('Ext.data.Store',{ 
            model: 'Challenge',
            proxy: {
                type: 'jsonp',
                url: 'http://<WEB_APP>.cloudfoundry.com/challenge/mobilefeed',
                callbackKey: 'myMobileKey',
                autoLoad: true
            }
        });

        // create our list
        Ext.create('Ext.List',{ 
            fullscreen: true,
            itemTpl: '{firstName} {lastName}',
            store: challengeStore
        });
    }
});

Save "app.js".

**UPDATE AND RESTART** the mobile application.

Go back to your mobile device and refresh your browser. Your application should now be displaying the data you entered in your Grails application. Pretty cool, eh?
15. Now we want to add some grouping functionality so that we can make our app look really slick if we start getting massive loads of data getting entered into our application.

First we are going to add a config property called “sorters” to our store. We will tell it to sort on the lastName property and put it after the autoLoad config:

```javascript
sorters: 'lastName'
```

Now we need to add a “grouper” config. This config is a function that handles the grouping functionality for us. We will put that right after the “sorters” config:

```javascript
grouper: {
    groupFn: function(record){
        return record.get('lastName')[0];
    }
}
```

This is another one of those syntax things that you shouldn’t worry too much about right now. We are definitely getting into the parts of the application where we are adding some features that will be more normal the more you work with the frameworks.

The last thing we need to do is add a grouped:true config to our Ext.create() block. Go ahead and put the code right after the store config. Your complete “app.js” should now look like this:
Ext.application({
  name: 'MyMobileApp',
  launch: function(){
    // define our model
    Ext.define('Challenge',{ 
      extend: 'Ext.data.Model',
      config:{
        fields:[
          {name:'firstName',type:'string'},
          {name:'lastName',type:'string'},
          {name:'bigChallenge',type:'string'}
        ]
      }
    });

    // create our store
    var challengeStore = Ext.create('Ext.data.Store',{ 
      model: 'Challenge',
      proxy: {
        type:'jsonp',
        url:'http://<WEB_APP>.cloudfoundry.com/challenge/mobilefeed',
        callbackKey: 'myMobileKey'
      },
      autoLoad: true,
      sorters: 'lastName',
      grouper: {
        groupFn: function(record){
          return record.get('lastName')[0];
        }
      }
    });

    // create our list
    Ext.create('Ext.List',{ 
      fullscreen: true,
      itemTpl: '{firstName} {lastName}',
      store: challengeStore,
      grouped: true
    });
  }
});

Save “app.js”.

UPDATE AND RESTART the mobile application.

Go back to your mobile device and refresh your browser. Your application should now have your names grouped by last name.
16. We’re almost in the home stretch. We want to add one piece of code to our application that will actually display the users biggest challenge when we touch their name on the screen. Type this code after the list creation:

```javascript
//adding event to List (touch)
challengeList.on("itemtap", function(dataView,index,target,record,e){
    Ext.Msg.alert(record.data.bigChallenge);
});
```

It’s a compressed piece of code, but if you look closely you can see that it is taking the “challengeList” object and adding an event listener looking for an “itemtap”. When the event occurs it will fire off a function that essentially calls “Ext.Msg.alert()” which ultimately will create a popup for us displaying that person’s biggest challenge.

But wait? What is this “challengeList” thing? 😊 Much like we had to add a variable to our Ext.data.Store, we now have to add a variable to our list. Add `var challengeList =` to the beginning of your List declaration.

Your completed code should look as follows:
Ext.application({
    name: 'MyMobileApp',
    launch: function(){
        // define our model
        Ext.define('Challenge', {
            extend: 'Ext.data.Model',
            config: {
                fields: [
                    {name: 'firstName', type: 'string'},
                    {name: 'lastName', type: 'string'},
                    {name: 'bigChallenge', type: 'string'}
                ]
            }
        });

        // create our store
        var challengeStore = Ext.create('Ext.data.Store', {
            model: 'Challenge',
            proxy: {
                type: 'jsonp',
                url: 'http://<WEB_APP>..cloudfoundry.com/challenge/mobilefeed',
                callbackKey: 'myMobileKey'
            },
            autoLoad: true,
            sorters: 'lastName',
            grouper: {
                groupFn: function(record){
                    return record.get('lastName')[0];
                }
            }
        });

        // create our list
        var challengeList = Ext.create('Ext.List', {
            fullscreen: true,
            itemTpl: '{firstName} {lastName}',
            store: challengeStore,
            grouped: true
        });

        // adding event to List (touch)
        challengeList.on('itemtap', function(dataView, index, target, record, e){
            Ext.Msg.alert(record.data.bigChallenge);
        });
    }
});
Save “app.js”.

**UPDATE AND RESTART** the mobile application.

Go back to your mobile device and refresh your browser. Your application should now popup the user’s biggest challenge when you touch their name!

17. This last piece is literally the easiest part of the entire exercise, and also the most rewarding. This is where we will install the application on our phones so that it looks exactly like a real mobile application. You have probably noticed by now that when you run the app from your phone it still looks like its running in a browser. I will do this with the iPhone to illustrate.

Find an image that is a perfect square. Name it “icon.png” and drop it in the same “WebContent” folder that you have “app.js”. You only have one line of code to add to your application:

```javascript
icon: 'icon.png',
```

Add it to the top of your application, right after the “name” config. Your final code should look like this:
Ext.application({
  name: 'MyMobileApp',
  icon: 'icon.png',
  launch: function(){
    // define our model
    Ext.define('Challenge', {
      extend: 'Ext.data.Model',
      config: {
        fields: [
          {name: 'firstName', type: 'string'},
          {name: 'lastName', type: 'string'},
          {name: 'bigChallenge', type: 'string'}
        ]
      }
    });

    // create our store
    var challengeStore = Ext.create('Ext.data.Store', {
      model: 'Challenge',
      proxy: {
        type: 'jsonp',
        url: 'http://<WEB_APP>.cloudfoundry.com/challenge/mobilefeed',
        callbackKey: 'myMobileKey'
      },
      autoLoad: true,
      sorters: 'lastName',
      grouper: {
        groupFn: function(record){
          return record.get('lastName')[0];
        }
      }
    });

    // create our list
    var challengeList = Ext.create('Ext.List', {
      fullscreen: true,
      itemTpl: '{firstName} {lastName}',
      store: challengeStore,
      grouped: true
    });

    // adding event to List (touch)
    challengeList.on('itemtap', function(dataView, index, target, record, e){
      Ext.Msg.alert(record.data.bigChallenge);
    });
  }
});
UPDATE AND RESTART the mobile application.

Go back to your mobile device and refresh your browser. The application should look and work exactly like it did last time. But now we will “install” in on your iPhone.

Press the middle button on the bottom of the browser and select “Add to Home Screen”. This will present you with a screen that will feature the “icon.png” file that we dropped in, and also feature the name of the application that maps to the <title> tag that we defined way back when. Simply tap the “Add” button in the upper right-hand corner and it will add the icon to your home screen,

But at this point when you launch the application, pay special attention to the fact that your application is now running in a chromeless browser and gives the appearance of a true native app.
Part Five – Wrap-up

So there you have it! Your very first mobile application AND a complete working web application built in Grails. As mentioned many times, this was simply meant as a way to get your hands dirty and have something tangible to show and have fun with. I mentioned “PhoneGap” during the blog as a way to package these web resources so that they function on your phone as a true native executing application instead of loaded web resources from an external server. Plus, you get access to native device features like the camera and you can publish to the various app stores! But there is literally a whole world of thoughts and considerations to be had when thinking about a mobile solution. I hope this document serves as a decent launching pad, and good luck in your mobile journey!