Intro to Sencha Architect: Creating Mobile Apps with Sencha Touch

Unit Objectives

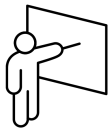
After completing this unit, you should be able to:

- Define Models and Stores in Sencha Architect
- Define a Viewport
- Define your application UI using Drag & Drop
- Define interactions between user interface elements

Unit Topics

- About this Tutorial
- Introducing Sencha Architect
- Working with Data
- Defining Views
- Visualizing Data with List Views
- Handling Interactions with Controllers
- Implementing GEO Features
- Theming Your Application
- Packaging a Native App

About this Tutorial



This tutorial was developed for the following audiences:

- Developers who are already familiar with coding apps in Sencha Touch but have never used Sencha Architect and would like to go through a quick, self-paced introduction.
- Developers who are unfamiliar with Sencha technologies and who wish to learn more in an instructor-led environment.

During this tutorial you will create a small app that reads data from the Yelp API and outputs the information to a variety of mobile devices. Completing this tutorial should take approximately 2.5 hours.

This tutorial was validated under Sencha Touch 2.3 and Sencha Architect 3.1.0 build 1851.

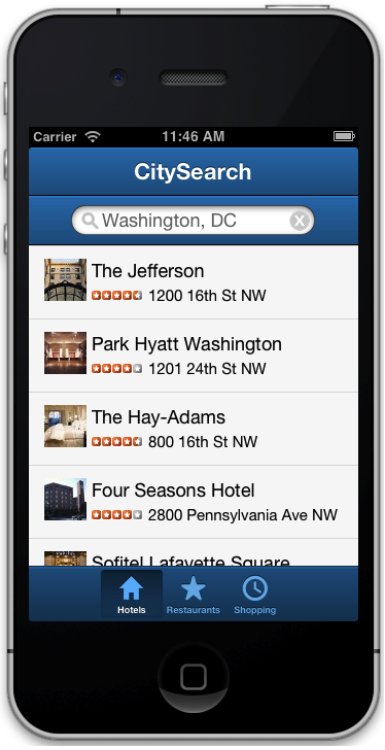


Figure 1: Build this app during this quick tutorial

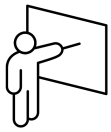
System Requirements

You will need to install the following software to successfully complete this tutorial:

1. Sencha Architect (http://www.sencha.com/products/architect/download/)
2. Java Runtime Environment (JRE) 1.7 http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html
3. Ruby http://www.rubyinstaller.org
4. A Yelp user account api key, which you can procure from http://www.yelp.com/developers/
5. The course files, which can be downloaded from https://github.com/sdruckerfig/IntroToSenchaArchitect

Unzip the course files to your desktop, open the senchaarchitectbyol/helpers.txt file and insert your Yelp ID where directed.

Introducing Sencha Architect



Sencha Architect is a desktop application that enables you to prototype Sencha Touch applications using a drag and drop interface. With Sencha Architect you can create the following types of classes:

- Models
- Stores
- Viewport
- Views
- Forms
- Basic Controllers

In addition, Sencha Architect automates the build and deploy processes of Sencha Cmd.

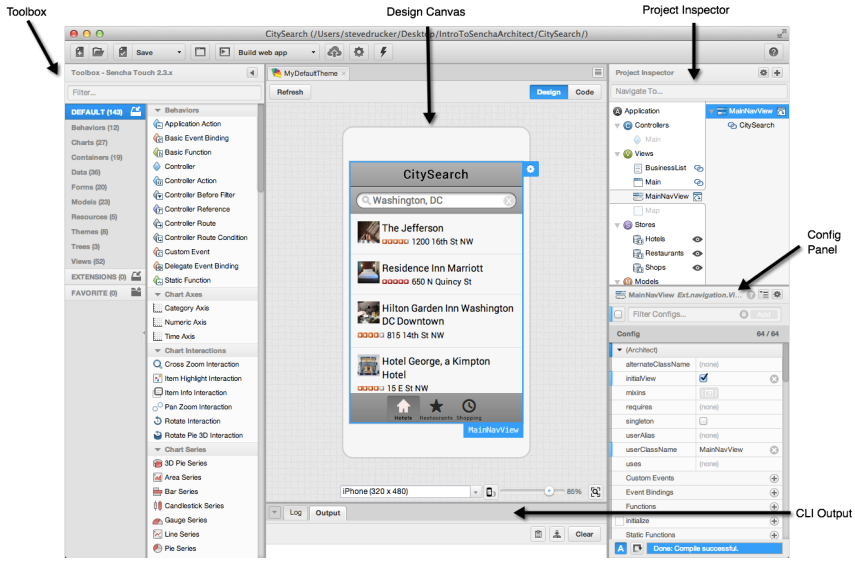
Exceptionally clean code is generated by Sencha Architect using the Sencha Model-View-Controller framework and closely follows other best practices for app development.

Note that you must have the Sencha Cmd command line interface (CLI) installed for all of the deployment and testing functions to operate properly. Sencha Architect will automatically install Sencha Cmd. Alternately, you can download Sencha Cmd from the following URL:

http://www.sencha.com/products/sencha-cmd/

Reviewing the Sencha Architect Graphical Interface

Sencha Architect uses a GUI that is similar to other integrated development environments:



About the Model-View-Controller Design Pattern

Large client side applications have always been difficult to maintain. They quickly grow out of control as you add more functionality and developers to a project. Sencha Touch 2 comes with an application architecture that not only organizes your code but reduces the amount of code that you have to write and maintain.

The application architecture follows an MVC-like pattern. While there are many MVC architectures, most of which are slightly different from one another, Sencha Touch MVC is defined as follows:

- Models are a collection of data fields which persist through the data package. They can be linked to other models through associations and linked to a data stream through the use of a proxy.
- Stores are a client-side cache of model instances.
- Views are any type of components that output information to your browser, i.e. Container, Map, TabPanels, FormPanel
- Controllers are used to add your application logic. You can add events to your views and their components as well as integrate data from your stores into your views.

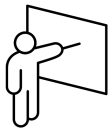
Understanding the Benefits of Sencha MVC

Using MVC in Sencha Touch 2 provides you with the following benefits:

- Consistent file structure for your code and classes
- Controllers associate models and stores with views
- Easy to add your app's logic to your controllers
- Easy to add events to any of your controller's view components using simple CSS selector syntax
- Create reusable components (views) that may be defined visually through Sencha Architect and not bounded to app logic
- Code is easy to read and maintain
- Dependencies between classes are easier to manage
- No need to explicitly manage required classes

MVC isn't required to develop applications, but when it is used, it brings benefits in code reuse and a predictable coding structure and pattern. Sencha Architect always generates code using the MVC pattern.

Working with Data



The Sencha Touch data package is comprised of the following three major classes:

- Models define the fields that comprise each record.
- Proxies define the connection between the browser and the server.
- Stores define client-side caches of multiple model instances and handle services such as filtering, sorting, and grouping.

Defining Models

Models primarily consist of a series of field names. You can optionally define data types for each field that will cause Sencha Touch to convert processed data to a specified type. Typically you would define data types only in cases where you are reading in dates or numeric values from a data feed.

```
Ext.define('FriendsWithBeer.model.Beer', {
 extend: 'Ext.data.Model',
 config: {
  idProperty: 'id',
  fields: [
   'id','name','type','country'
  ]
 }
});
```

Type Casting Data

Since Javascript is a loosely typed language, Sencha Touch enables you to cast the values read from a server into one of the following types:

• int	• boolean	• date
• string	• float	• auto (default)

A type of 'auto' specifies that there will not be any type conversion.

Specifying a type on a field is required if your app will be performing any client-side sorting on that field. Also, you should always specify a type on date fields so that you may Sencha Touch's powerful Ext.Date methods to transform and format native javascript date objects.

The following example illustrates defining a model with type casting:

```
Ext.define('MyApp.model.Person', {
 extend: 'Ext.data.Model',
 config: {
  fields: [
   { name: 'id', type: 'int'},
   'fName', // string, so no conversion required
   'lName',
   { name: 'dob', type: 'date', dateFormat: 'm/d/Y'},
  ]
 }
});
```

Defining a Proxy

Models provide a one to one mapping between our client and remote data sources. Once the model is defined, loading the models with data is handled through a proxy. Sencha Touch and Ext JS 4 include the following proxies:

- Local Storage - The Local Storage object is an HTML5 feature for storing data locally in the browser. Unlike cookies the local storage object allows storage of up to 5 MB of data, and many mobile browsers now allow larger amounts of storage to be allocated.
- Local Database (SQL) - Sencha introduced a WebSQL proxy in Sencha Touch 2.1 that enables you to easily create client-side relational databases.
- Ajax - Ajax proxies are limited to request within the same domain.
- JSON-P - JSON-P uses the script tag to access data that lives in another domain.
- REST - The REST proxy supports the RESTful URL structure and HTTP methods.
- SOAP - The SOAP proxy enables you to access SOAP-based web services.

Each proxy can optionally define a reader and writer property.

- The reader property specifies the format of the data that will be received from the server.
- The writer determines the format used to send data back to a remote service.

Proxies assume that the default format will be JSON (Java Script Object Notation), however, an XML reader and writer is also available.

Defining an AJAX Proxy

The following example illustrates how to define a proxy that reads data from an external url. Note how the field names map to the structure of the corresponding JSON dataset.

```
Ext.define('FriendsWithBeer.model.Beer', {
 extend: 'Ext.data.Model',
 requires: ['Ext.data.proxy.Ajax'],
 config: {
  idProperty: 'id',
  fields: [
   { name: 'id' },
   { name: 'name'},
   { name: 'type'},
   { name: 'country'}
  ],
  proxy: {
   type: 'ajax',
   url: 'data/beers.json',
   reader: {
     type: 'json'
   }
  }
 }
});
```

```
[
    {
        "id": 1,
        "name": "Budweiser",
        "type": "Lager",
        "country": "USA"
    }, {
        "id": 2,
        "name": "Iron City",
        "type": "Pilsner",
        "country": "USA"
    }
]
```

Defining a LocalStorage Proxy

LocalStorage proxies are defined using the following syntax. Note that you must specify a unique identifier that is used by Sencha Touch to create unique key names in the HTML5 LocalStorage repository.

```
Ext.define('FriendsWithBeer.model.Contact', {
 extend: 'Ext.data.Model',
 requires: [
  'Ext.data.proxy.LocalStorage',
  'Ext.data.identifier.Uuid'
 ],
 config: {
  identifier: 'uuid',
  fields: [
   'firstName','lastName'
  ],
  proxy: {
   type: 'localstorage',
   id: 'FriendsWithBeerContacts'
  }
 }
});
```

Defining Data Stores

Stores represent a local container of model instances. Most of your interaction with Models with happen through a Store.

Stores operate in a manner similar to arrays. The creation, deleting, and updating of models are handled by the Store's methods. A Store also provides methods for filtering, sorting, and grouping its records.

Each store in your application will have a corresponding model. As a best practice Store class names are the plural of its corresponding model, thereby indicating that it contains multiple model instances. Note that in the code snippet below, the class name for the store is the plural of its associated model name.

```
Ext.define( 'FriendsWithBeer.store.Users', {
 extend: 'Ext.data.Store',
 requires: ['FriendsWithBeer.model.User'],

 config: {
   autoLoad: true,
   model: 'FriendsWithBeer.model.User',
   storeId: 'Friends',
   autoSync: true
 }
});
```

Configuring a Store

You will typically configure the following Ext.data.Store properties:

Property	Description
autoLoad	Boolean. When set to true the store uses its proxy, or the proxy of its model if not defined, to populate on instantiation.
storeId	String. A unique identifier used to access this store globally using the Ext.getStore method. The storeId is also used to associate this store with views.
model	String/Object. A reference to the model class for this store.
autoSync	Boolean. When true, CRUD operations are enabled allowing the store to auto commit any models created, deleted, or updated. This will require additional configuration for Ajax proxies. JSON-P is not supported.

Invoking Store Methods

Commonly used methods for accessing data from a Store include the following:

Method	Description
add()	Add a model instance to a Store.
clearFilter()	Removes the filter and restores any filtered records.
filter()	Filters the records by a specified criteria.
getAt(Number)	Get the model record at the specified position in the store.
getCount()	Returns the number of records in the store. Filtered records are not included in the count.
sort(sorters)	Sorts the records in the Store by one or more properties.

Lab 1: Starting a New Project

In this lab, you will perform the following tasks:

- Launch Sencha Architect
- Start a new project
- Configure the Environment
- Define a data model and related stores

Steps

Configure Google Chrome for Debugging

1. Open Google Chrome
2. Open Google Chrome's **Settings** panel
3. Select Google Chrome as your default web browser.
4. Click on the **Customize and Control Google Chrome** button
5. Select **Tools** > **Developer Tools**
6. Click on the **Settings** button, located in the lower right corner of Chrome.
7. On the **General** tab, turn on the following checkboxes:
   - Disable cache
   - Log XMLHttpRequests
8. Click on the **Emulation** button and access the **Emulation** tab.

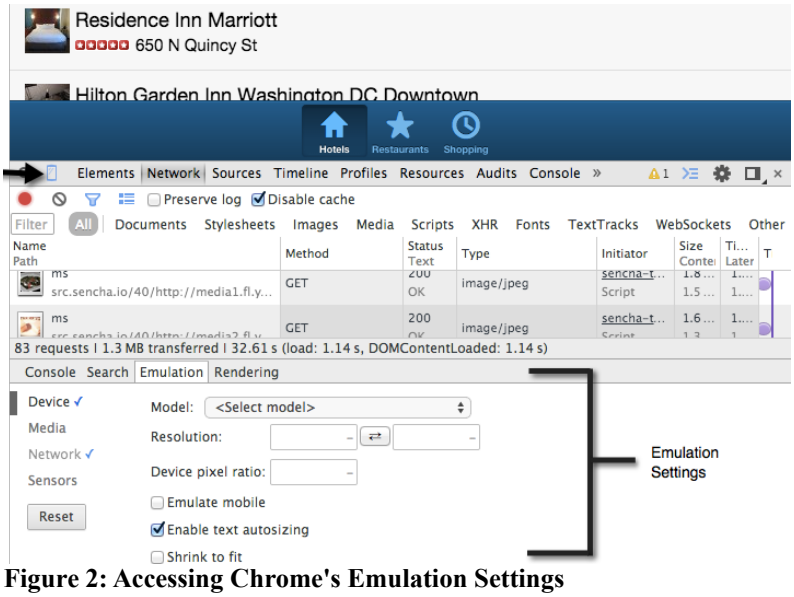


Figure 2: Accessing Chrome's Emulation Settings

9. Set the User Agent to **Apple iPhone 5**

10. Minimize the **Emulation** panel

11. Click on the Dock to Main Window button, located in the bottom left corner of the browser, in order to dock the developer tools to the right of the browsing area. Your browser should appear similar to the following:

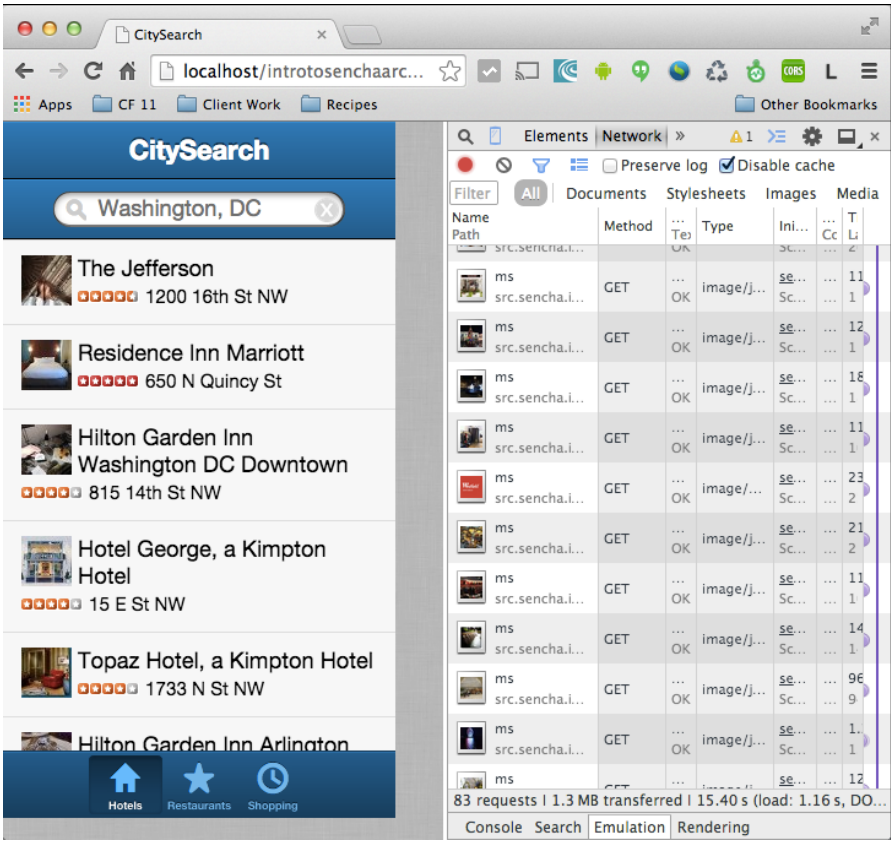


Figure 3: Google Chrome w/On-Screen Debugging and Device Emulation

Create a New Sencha Architect Project

12. Open Sencha Architect
13. On the splash screen, click the **Create New** button.
14. Click the radio button for Sencha Touch 2.3.x
15. Click on **Blank Project**
16. Click the **Create** button.
17. Click the **Save** button on the main toolbar. The Save Project dialog box appears.

18. Enter the following information:
    - Save Path: */path/to/Desktop/*IntroToSenchaArchitect
    - Project Name: **CitySearch**
    - App Name: **CitySearch**

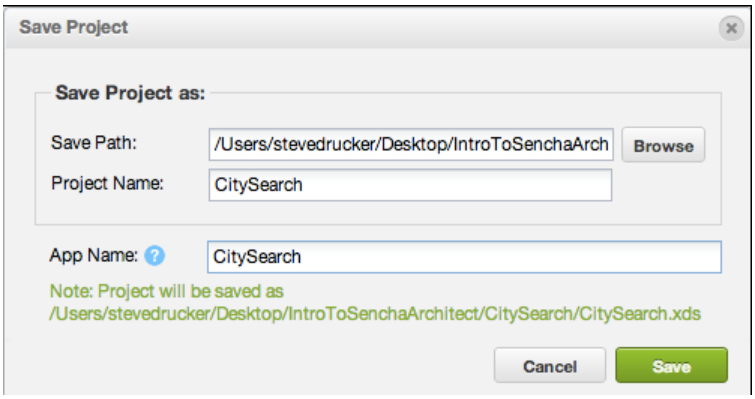


Figure 4: Saving your project

19. Click the **Save** button.
20. In the **Config** panel, set the **name** property to **CitySearch**.

Define a Data Access Model

21. In the **Toolbox**, click on the **Models** group
22. Drag a **Model** onto the Design Canvas
23. Set the following property in the **Config** panel for the Model:
    - userClassName: **Business**
24. In the Property Inspector, click the plus icon ⊕ to the right of the **Fields** heading to add a new field.
25. Enter a field name of **id**, and press [Enter]. A new field will appear in the Project Inspector.
26. Click on the **id** field in the Project Inspector.
27. In the Properties panel, set the **type** property to **int**
28. In the Project Inspector, click on the **Business** model.
29. In the **Config** panel, click on the [+] adjacent to the **Fields** property
30. Type the following string to add additional fields to the Business model (you do not need to specify data types):

```
name,latitude,longitude,address1,phone,mobile_url,
rating_img_url_small,photo_url
```

Define Data Stores

31. In the **Toolbox**, click on the **Data** Group
32. Drag a **JsonP Store** from the toolbox on to the design canvas.

33. Set the properties for the JsonP Store as follows:

    - userClassName: **Hotels**
    - model: **Business**
    - autoLoad: **true** (ensure box is checked)

34. In the Project Inspector, click on the **MyJsonPProxy** object for the Hotels store and set the URL property to the Business value contained in the file located in your /IntroToSenchaArchitect/helpers.txt file.

35. Click on the **MyJsonReader** object in the Hotels store and set the following property:

    - rootProperty: **businesses**

36. In the Property Inspector, right-click on the **Hotels** store and select **Load Data**. After a brief pause, you should see an eye icon appear to the right of the store reference.

37. Click on the eye icon to confirm that your JSON data is loading properly.

38. In the Project Inspector, right-click on the **Hotels** store and select **Duplicate**.

39. In the Project Inspector, click on MyJsonPStore and configure the following properties:

    - userClassName: Restaurants

40. In the Project Inspector, click on MyJsonPProxy1 and set the URL property to the Restaurants value contained in the file located in /IntroToSenchaArchitect/helpers.txt

41. Repeat steps 38-41 for the **Shops** store.

42. Save the Project

43. Click on the **Code** button and review the code generated for each of the stores.

44. Click on the **Build Web App** button.

45. Click on the Preview App button to test the application. Click the Preview button as illustrated in figure 6.

    You will notice the URL will contain a '_dc' parameter to automatically defeat any caching enabled.

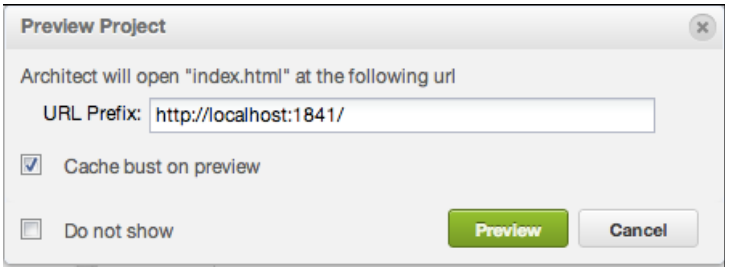


Figure 5: Sencha Cmd has a built-in Jetty webserver

46. In your browser's Debugger **Console** type the following to confirm that your Hotels data store contains data.

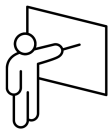
    ```
    Ext.getStore('Hotels').getAt(0)
    ```

    You will be presented with a structured tree view of data returned from the JSON data call using the URL that you entered into the MyJsonPProxy object

47. Using your file browser, review the contents of the **citysearch** folder that Sencha Architect created for you.

– End of Exercise –

Defining Views



Views output data to the user in a variety of formats. View classes covered in this course include the following:

- The **Component** class (Ext.Component) is typically used to output HTML markup that does not require scrolling. Components have the least amount of overhead of any of the view classes.
- The **Container** class (Ext.Container) outputs markup that may require scrolling. It also enables you to dock toolbars at fixed positions (top,bottom,left,right) of your GUI and allows you to nest other types of view components.
- The **Tab Panel** (Ext.tab.Panel) allows the user to toggle between several views that are typically represented by an iconic button. Tapping on the button brings the associated view to the "top" of a "card" layout.
- **Lists** (Ext.dataview.List) are linked to a data Store and are typically used to display a small subset of a record's information (such as firstname,lastname) for the purposes of data-drilldown or master-detail scenarios.

Other useful view classes include:

- The **NavigationView** (Ext.navigation.View) is an Ext.Container with a card layout that supports beautify animated transitions between views and dynamically generates "back" buttons to make your app easily navigable.
- The **Form Panel** (Ext.form.Panel) enables you to collect data from a user and submit it to an application server.
- The **Map class** (Ext.Map) enables you to easily integrate Google Maps into your application.
- **Carousels** (Ext.carousel.Carousel) are similar to tabs in that they enable the user to quickly toggle through different content views. Unlike tabs, however, moving between views is accomplished by a finger swipe instead of a button tap.
- **Panels** are used as overlays to display content that "floats" over your application.
- **CartesianChart** is used to display line graphs, bar charts, and other types of charts that use an x/y coordinate system.

Lab 2: Defining the Viewport

In this lab, you will perform the following tasks:

- Define a Container to act as the Viewport
- Define a Toolbars
- Define a Segmented Button

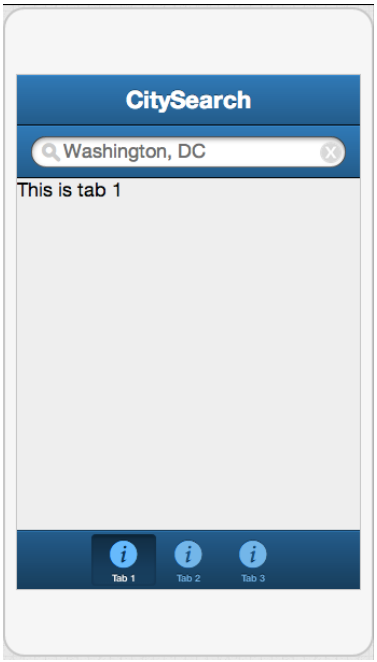


Figure 6: The result from this lab

Steps

Define the Viewport

1. Open the **CitySearch** project in Sencha Architect
2. Click on the **Design View** button to display the Design Canvas
3. From the Toolbox, drag a **Tab Panel** into the Design Canvas and configure the following properties:
   - intialView: **checked**
   - userClassName: **Main**
   - animation: **flip**
4. In the Config panel click on the [+] adjacent to the **Tab Bar Config** property.
5. In the Project Inspector, click on the **Tab Bar** component.
6. In the Config panel, configure the following property:
   - docked: bottom

Define the Main Titlebar

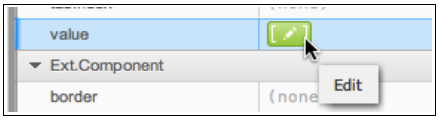
7. From the Toolbox, drag a Titlebar component and drop it onto the **Main** view in the Project Inspector.
8. Double-click on the center of the Titlebar and type the following text:
   - CitySearch

Define the Search Toolbar

9. From the Toolbox, drag a Toolbar component and drop it onto the **Main** view in the Project Inspector.

10. Configure the following Toolbar properties:
    - pack: center
11. Click on the Forms section in the toolbox, then drag a **Search Field** onto the toolbar and configure the following properties:
    - placeholder: **Search**
    - itemId: **searchfield**
    - flex: **1**
    - label: **(none)**
    - value: **Washington, DC**

    Note: you will see a '[…]' in the value field shown below.

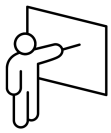


    Clicking this will open a screen in Code View in the center editor panel. Enter "Washington, DC" and then click the Design View button to return to the system design view.

12. In the Project Inspector, click on Tab 1 of the Main view and configure the following property:
    - html: This is tab 1
13. Repeat step 12 for **Tab 2** and **Tab 3**, entering in the appropriate HTML text to differentiate the content on each tab.
14. Save the Project
15. Click the **Preview** button. Click the Preview button on the dialog box to approve the URL prefix.
16. Test your work.

– End of Exercise --

Visualizing Data with List Views



Use the Ext.dataview.List component to output rows of data records. Lists are always bound to an Ext.data.Store, automatically looping through store and generating markup by passing model instance data through an Ext.XTemplate.

You will typically configure the following Ext.dataview.List properties:

Config Property	Description
store	Required Ext.data.Store. Supplies the data to generate the output markup.
itemTpl	Required Ext.XTemplate. An Ext.XTemplate that is invoked for each model instance in the store, outputting markup to each item in the Ext.List.
grouped	Boolean. Whether to group items under a common heading.
onItemDisclosure	Boolean/Function/Object. True to display a disclosure icon (right arrow).
variableHeights	Boolean. Whether the list contains items of variable height (typical when items contain image elements or line breaks). Set variableHeights to false to improve scrolling performance.
itemHeight	Integer. The height, in pixels, of each line in the list. Should be set if variableHeights is set to false.

Working with XTemplates

The Ext.XTemplate class generates markup from abstract data structures. XTemplate syntax supports the following:

- Automatic looping through arrays and records
- Conditional processing with if and switch constructs
- Executing inline Javascript code
- Custom template functions
- Basic math function support

Templates can be bound to an Ext.panel.Panel or to an Ext.view.View (xtype: dataview). The primary difference between these view classes is the following:

- Panels automatically update whenever their setData() method is explicitly invoked.
- Ext.view.View instances are bound to an Ext.data.Store and automatically refresh their contents whenever data in the Store is changed.

You can output the data from a Model by wrapping the field names in curly braces as illustrated below:

```
var tpl = Ext.create('Ext.XTemplate',
  '<ul>',
  '<tpl for=".">',
    '<li>{fName} {lName}</li>',
  '</tpl>',
  '</ul>'
);

var markup = tpl.apply([
  {fname: 'Steve', lName: 'Drucker'},
  {fname: 'Jason', lName: 'Perry'}
]);
```


Lab 3: Defining a View

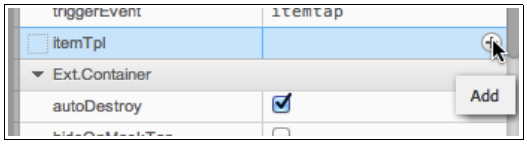
In this lab, you will perform the following tasks:

- Define and configure a List view
- Define an XTemplate
- Promote a View to a Class

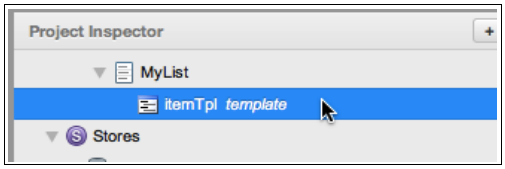
Steps

Define a List View

1. Open the CitySearch project in Sencha Architect
2. Click on the **Design** button to display the Design Canvas
3. In the Project Inspector, right-click on Tab 1 and select **Transform** > **Ext.dataview.List**
4. Configure the following List properties:
   - title: Hotels
   - iconCls: home
   - store: Hotels
   - html: (none)
5. In the **itemTpl** field, click the (+) icon at the far right of the row as shown below to add new template code.



This will add a new item below the **Hotels** view in the Project Inspector. Click this new item, and you will see a code window appear in the center window.



6. Set the itemTpl of the List to the following string (note: you can copy and paste this from your helpers.txt file.

```
<img class="photo"
     src="{photo_url}"
     align="left"
     style="margin-right: 5px; width:40px; height:40px;
margin-right: 5px"/>
{name}<br/>
<img src="{rating_img_url_small}"/> 
<small>{address1}</small>
```

Figure 7: The code snippet editor

7. Click the checkmark icon in the top-right of the code snippet editor as illustrated by Figure 8.
8. Click the **Design View** button to review the view.
9. In the Project Inspector, right-click on the **Hotels** store and select **Load Data**. You should see the **List** view reappear with data in it.

Promote the View to a Class To Enhance Code Reuse

10. In the Project Inspector, right-click on the **Hotels** list and select **Promote to Class**
11. In the Project Inspector, click on the new **MyList** Class and configure the following properties:
    - userAlias: **businesslist**
    - userClassName: **BusinessList**
12. Deploy the project and test in a browser.

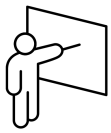
Define New Instances of the View

13. In the Project Inspector, click on the **Main** view.
14. In the Project Inspector, right-click on the **Tab 2** view and select **Delete**
15. In the Project Inspector, right-click on the **Tab 3** view and select **Delete**

16. In the Project Inspector, click and drag the **BusinessList** view onto the **Main** view.
17. Click the **Link** button
18. In the Project Inspector, click on the businesslist1 view and enter the following configuration properties:
    - title: **Restaurants**
    - store: **Restaurants**
    - iconCls: **favorites**
19. In the Project Inspector, click and drag the **BusinessList** view onto the Main View.
20. Click the **Link** button
21. In the Project Inspector, click on the **businesslist2** view and enter the following configuration properties:
    - title: **Shopping**
    - store: **Shops**
    - iconCls: **time**
22. Save and test your work

– End of Exercise --

Handling Interactions with Controllers



Controllers perform the following functions:

- Dynamically loads model, store, and view classes
- Creates an instance of each store class that is loaded.
- Attaches event listeners to visual components
- Responds to events thrown by visual components

By convention, Controllers are named after the data store that they primarily interact with. There is typically a one-to-one relationship between controllers and stores, and a one-to-many relationship between controllers and views.

Controllers typically have the following structure:

```
Ext.define('MyApp.controller.Beers', {
 extend: 'Ext.app.Controller',
 config: {
  models: ['MyApp.model.Beer'],
  stores: ['MyApp.store.Beers'],
  views: [
         'MyApp.view.beer.List',
         'MyApp.view.beer.Detail'
  ],
  refs: {
     /* references to view components */
  },
  control: {
    /* attach event listeners here */
  }
 }
 /* event listener handler methods go here */
});
```


Using Ext.ComponentQuery to Locate Instances

Refs use **Ext.ComponentQuery** syntax to define aliases to instantiated view components. The Ext.ComponentQuery expression that you create should be specific enough so as to return a maximum of one single component instance.

Ext.ComponentQuery query syntax is patterned after CSS selectors. Assume the following view class definition:

```
Ext.define("MyApp.view.MyContainer",{
  extend: 'Ext.Container',
  xtype: 'mycontainer',
  config: {
   layout: 'hbox',
   defaults: {
     xtype: 'component'
   },
   items: [
    {
      html: 'Hello',
      cls: 'white',
      flex:2,
      itemId: 'whitebox'
    },
    {
      html: 'World',
      cls: 'black',
      flex:1,
      itemId: 'blackbox'
    }
   ]
  }
});
```

Given the class definition above, the following Ext.ComponentQuery.query() expressions are valid:

```
// Returns all instances of the "mycontainer" view
Ext.ComponentQuery.query("mycontainer");

//Returns all instances of the component with an itemId
//of "whitebox" that is nested exactly one level deep
// in the items array of mycontainer.
Ext.ComponentQuery.query("mycontainer > #whitebox");

// Returns all instances of the component with an
// itemId of "whitebox" that is nested at any level
// in the items array of mycontainer.
Ext.ComponentQuery.query("mycontainer #whitebox");
```

Defining Controller Refs and Event Handlers

Refs use Ext.ComponentQuery syntax to identify a single component instance within your app and create a getter() method to it. Using refs are a convenient way to get access to a component instance from within a custom event handler. They can also significantly improve the overall performance of your app since the results from evaluating a ref are cached.

Refs can also be used in the control block of a controller in order to assign custom event handlers

```
Ext.define('CitySearch.controller.Main', {
 extend: 'Ext.app.Controller',
 config: {

   views: ['BusinessList'],

   refs: {
    fitlerfield: 'searchfield#filter'
   },

   control: {
    "filterfield": {
       keyup: 'onFilterKeyup',
       change: 'onFilterChange'
    }
   },

   onFilterKeyup: function(textfield, e, eOpts) {
     if (e.event.keyCode == 13) {
       this.onSearch(textfield.getValue());
     }
   },

   onFilterChange: function(textfield, newValue,
                                        oldValue, eOpts) {
     this.onSearch(newValue);
   },

   onSearch: function(searchterm) {
      Ext.Msg.alert("Searching...","To be continued...");
   }

});
```

The Controller detailed in the preceding code snippet is performing the following tasks:

- Dynamically loads the file app/view/BusinessList.js
- Defines a reference named “filterfield” that points to a searchfield component with an itemId of “filter”
- Defines two event handlers for the searchfield
- The keyup event listener takes action if the user taps the “search” button on the iOS virtual keyboard

- The change event listener fires when the user modifies the cotnents of the search field.

As illustrated below, the online documentation system enables you to quickly lookup the events that are supported by a component as well as the arguments that are automatically passed to custom handlers:

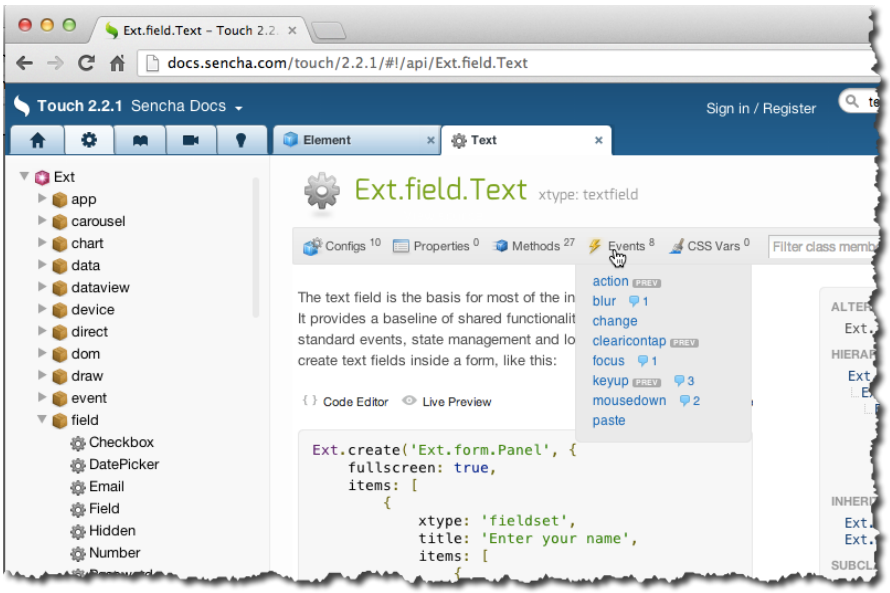
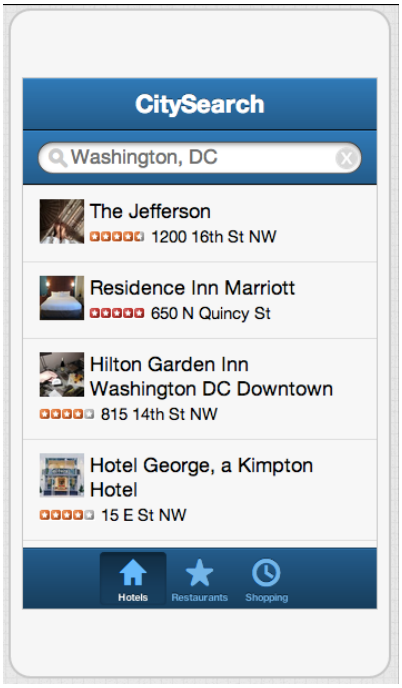


Figure 8: Events supported by the Ext.field.Text class.

Lab 4: Defining Controllers and Actions

In this lab, you will perform the following tasks:

- Define a Controller
- Link the Controller to Models, Stores, and Views
- Define an interaction



Steps

Define a Controller

1. Open the **CitySearch** project in Sencha Architect
2. Click on the **Design** button
3. Click on Behaviors in the Toolbox and drag a **Controller** onto the **Controllers** folder in the **Project Inspector**
4. Assign the following properties to the Controller:
   - userClassName: **Main**

Define a Controller Reference

5. In canvas, right-click on your app's search field and select **Global Controller** > **Add Reference** > **Main**

Define Event Handlers

6. In the design canvas, right-click on the **search** field and select **Global Controller** > **Add Action** > **Main**
7. In the Project Inspector, click on the Main controller's searchfield reference and configure the following property:
   - ref: filterField
8. In the Project Inspector, click on the Main controller's **ControllerAction**

9. Configure the following properties:

   - controlQuery: filterField
   - name: keyup
   - fn: onFilterFieldKeyUp

10. Enter the following code into the Editor to run a controller function named onSearch if the user taps the iOS virtual Go key.

```
if (e.event.keyCode == 13) {
  this.onSearch(textfield.getValue());
}
```

Listen for another event

11. In the Project Inspector, right-click on the **onFilterFieldKeyup** function and select **Duplicate**
12. In the Project Inspector, click on the function that you just duplicated and configure the following property:

    - name: change
    - fn: onFilterFieldChange

13. In the Project Inspector, in the Main Controller, right-click on the **onFilterFieldChange** handler and select **Edit Code**
14. Replace the code in the editor with the following:

```
this.onSearch(newValue);
```

Handle the Search Operation

15. In the Project Inspector, click on the **Main** controller.
16. In the Config panel, click the [+] adjacent to the **Functions** property and enter the following config properties:

    - Name for the function: **onSearch**
    - Parameters: **searchterm**

(this space intentionally left blank)

17. In the Project Inspector, double-click on the onSearch function and enter the following code to pass data to Yelp and update the views.

    (note: you can copy and paste this code from the helpers.txt file

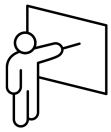
```
var baseUrl =
"http://api.yelp.com/business_review_search";

var stores = [
    {
        store: Ext.getStore('Hotels'),
        term: 'hotels'
    },
    {
        store: Ext.getStore('Restaurants'),
        term: 'eat'
    },
    {
        store: Ext.getStore('Shops'),
        term: 'shops'
    }
];

for (var i=0; i<stores.length; i++) {
    stores[i].store.setProxy({
        type: 'jsonp',
        url: baseUrl,
        extraParams: {
            location: searchterm,
            ywsid: [insert your yws id here],
            term: stores[i].term
        },
        reader: {
          type: 'json',
          rootProperty: 'businesses'
        }
    });
    stores[i].store.load();
}
```

Test your App

18. Click **Save**
19. Test the app. Try entering different cities in the search field.

– End of Exercise --

Implementing GEO Features



The proliferation of mobile devices has ushered in a new breed of software -- "finder" apps that locate areas of interest, sorted by the user's proximity to them. Sencha Touch contains the following classes to assist you in building these types of apps:

Class	Description
Ext.util.GeoLocation	Provides a cross-browser compatibale class for retrieving the current location of the device.
Ext.Map	Wraps a Google Map with an Ext.Component. Uses the Google Maps API.

While most developers use Google Maps as their core mapping framework, you should note that Sencha Touch is capable of supporting virtually any maps API that has a javascript interface. For example, the Sencha Market contains an extension for Sencha Touch that wraps the OpenLayers maps api.

Figure 9: Adding GEO features

Mobile mapping applications typically support the following features:

- Conversion of a street address to a lat/lng coordinate (Geocoding)
- Dynamically calculating the distance between two points on the globe
- Plotting locations on a map

Geocoding Addresses

You can translate an address into latitude/longitude coodinates through a process referred to as "geocoding." In the Google Maps API, geocoding is accomplished by invoking the google.maps.Geocoder() method as illustrated by the following code sample.

```
// instantiate the Geocoder class
var geocoder = new google.maps.Geocoder();

// define a sample street address + zip
var address = "1400 16th Street NW 20036";

// Instantiate a "please wait" message
Ext.Viewport.setMasked({xtype:'loadmask', message:'Please
Wait...'});

// geocode the address
geocoder.geocode( { 'address': address },
function(results, status)
 {
  if (status == google.maps.GeocoderStatus.OK) {
    var lat=results[0].geometry.location.lat();
    var lng=results[0].geometry.location.lng();
    Ext.Msg.alert("Success!",lat + "," +lng);
  }

  // turn off please wait message
  Ext.Viewport.setMasked(false);
 }
);
```

Note the following:

- The geocoding service executes asynchronously, so you'll need to specify a handler function to capture the results from the service.
- To communicate to the user that an async process is running, consider displaying a "loadmask" as illustrated by line 8
- The geocoding service returns a results array and a status message. If the geocoding operation was successful, status will be equal to the constant google.maps.GeocoderStatus.OK.
- The location of the address as a latitude,longitude point is available through results[0].geometry.location

Getting the Current Location

Using the `Ext.util.Geolocation` class enables you to retrieve the device's current geographc position at scheduled intervals.

As illustrated by the following code sample, determining a device's position is an asynchronous event due to the latency associated with locating the proper satellites from which to calculate a "fix". When the position of the device has been calculated, the class fires a `locationupdate` event that is passed the current location as an argument. You can programatically request that the current location be recalculated by invoking the Ext.util.Geolocation.updateLocation() method.

By default this class invokes the browser's native `navigator.geolocation` class.

```
var geo = Ext.create('Ext.util.Geolocation', {
 autoUpdate: false,
 listeners: {
  locationupdate: function(geo) {
     var currentLat = geo.getLatitude();
     var currentLng =  geo.getLongitude();
     var altitude = geo.getAltitude();
     var speed = geo.getSpeed();
     var heading= geo.getHeading();
  },
  locationerror: function(geo, bTimeout,
bPermissionDenied, bLocationUnavailable, message) {
     if(bTimeout)
       Ext.Msg.alert('Timeout occurred',"Could not get
current position");
     else
       alert('Error occurred.');
     }
  }
 }
});
geo.updateLocation();
```

Additional configuration options include the following:

Config Property	Description
AllowHighAccuracy	Boolean. Defaults to false. When set to true, the user will typically be prompted to approve the request. Using this option may result in higher latency and greater power consumption as it activates the device's internal GPS sensor instead of calulating position using wi-fi networks
frequency	Number. Defaults to 10000. The interval to use for re-calculating the current position.

autoUpdate	Boolean. Defaults to true. When true, the device's position is calculated at regular intervals, determined by the value of the frequency config property.
timeout	Number. Defaults to Infinity. The maximum number of ms allowed to elapse between a location update operation and the corresponding locationupdate event being raised. If the current position was not successfully acquired before the timeout elapses a locationerror event is raised.

Calculating the Distance Between Two Points

You can calculate the distance between two points by using either the Haversine formula or the Spherical Law of Cosines. The following function reprsents a JavaScript implementation of the Haversine forumla:

```
var calcDistance = function(lat1,lng1,lat2,lng2) {

 var R = 3959; // use 3959 for miles or 6371 for km
 var dLat = (lat2-lat1) * Math.PI / 180 ;
 var dLon = (lng2-lng1)* Math.PI / 180;
 var lat1 = lat1 * Math.PI / 180;
 var lat2 = lat2 * Math.PI / 180;

 var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
Math.sin(dLon/2) * Math.sin(dLon/2) * Math.cos(lat1) *
Math.cos(lat2);
 var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));

 return R * c;
}
```

Math is hard. See the Movable Type website for more detail regarding the mathematics of calulating distance from lat/lon.

Instantiating a Map View

The Ext.Map class (xtype: map) is a simple wrapper for a Google Map. You will typically set the following configuration properties:

Config Option	Description
mapOptions	Object. A passthrough for the google.maps.Mapoptions object specification.
useCurrentLocation	Boolean/GeoLocation. Set to true to center the map on the device's current coordinates. Alternately, pass an Ext.util.Geolocation instance to control update intervals.

Configuring the Map

Developers typically need to configure the following google.maps.Mapoptions:

Property	Description
center	LatLng. The center of the map. Use the google.maps.LatLng() method to translate lat,lng coordinates to a google map LatLng object.
mapTypeId	The type of map to display. May be one of the following: • google.maps.MapTypeId.HYBRID • google.maps.MapTypeId.ROADMAP • google.maps.MapTypeId.SATELLITE • google.maps.MapTypeId.TERRAIN Defaults to google.maps.MapTypeId.ROADMAP
zoom	Number. The initial zoom level (1-20)

The following example illustrates how to instantiate a Google Map that is centered on Fig Leaf Software's World Headquarters:

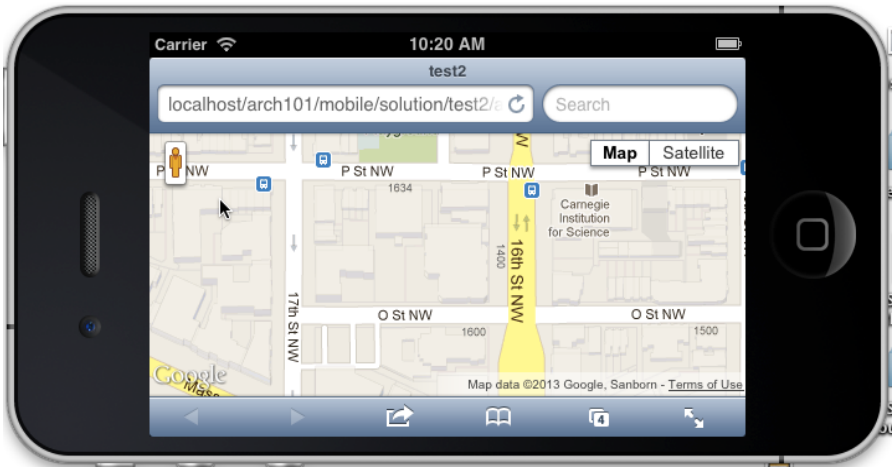


Figure 10: Centering a map on a specific location

```
Ext.Viewport.add([
 {
  xtype: 'map',
  mapOptions: {
   center: new google.maps.LatLng (38.909027,-77.037165),
   mapTypeId: google.maps.MapTypeId.ROADMAP,
   zoom: 17
  }
 }
]);
```

Packaging Complex Functionality into Components

As illustrated in figure 12, Sencha Architect enables you to export classes into packages that can be shared among your coworkers and their many projects.

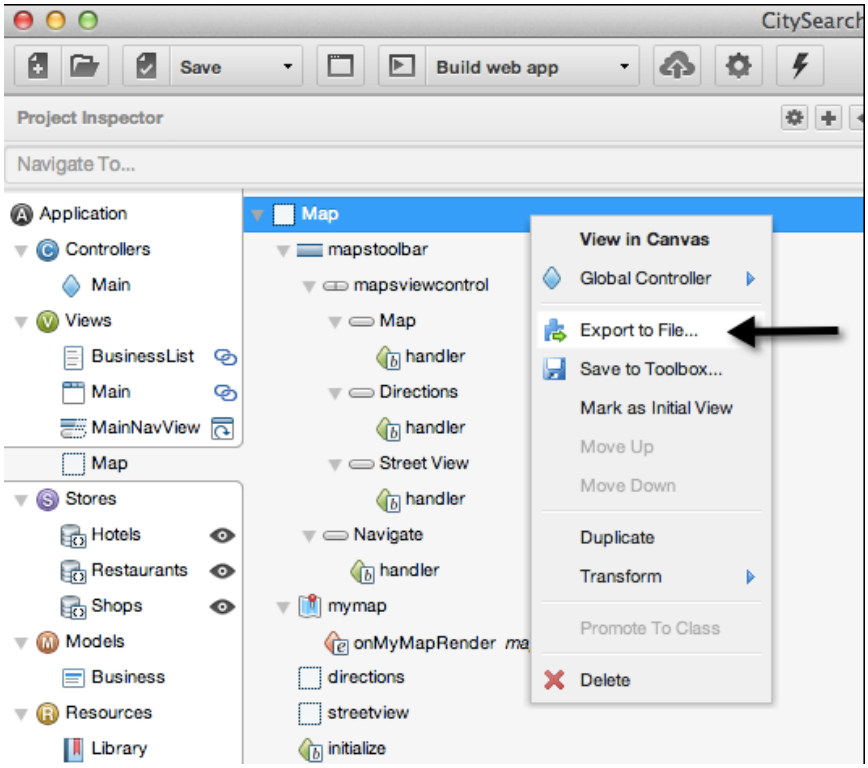


Figure 11: Exporting a shareable component

Lab 5: Adding GEO Features

In this lab, you will perform the following tasks:

- Implement a Navigation View to auto-generate a “back” button.
- Import a Component
- Deploy a Map

Steps

Refactor the App to use a Navigation View

1. Open the **CitySearch** project in Sencha Architect
2. Click on the **Design** button
3. Click on **Views** in the Toolbox and drag a **Navigation View** onto the **Views** folder in the **Project Inspector**
4. Assign the following properties to the Controller:
   - userClassName: MainNavView
   - initialView: checked
   - animation: fade
5. In the Project Inspector, drag the **Main** view and drop it onto the **MainNav** view**.**
6. Click the **Link** button.
7. Since the Navigation View will auto-generate a titlebar, remove the titlebar from the Main View.
8. In the Project Inspector, click on the **Main** view
9. Right-click on the **CitySearch** titlebar and select **Delete**
10. In the Project Inspector, click on MainNavView
11. Click on the **main1** subview and configure the following property:
    - title: CitySearch
12. Save and test the application. It should continue to operate as before.

Add a reference to the Navigation View

13. In the Project Inspector, right-click on **MainNavView** and select **Global Controller** > **Add Reference** > **Main**
14. Configure the reference using the following properties:

    - ref: mainNavView
    - selector: navigationview

Install the Maps Class

15. In Sencha Architect, select **Edit** > **Import Component**
16. Select the **{desktop}/IntroToSenchaArchitect/Map.xdc** file
17. Enter the following properties:
    - Name: Maps
    - Group: Views
    - Category: Containers
18. Click the **OK** button.
19. Verify that the Maps class is now available in your Toolbox.
20. Drag the Maps class from the Toolbox and drop it onto Views in the Project Inspector.
21. Configure the following properties:

    - userAlias : maps

Add a Controller Action

22. In the Project Inspector, right-click on the businesslist view and select **Global Controller** > **Add Action** > **Main**
23. Configure the action using the following properties:

    - controlQuery: businesslist
    - name: itemtap
    - fn: onBusinessTap

24. In the Project Inspector, double-click on the **onBusinessTap** function to enter code view.
25. Enter the following code to invoke the Map view:

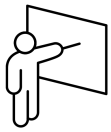
```
this.getMainNavView().push({
    xtype: 'maps',
    title: record.get('name'),
    lat: record.get('latitude'),
    lng: record.get('longitude')
});
```

Load the Google Maps API

26. Click the [+] on the Project Inspector and select **Resources** > **Google Maps API**
27. Save and test the application.

– End of Exercise –

Theming Your Application



Sencha uses Sass and Compass as the foundation of their theming engine for both Sencha Touch and Ext JS 4.

Sass - Syntactically Awesome Style Sheets, is a style sheet compiler that interprets the SCSS (Sassy CSS) meta language and outputs production-ready CSS. SASS extends CSS3 by adding nested rules, variables, mixins, selector inheritance, and other useful features. It's available under an MIT license from **http://sass-lang.com.**

Compass is an open-source CSS authoring framework, providing a design pattern for SASS, that makes your stylesheets smaller and easier to maintain. Compass Mixins (covered later in this unit) help you to normalize the inconsistencies in CSS3 support as indicated by the following example:

SCSS	Generated CSS
`.myCustomClass { @include border-radius(5px); }`	`.myCustomClass{ -webkit-border-radius:5px; border-radius:5px }`

About Sass

SASS enables you to add programatic features to your CSS declarations by supporting following constructs:

- Variables
- Nesting
- Inheritance
- Math Functions
- Color Transformations
- Mixins (parameterized functions)

Declaring and Using Variables

Using SASS you can declare variables that can be used throughout your stylesheet, thereby enabling you to make global changes by tweaking a single value.

Variables begin with a dollar sign ($) and are declared just like properties. They can have any value that's allowed for a CSS property, such as colors, numbers (with units), or text.

The following example illustrates the use of variables and their affect on the generated CSS file:

Sass	Generated CSS

`$silver: #c0c0c0;`
`.myClass {`
`background-color: $silver;`
`}`
	`.myClass {`
	`background-color:`
	`#c0c0c0;`
	`}`

Invoking Mixins

Mixins are user-defined functions that generate CSS directives.

- Define mixins using the @mixin command.
- Invoke mixins using the @include command.

Sass	Generated CSS

`@mixin dashedBorder($color, $width)`
`{`
`border: {`
`color: $color;`
`width: $width;`
`style: dashed;`
`}`
`}`
 	
`.validContent {`
`@include dashedBorder(green, 1px);`
`}`
`.invalidContent {`
`@include dashedBorder(red, 2px);`
`}`
	`.validContent {`
	`color: green;`
	`width: 1px;`
	`style: dashed;`
	`}`
	
	`.invalidContent {`
	`color: red;`
	`width: 2px;`
	`style: dashed;`
	`}`

Modifying the Sencha Touch Default Theme

Modifying a series of global variables can significantly impact the look and feel of your application. Most of these variables are documented within Sencha's official API docs under the Global_CSS heading as illustrated below:

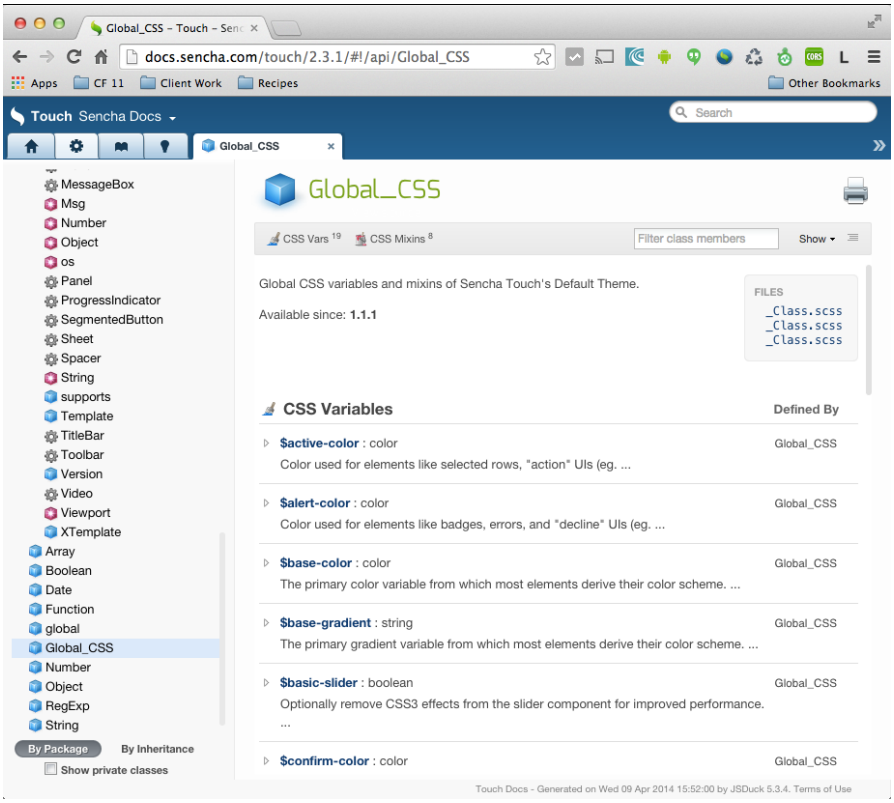


Figure 12: Reviewing Global CSS Sass Variables

Configuring these Sass variables enables you to quickly and radically change the appearance of your application:

Variable	Description
$base-color	The primary color variable from which most elements derive their color scheme. Defaults to #1985D0
$base-gradient	The primary gradient variable from which most elements derive their color scheme. Valid options include bevel, glossy, recessed, and matte (default)
$font-family	The font-family to be used throughout the theme. Defaults to "Helvetica Neue"

$alert-color	Color used for elements like badges, errors, and "decline" UIs (eg. on buttons). Defaults to "red"
$confirm-color	Color used for elements like success messages, and "confirm" UIs (eg. on buttons). Defaults to #92cf00 (green)
$active-color	Color used for elements like selected rows, "action" UIs (eg. on buttons) and certain overlays like the picker mask. Defaults to darken(saturate($base-color, 55%), 10%)
$neutral-color	Color used for the neutral 'ui' for Toolbars and Tabbars. Defaults to #e0e0e0 (white-gray)
$page-bg-color	The background color for fullscreen components. Defaults to #eee (gray-white)
$global-row-height	The minimum row height for toolbars. Defaults to 2.6em
$global-list-height	The minimum height for list items. Defaults to 46px.

You can set these variables in Sencha Architect by clicking on the MyDefaultTheme entry in the Project Inspector and then opening the Theme panel in the Config section.

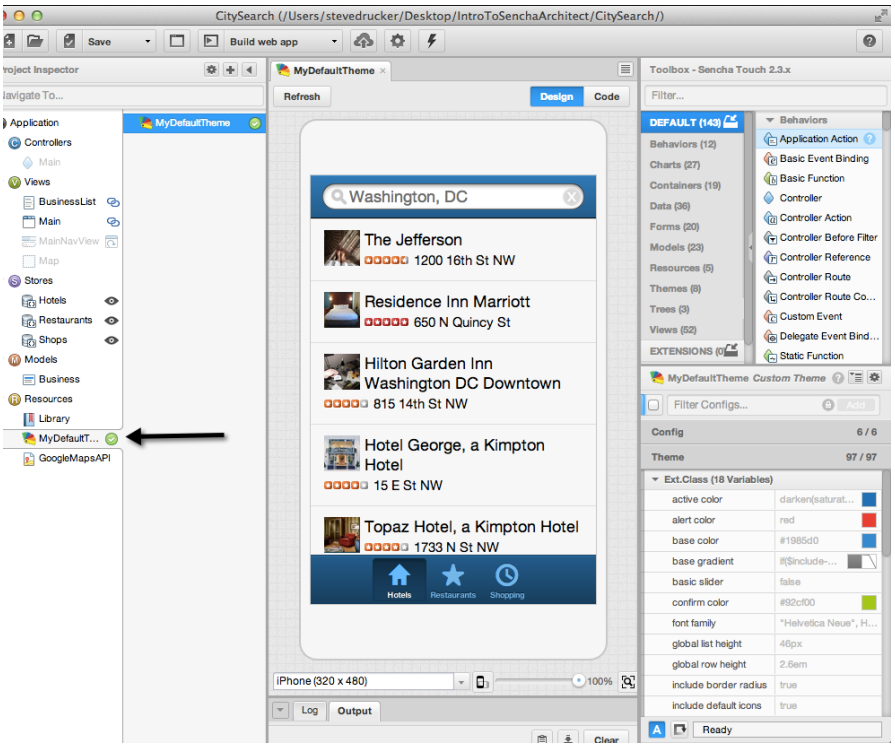


Figure 13: Modifying Basic Theme Properties

Using Web Fonts and Typography

iOS webkit supports a large number of fonts, however, very few of these are also supported by Android's native browser. To ensure that your applications uses the same font across a wide variety of devices you should make use of Webkit's support for downloadable fonts.

Google Web Fonts contains over 600 free, open-source fonts that have been optimized for the web. Adobe Edge Web Fonts supports another 500+ royalty free fonts.

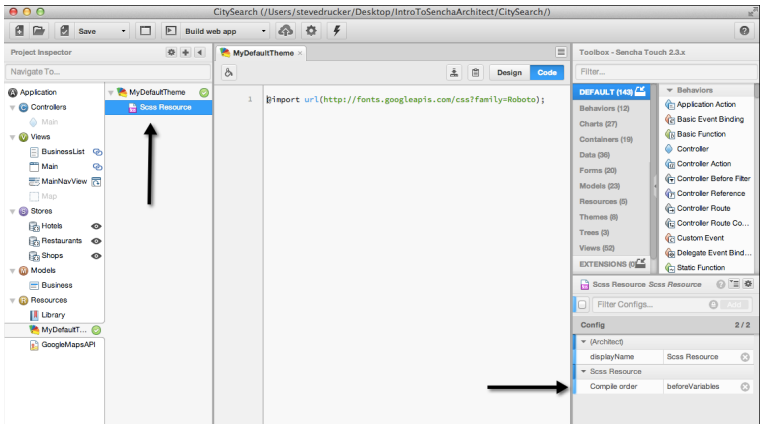
Downloading a Font

CSS-3 capable browsers can download fonts and use them to format your text. Use the @font-face selector to define a font name and associated download url. You can then reference the font in subsequent style definitions as indicated below:

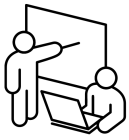
```
<style>
@font-face {
  font-family: 'Droid Sans';
  src: url(fonts/Droid_Sans.ttf);
}

h1{
  font-family: 'Droid Sans';
}
</style>
```

Adding Downloadable Fonts and Style Overrides to your Theme

You can add an SCSS override file by right-clicking on your theme in the Project Inspector and selecting “Add New Scss Resource”. This feature enables you to code custom Sass/CSS directives and indicate when they should be executed during the theme compilation process.



Lab 6: Theming



In this lab, you will perform the following tasks:

- Change the "base color" of the app
- Deploy a web font

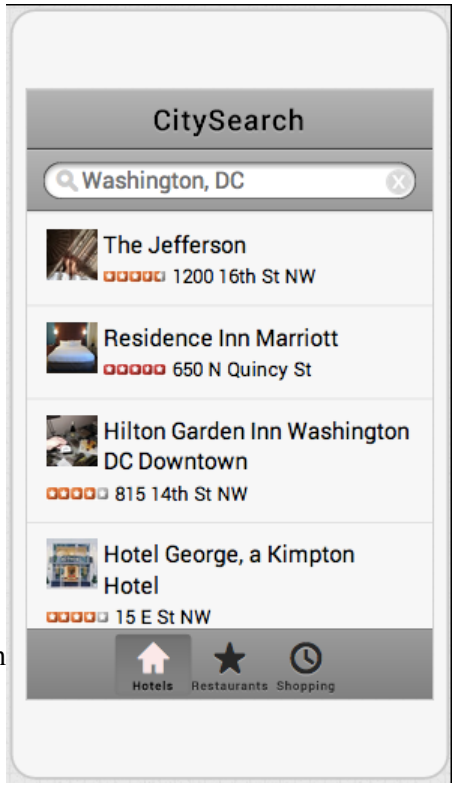


Figure 14: Applying base-color and downloadable fonts

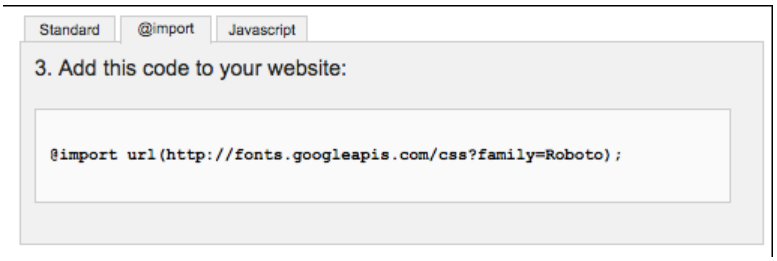
Steps

Modify the App's Base Color

1. Open the **CitySearch** project in Sencha Architect
2. In the Project Inspector, click on Resources > MyDefaultTheme
3. In the Properties Panel, click on the Theme bar to expose the theme options.
4. Set the base color value to #c0c0c0 (or whatever color you'd like)

Deploy a Web Font

5. Open a browser to http://www.google.com/fonts
6. Click on the "Quick Use" button for the **Roboto** font.
7. On item #3, Click on the @import tab as illustrated below:

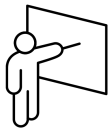


8. Copy the @import code to your clipboard
9. Return to Sencha Architect
10. In the Project Inspector, right-click on **MyDefaultTheme** and select **Add scss resource**

11. In the properties panel, set the following property for the scss resource:
    - Compile order: beforeVariables
12. Paste the contents of your clipboard into the new scss resource
13. In the Project Inspector, click on **MyDefaultTheme**
14. In the config panel, enter **Roboto, sans-serif** as the value for **font-family**.
15. Save and test (note that you may need to clear your browser's cache in order to see the changes)

– End of Exercise –

Packaging a Native App



Sencha Cmd and Sencha Architect integrate with Apache Cordova and Adobe PhoneGap to package web applications for distribution on app stores as well as access native device functionality through a JavaScript API.

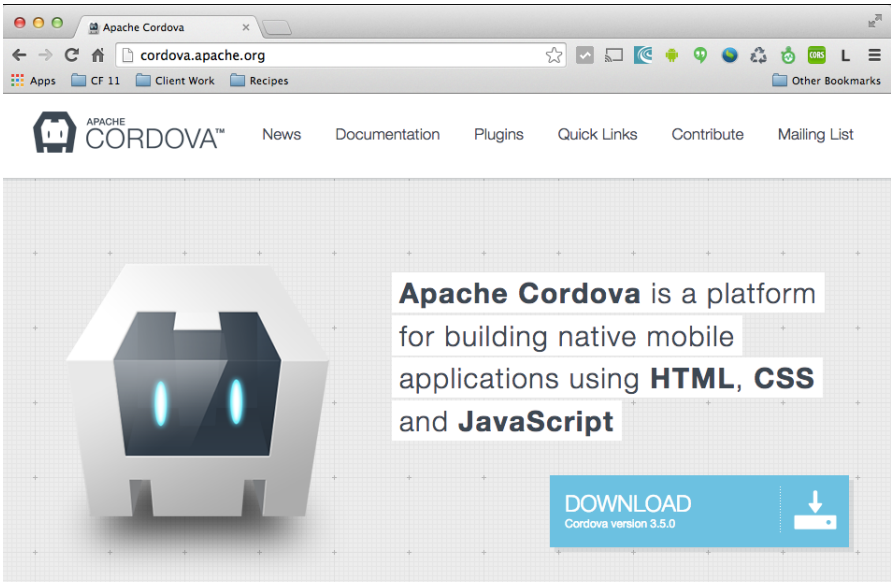


Figure 15: The Apache Cordova Home Page

To get started, you will need to download and install Cordova by using the npm command-line interface. The easiest way to install npm is to download and install Node.JS from http://nodejs.org/download

Once Node.JS is installed, you can simply open a command prompt and install cordova by typing the following command:

```
npm install -g cordova
```

To install Cordova support into your application, open a command prompt to your app's root folder and enter the following command:

```
sencha phonegap init [APP_ID] [APP_NAME]
```

The APP_ID / APP_NAME should be entered using the following pattern:

```
sencha cordova init com.figleaf.CitySearch CitySearch
```

This command will add a ./cordova folder to your app.

Adding Platform Support

Once Cordova has been installed into your app, use a text editor to modify your app's app.json file to add support for various mobile platforms as illustrated below:

```
{
    "builds": {
        "web": {"default": true},
        "native": {
            "packager": "cordova",
            "cordova" : {
                "config": {
                    // Uncomment the line below and add the platforms you wish to build for
                    "platforms": "ios android",

                    "id": "com.figleaf.CitySearch",
                    "name": "CitySearch"
                }
            }
        }
    },
```

Figure 16: Adding platform support for iOS and Android

Sencha Architect will automatically detect the presence of Cordova and make the following menu options available for your selection:

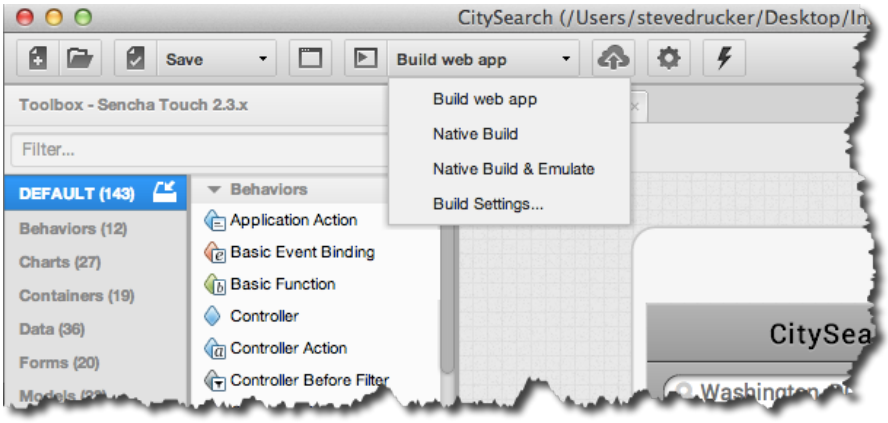


Figure 17: You can compile and run the app in emulators directly from Architect

Creating a Production Build

After selecting the Native Build & Emulate feature and testing in your iPhone and Android emulators, you will want to actually create a production build.

Cordova will auto-generate xcode and android projects that you can load into your device vendor's native development environments in order to complete the production build and distribution process.

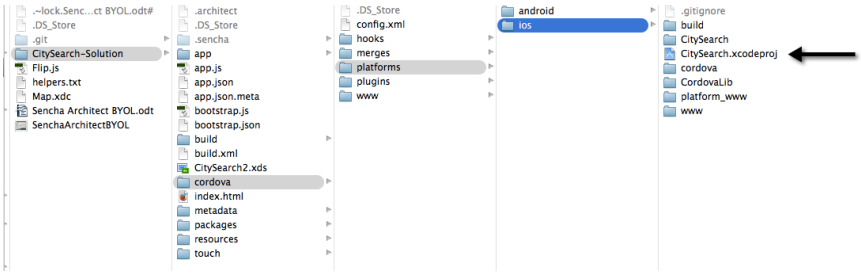


Figure 18: Cordova generates native iOS and Android project files

You can interface with native device resources by using Sencha's Ext.Device API and the Cordova API plugins.

Demonstration 7: Creating a Native Build

In this lab, you will perform the following tasks:

- Create a native IOS project
- Package and run the app in the iOS Simulator
- Open the generated project in XCode

Steps

Install the prerequisites

1. Install XCode
2. Install Node JS
3. Open a command prompt and type the following command:

```
npm install -g cordova
```

4. Change directories to the /CitySearch folder

Use Cordova to Package a Build for Testing

5. Enter the following command:

```
sencha cordova init com.figleaf.CitySearch CitySearch
```

6. Using a text editor, open /CitySearch/app.json
7. Configure the native build instructions as illustrated below:

```
"native": {
          "packager": "cordova",
          "cordova": {
             "config": {
                "platforms": "ios",
                "id": "com.figleaf.CitySearch",
                "name": "CitySearch"
             }
          }
  }
```

8. Open the CitySearch project in Sencha Architect. You should now see native build options available to you under the several more items available to you under the Build & Emulate button.
9. Click on **Native Build & Emulate**. After a few moments the app should launch in Xcode's iOS emulator.
10. Double-click on the generated XCode project located in /CitySearch/cordova/platforms/ios/CitySearch.xcodeproj to load it into XCode.

– End of Demonstration --

Unit Summary

- Sencha Architect enables you to rapidly prototype your mobile applications
- Using Sencha Architect can help you learn Sencha Touch syntax
- Sencha Touch includes a vast API for downloading data via XML or JSON and outputting data to a variety of different views.
- Sencha Touch apps are built using object oriented JavaScript in a Model-View-Controller design pattern.
- The Sencha Touch theming engine uses the Sass compiler.
- A number of plugins exist that new Views to Sencha Architect.
- Views can be exported and shared among developers.
- You can use Sencha Architect and Apache Cordova/Adobe PhoneGap to publish and package your applications.
- Sencha Touch apps run on iOS, Android, Kindle, Blackberry, and Windows Mobile operating systems.

Unit Review

1. Which devices does Sencha Architect emulate?

2. Sencha Architect can not display information loaded from a JSON store interactively in Design view (true/false)

3. You cannot customize the code generated by Sencha Architect (true/false)

4. You must have Sencha Cmd installed in order to publish your applications (true/false)

5. You may not import code into Sencha Architect (true/false)

6. In order to package your app for distribution on an app store, you must first compile it using which third-party tools?