# PhoneGap Workshop

## Part 1: Exploring the App and Choosing a Local Storage Option

#### Step 1: Explore different persistence mechansisms

Open the following files in **js/storage**, and explore the different persistence stores:

1. memory-store.js (MemoryStore)
2. ls-store.js (LocalStorageStore)
3. websql-store.js (WebSqlStore)

#### Step 2: Test the application with different persistence stores

To change the local persistence store for the application:

1. In **js/main.js**: Instantiate the specific store in the initialize() function of the app object: **MemoryStore**, **LocalStorageStore**, or **WebSqlStore**.
2. In **index.html**: add a script tag for the corresponding .js file: **memory-store.js**, **ls-store.js**, or **websql-store.js**.

## Part 2: Building with PhoneGap Build

1. If you don’t already have one, create an account on http://build.phonegap.com.
2. Create a new application in PhoneGap Build. Either point to a GitHub repository, or zip up your phonegap-workshop directory and upload it to PhoneGap Build.
3. Click the **Ready to build** button
4. When the build process completes, use a QR Code reader app to install the Employee Directory application on your device.

To fine tune your build preferences:

1. In the phonegap-workshop directory, add a **config.xml** file defined as follows (make the necessary adjustments for id, author, etc.):

* ```xml
* <name>Employee Directory</name>  
    
  <description>  
   A simple employee directory application  
  </description>  
    
  <author href="http://coenraets.org" email="ccoenraets@gmail.com">  
   Christophe Coenraets  
  </author>
* ```

1. If you used the GitHub approach, push your changes to your repository and click the **Update Code** button in PhoneGap Build. If you used the zip file approach, zip up your phonegap-workshop directory again and upload the new version to PhoneGap Build

## Part 3: Using Native Notification

1. In index.html, add the following script tag:

* <script src="phonegap.js"></script>
* NOTE: This tells PhoneGap Build to inject a platform specific phonegap.js at build time. In other words, phonegap.js doesn’t need to be (and shouldn’t be) present in your project folder.

1. In main.js, define a function named showAlert() inside the app object. If *navigator.notification* is available, use its alert() function. Otherwise, use the default browser alert() function.

* showAlert: function (message, title) {  
   if (navigator.notification) {  
   navigator.notification.alert(message, null, title, 'OK');  
   } else {  
   alert(title ? (title + ": " + message) : message);  
   }  
  }

1. Test the notification logic by displaying a message when the application store has been initialized: Pass an anonymous callback function as an argument to the constructor of the persistence store (the store will call this function after it has successfully initialized). In the anonymous function, invoke the showAlert() function. When you run the application in the browser, you see a standard browser alert. When you run the application on your device, you see a native alert.

* initialize: function() {  
   var self = this;  
   this.store = new MemoryStore(function() {  
   self.showAlert('Store Initialized', 'Info');  
   });  
   $('.search-key').on('keyup', $.proxy(this.findByName, this));  
  }

## Part 4: Setting Up a Single Page Application

1. In index.html: remove the HTML markup inside the body tag.
2. In main.js, define a function named renderHomeView() inside the app object. Implement the function to programmatically add the Home View markup to the body element.

* renderHomeView: function() {  
   var html =  
   "<div class='header'><h1>Home</h1></div>" +  
   "<div class='search-view'>" +  
   "<input class='search-key'/>" +  
   "<ul class='employee-list'></ul>" +  
   "</div>"  
   $('body').html(html);  
   $('.search-key').on('keyup', $.proxy(this.findByName, this));  
  }

1. Modify the initialize() function of the app object. In the anonymous callback function of the store constructor, call the renderHomeView() function to programmatically display the Home View.

* initialize: function() {  
   var self = this;  
   this.store = new MemoryStore(function() {  
   self.renderHomeView();  
   });  
  }

## Part 5: Using Handlebar Templates

Modify index.html as follows:

1. Add a script tag to include the handlebar.js library:

* <script src="lib/handlebars.js"></script>

1. Create an HTML template to render the Home View:

* <script id="home-tpl" type="text/x-handlebars-template">  
   <div class='header'><h1>Home</h1></div>  
   <div class='search-bar'><input class='search-key' type="search"/></div>  
   <ul class='employee-list'></ul>  
  </script>

1. Create an HTML template to render the employee list items:

* <script id="employee-li-tpl" type="text/x-handlebars-template">  
   {{#.}}  
   <li><a href="#employees/{{this.id}}">{{this.firstName}} {{this.lastName}}<br/>{{this.title}}</a></li>  
   {{/.}}  
  </script>

Modify main.js as follows:

1. In the initialize() function of the app object, add the code to compile the two templates defined above:

* this.homeTpl = Handlebars.compile($("#home-tpl").html());  
  this.employeeLiTpl = Handlebars.compile($("#employee-li-tpl").html());

1. Modify renderHomeView() to use the homeTpl template instead of the inline HTML:

* renderHomeView: function() {  
   $('body').html(this.homeTpl());  
   $('.search-key').on('keyup', $.proxy(this.findByName, this));  
  }

1. Modify findByName() to use the employeeLiTpl template instead of the inline HTML:

* findByName: function() {  
   var self = this;  
   this.store.findByName($('.search-key').val(), function(employees) {  
   $('.employee-list').html(self.employeeLiTpl(employees));  
   });  
  }

1. Test the application.

## Part 6: Creating a View Class

#### Step 1: Create the HomeView Class

1. Create a file called HomeView.js in the js directory, and define a HomeView class as follows:

* var HomeView = function(store) {  
    
  }

1. Add the two templates as *static* members of HomeView.

* var HomeView = function(store) {  
    
  }  
    
  HomeView.template = Handlebars.compile($("#home-tpl").html());  
  HomeView.liTemplate = Handlebars.compile($("#employee-li-tpl").html());

1. Define an initialize() function inside the HomeView class. Define a div wrapper for the view. The div wrapper is used to attach the view-related events. Invoke the initialize() function inside the HomeView constructor function.

* var HomeView = function(store) {  
    
   this.initialize = function() {  
   // Define a div wrapper for the view. The div wrapper is used to attach events.  
   this.el = $('<div/>');  
   this.el.on('keyup', '.search-key', this.findByName);  
   };  
    
   this.initialize();  
    
  }  
    
  HomeView.template = Handlebars.compile($("#home-tpl").html());  
  HomeView.liTemplate = Handlebars.compile($("#employee-li-tpl").html());

1. Move the renderHomeView() function from the app object to the HomeView class. To keep the view reusable, attach the html to the div wrapper (this.el) instead of the document body. Because the function is now encapsulated in the HomeView class, you can also rename it render(), instead of renderHomeView().

* this.render = function() {  
   this.el.html(HomeView.template());  
   return this;  
  };

1. Move the findByName() function from the app object to the HomeView class.

* this.findByName = function() {  
   store.findByName($('.search-key').val(), function(employees) {  
   $('.employee-list').html(HomeView.liTemplate(employees));  
   });  
  };

#### Step 2: Using the HomeView class

1. In index.html, add a script tag to include HomeView.js:

* <script src="js/HomeView.js"></script>

1. Remove the renderHomeView() function from the app object.
2. Remove the findByName() function from the app object.
3. Modify the initialize function() to display the Home View using the HomeView class:

* initialize: function() {  
   var self = this;  
   this.store = new MemoryStore(function() {  
   $('body').html(new HomeView(self.store).render().el);  
   });  
  }

## Part 7: Adding Styles and Touch-Based Scrolling

#### Step 1: Style the Application

1. Add the Source Sans Pro font definition to the head of index.html

* <script src="css/source-sans-pro.js"></script>

1. Add styles.css to the head of index.html

* <link href="css/styles.css" rel="stylesheet">

1. In index.html, modify the home-tpl template: change the search-key input type from text to **search**.
2. Test the application. Specifically, test the list behavior when the list is bigger than the browser window.

#### Step 2: Native Scrolling Approach

1. Modify the home-tpl template in index.html. Add a div wrapper with a *scroll* class around the ul element:

* <script id="home-tpl" type="text/x-handlebars-template">  
   <div class='header'><h1>Home</h1></div>  
   <div class='search-bar'><input class='search-key' type="search"/></div>  
   <div class="scroll"><ul class='employee-list'></ul></div>  
  </script>

1. Add the following class definition to css/styles.css:

* .scroll {  
   overflow: auto;  
   -webkit-overflow-scrolling: touch;  
   position: absolute;  
   top: 84px;  
   bottom: 0px;  
   left: 0px;  
   right: 0px;  
  }

#### Step 3: iScroll Approach

1. In index.html, add a script tag to include the iscroll.js library:

* <script src="lib/iscroll.js"></script>

1. In HomeView.js, modify the findByName() function: Instantiate an iScroll object to scroll the list of employees. If the iScroll object already exists (), simply refresh it to adapt it to the size of the new list.

* this.findByName = function() {  
   store.findByName($('.search-key').val(), function(employees) {  
   $('.employee-list').html(HomeView.liTemplate(employees));  
   if (self.iscroll) {  
   console.log('Refresh iScroll');  
   self.iscroll.refresh();  
   } else {  
   console.log('New iScroll');  
   self.iscroll = new iScroll($('.scroll', self.el)[0], {hScrollbar: false, vScrollbar: false });  
   }  
   });  
  };

## Part 8: Highlighting Tapped or Clicked UI Elements

1. In styles.css, add a *tappable-active* class definition for *tapped* or *clicked* list items. The class simply highlights the item with a blue background:

* li>a.tappable-active {  
   color: #fff;  
   background-color: #4286f5;  
  }

1. In main.js, define a registerEvents() function inside the app object. Add a the *tappable\_active* class to the selected (*tapped* or *clicked*) list item:

* registerEvents: function() {  
   var self = this;  
   // Check of browser supports touch events...  
   if (document.documentElement.hasOwnProperty('ontouchstart')) {  
   // ... if yes: register touch event listener to change the "selected" state of the item  
   $('body').on('touchstart', 'a', function(event) {  
   $(event.target).addClass('tappable-active');  
   });  
   $('body').on('touchend', 'a', function(event) {  
   $(event.target).removeClass('tappable-active');  
   });  
   } else {  
   // ... if not: register mouse events instead  
   $('body').on('mousedown', 'a', function(event) {  
   $(event.target).addClass('tappable-active');  
   });  
   $('body').on('mouseup', 'a', function(event) {  
   $(event.target).removeClass('tappable-active');  
   });  
   }  
  }

1. Invoke the registerEvents() function from within the app object’s initialize() function.
2. Test the application.

## Part 9: Routing to Multiple Views

#### Step 1: Create the employee template

Open index.html and add a template to render a detailed employee view:

<script id="employee-tpl" type="text/x-handlebars-template">  
 <div class='header'><a href='#' class="button header-button header-button-left">Back</a><h1>Details</h1></div>  
 <div class='details'>  
 <img id='image' src='img/{{firstName}}\_{{lastName}}.jpg' style="float:left;margin:10px;"/>  
 <h1>{{firstName}} {{lastName}}</h1>  
 <h2>{{title}}</h2>  
 <ul>  
 <li><a href="tel:{{officePhone}}">Call Office<br/>{{officePhone}}</a></li>  
 <li><a href="tel:{{cellPhone}}">Call Cell<br/>{{cellPhone}}</a></li>  
 <li><a href="sms:{{cellPhone}}">SMS<br/>{{cellPhone}}</a></li>  
 </ul>  
 </div>  
</script>

#### Step 2: Create the EmployeeView class

1. Create a file called EmployeeView.js in the js directory, and define an EmployeeView class as follows:

* var EmployeeView = function() {  
    
  }

1. Add the template as a *static* member of EmployeeView.

* var EmployeeView = function() {  
    
  }  
    
  EmployeeView.template = Handlebars.compile($("#employee-tpl").html());

1. Define an initialize() function inside the HomeView class. Define a div wrapper for the view. The div wrapper is used to attach the view-related events. Invoke the initialize() function inside the HomeView constructor function.

* var EmployeeView = function(employee) {  
    
   this.initialize = function() {  
   this.el = $('<div/>')  
   };  
    
   this.initialize();  
    
   }  
    
  EmployeeView.template = Handlebars.compile($("#employee-tpl").html());

1. Define a render() function implemented as follows:

* this.render = function() {  
   this.el.html(EmployeeView.template(employee));  
   return this;  
  };

#### Step 3: Implement View Routing

1. In the app’s initialize() function, define a regular expression that matches employee details urls.

* this.detailsURL = /^#employees\/(\d{1,})/;

1. In the app’s registerEvents() function, add an event listener to listen to URL hash tag changes:

* $(window).on('hashchange', $.proxy(this.route, this));

1. In the app object, define a route() function to route requests to the appropriate view:
   * If there is no hash tag in the URL: display the HomeView
   * If there is a has tag matching the pattern for an employee details URL: display an EmployeeView for the specified employee.

* route: function() {  
   var hash = window.location.hash;  
   if (!hash) {  
   $('body').html(new HomeView(this.store).render().el);  
   return;  
   }  
   var match = hash.match(app.detailsURL);  
   if (match) {  
   this.store.findById(Number(match[1]), function(employee) {  
   $('body').html(new EmployeeView(employee).render().el);  
   });  
   }  
  }
* Modify the initialize() function to call the route() function:
* initialize: function() {  
   var self = this;  
   this.detailsURL = /^#employees\/(\d{1,})/;  
   this.registerEvents();  
   this.store = new MemoryStore(function() {  
   self.route();  
   });  
  }
* Test the application.

## Part 10: Using the Location API

1. In index.html, add the following list item to the employee-tpl template:

* <li><a href="#" class="add-location-btn">Add Location</a></li>

1. In the initialize() function of EmployeeView, register an event listener for the click event of the *Add Location* list item:

* this.el.on('click', '.add-location-btn', this.addLocation);

1. In EmployeeView, define the *addLocation* event handler as follows:

* this.addLocation = function(event) {  
   event.preventDefault();  
   console.log('addLocation');  
   navigator.geolocation.getCurrentPosition(  
   function(position) {  
   $('.location', this.el).html(position.coords.latitude + ',' +position.coords.longitude);  
   },  
   function() {  
   alert('Error getting location');  
   });  
   return false;  
  };

1. Test the Application

## Part 11: Using the Contacts API

1. In index.html, add the following list item to the employee template:

* <li><a href="#" class="add-contact-btn">Add to Contacts</a></li>

1. In the initialize() function of EmployeeView, register an event listener for the click event of the *Add to Contacts* list item:

* this.el.on('click', '.add-contact-btn', this.addToContacts);

1. In EmployeeView, define the *addToContacts* event handler as follows:

* this.addToContacts = function(event) {  
   event.preventDefault();  
   console.log('addToContacts');  
   if (!navigator.contacts) {  
   app.showAlert("Contacts API not supported", "Error");  
   return;  
   }  
   var contact = navigator.contacts.create();  
   contact.name = {givenName: app.currentEmployee.firstName, familyName: app.currentEmployee.lastName};  
   var phoneNumbers = [];  
   phoneNumbers[0] = new ContactField('work', app.currentEmployee.officePhone, false);  
   phoneNumbers[1] = new ContactField('mobile', app.currentEmployee.cellPhone, true); // preferred number  
   contact.phoneNumbers = phoneNumbers;  
   contact.save();  
   return false;  
  };

1. Test the Application

## Part 12: Using the Camera API

1. In index.html, add the following list item to the employee template:

* <li><a href="#" class="change-pic-btn">Change Picture</a></li>

1. In the initialize() function of EmployeeView, register an event listener for the click event of the Change Picture list item:

* this.el.on('click', '.change-pic-btn', this.changePicture);

1. In EmployeeView, define the changePicture event handler as follows:

* this.changePicture = function(event) {  
   event.preventDefault();  
   console.log('changePicture');  
   if (!navigator.camera) {  
   app.showAlert("Camera API not supported", "Error");  
   return;  
   }  
   var options = { quality: 50,  
   destinationType: Camera.DestinationType.DATA\_URL,  
   sourceType: 1, // 0:Photo Library, 1=Camera, 2=Saved Photo Album  
   encodingType: 0 // 0=JPG 1=PNG  
   };  
    
   navigator.camera.getPicture(  
   function(imageData) {  
   $('#image').attr('src', "data:image/jpeg;base64," + imageData);  
   },  
   function() {  
   alert('Error taking picture');  
   },  
   options);  
    
   return false;  
  };

1. Test the Application

## Part 13: Sliding Pages with CSS Transitions

1. Add the following classes to styles.css:

* .page {  
   position: absolute;  
   width: 100%;  
   height: 100%;  
   -webkit-transform:translate3d(0,0,0);  
  }  
    
  .stage-center {  
   top: 0;  
   left: 0;  
  }  
    
  .stage-left {  
   left: -100%;  
  }  
    
  .stage-right {  
   left: 100%;  
  }  
    
  .transition {  
   -moz-transition-duration: .375s;  
   -webkit-transition-duration: .375s;  
   -o-transition-duration: .375s;  
  }

1. Inside the app object, define a slidePage() function implemented as follows:

* slidePage: function(page) {  
    
   var currentPageDest,  
   self = this;  
    
   // If there is no current page (app just started) -> No transition: Position new page in the view port  
   if (!this.currentPage) {  
   $(page.el).attr('class', 'page stage-center');  
   $('body').append(page.el);  
   this.currentPage = page;  
   return;  
   }  
    
   // Cleaning up: remove old pages that were moved out of the viewport  
   $('.stage-right, .stage-left').not('.homePage').remove();  
    
   if (page === app.homePage) {  
   // Always apply a Back transition (slide from left) when we go back to the search page  
   $(page.el).attr('class', 'page stage-left');  
   currentPageDest = "stage-right";  
   } else {  
   // Forward transition (slide from right)  
   $(page.el).attr('class', 'page stage-right');  
   currentPageDest = "stage-left";  
   }  
    
   $('body').append(page.el);  
    
   // Wait until the new page has been added to the DOM...  
   setTimeout(function() {  
   // Slide out the current page: If new page slides from the right -> slide current page to the left, and vice versa  
   $(self.currentPage.el).attr('class', 'page transition ' + currentPageDest);  
   // Slide in the new page  
   $(page.el).attr('class', 'page stage-center transition');  
   self.currentPage = page;  
   });  
    
  }

1. Modify the route() function as follows:

* route: function() {  
   var self = this;  
   var hash = window.location.hash;  
   if (!hash) {  
   if (this.homePage) {  
   this.slidePage(this.homePage);  
   } else {  
   this.homePage = new HomeView(this.store).render();  
   this.slidePage(this.homePage);  
   }  
   return;  
   }  
   var match = hash.match(this.detailsURL);  
   if (match) {  
   this.store.findById(Number(match[1]), function(employee) {  
   self.slidePage(new EmployeeView(employee).render());  
   });  
   }  
  }

## More topics

* Conditional css
* Touch vs click
* Viewport
* Testing
* External template loader
* Plugins