SenchaTouch

Aplicaciones Web para móviles que se ven y se sienten como nativas

Antonio Javier Gallego Sánchez

Tabla de contenido

Contenidos

Existen muchas librerías o frameworks de desarrollo que están orientados a la creación de webs para móviles y que intentan darle el aspecto de una aplicación nativa. Una de las más conocidas y utilizadas en la actualidad es *Sencha Touch*. Este tipo de librerías son de gran ayuda ya que nos ahorran muchísimo tiempo en la programación ya que con un único desarrollo Web lo podemos aprovechar de forma generalizada para los diferentes sistemas de dispositivo móvil, como Android, iOS, etc.).

En este libro se tratan desde los aspectos más básicos de *Sencha Touch*, como la instalación de la libreria y la creación de una primera aplicación, hasta otros más avanzados como el almacenamiento de datos y el uso de componentes asociados a datos.

Los contenidos principales del libro son:

Introducción

Sencha Touch es un framework para el desarrollo de aplicaciones móviles centrado en WebKit (base de los navegadores web Safari, Google Chrome, Epiphany, Maxthon, y Midori entre otros). Fue el primer framework basado en HTML5 y JavaScript, además utiliza CSS3 para realizar animaciones. La apariencia de las aplicaciones desarrolladas es similar al de las aplicaciones nativas en Android, BlackBerry e iOS. Sencha Touch está disponible tanto en versión con licencia comercial como con licencia Open Source GPL v3.

Una de las principales ventajas de Sencha Touch es la cantidad de controles IU o elementos de interfaz que incluye, todos ellos muy fáciles de usar y personalizar. Ha sido diseñado específicamente para dispositivos táctiles por lo que incluye una amplia gama de eventos táctiles o *gestures*, que comúnmente son usados en este tipo de dispositivos. Además de los eventos estándares como *touchstart* o *touchend*, han añadido una extensa lista de eventos como *tap*, *double tap*, *tap and hold*, *swipe*, *rotate* o *drag and drop*.

Interoperabilidad con PhoneGap

Sencha Touch funciona perfectamente junto a *PhoneGap* (ver capítulo correspondiente), por lo que puede ser usado para distribuir nuestras aplicaciones en la *App Store* o en *Android Market*. Se basa en el uso de un mecanismo que empotra nuestra aplicación en una *shell*

nativa de la forma más sencilla posible. Además, gracias a *PhoneGap* podemos hacer uso de la API nativa del dispositivo para acceder a la lista de contactos, la cámara y muchas otras opciones directamente desde JavaScript.

Integración de datos

Al igual que con *ExtJS* (biblioteca de JavaScript para el desarrollo de aplicaciones web interactivas), Sencha Touch implementa el patrón de diseño MVC en el lado del cliente y nos ofrece una API rica y poderosa para manejar flujos de datos desde una increíble variedad de fuentes. Podemos leer datos directamente a través de AJAX, JSON, YQL o la nueva capacidad *local storage* de HTML5. Podemos enlazar esos datos a elementos específicos de nuestras vistas, y utilizar los datos sin conexión gracias a los almacenes locales.

Sencha Touch vs. JQuery Mobile

A continuación se enumeran las principales diferencias entre Sencha Touch y JQuery Mobile:

Sencha Touch:

- Tiene una curva de aprendizaje mucho mayor y necesita una mayor comprensión del lenguaje de programación JavaScript, pero gracias a esto proporciona una API mucho más potente.
- Dispone de un mayor número de controles para la interfaz de usuario, así como efectos de transición y animaciones entre páginas mucho más personalizables.
- Más rápido en mayor número de dispositivos móviles (en Android a partir de la versión 2.1). El comportamiento y velocidad de Sencha Touch es mucho mejor que el de otros frameworks, a excepción del tiempo de carga inicial, pues JQuery Mobile pesa menos.
- Al estar basado en ExtJS (usan el mismo núcleo), es muy robusto y potente, además de ser un framework ampliamente probado y usado (también debido a que fue uno de los primeros en aparecer).
- Al igual que en ExtJS, y en comparación con JQuery Mobile, se escribe mucho código. Esto podría ser tomado como un pro o como un contra. Es bueno porque indica una mayor potencia de configuración y personalización, pero por contra conlleva más tiempo de desarrollo y de aprendizaje.

JQuery Mobile:

- Muy sencillo de aprender y de implementar aplicaciones.

- Es necesario escribir muy poco código (y casi no se usa JavaScript) para lograr aplicaciones móviles muy interesantes. En lugar de orientarse a la programación JavaScript, JQuery Mobile se centra en usar etiquetas HTML con atributos definidos por el framework.
- No dispone de muchos controles para el diseño de la interfaz.
- Tiene una ejecución algo más lenta que Sencha Touch.
- Al estar basado en un framework muy desarrollado, como es JQuery, funciona correctamente en un mayor número de dispositivos móviles y de navegadores, como Symbian, Android, iOS, Blackberry, Window Phone 7 o WebOS.

Ambos frameworks son buenas opciones para el desarrollo de aplicaciones móviles. Los dos utilizan HTML5, JavaScript e integran la tecnología AJAX. La decisión dependerá de las necesidades de la aplicación a desarrollar. En principio, Sencha Touch es más apropiado para aplicaciones grandes, que necesiten de mayor personalización o configuración y que vayan a hacer un mayor uso del lenguaje de programación JavaScript. JQuery Mobile se suele utilizar para aplicaciones en las que se necesite una interfaz de usuario que conecte directamente con un servidor y que haga un menor uso de JavaScript.

Instalar Sencha Touch

En primer lugar descargamos el SDK de Sencha Touch desde su página Web "http://www.sencha.com/products/touch/download/". Se nos descargará un fichero comprimido con ZIP (llamado algo como "sencha-touch-VERSION.zip"), que descomprimiremos en una carpeta.

A continuación tenemos que instalar el Cmd de Sencha, el cual descargamos desde http://www.sencha.com/products/sencha-cmd/download/, lo descomprimimos e instalamos mediante el *wizard* que incorpora. Una vez instalado ya tendremos disponible desde la consola el ejecutable `sencha` para crear aplicaciones y gestionarlas.

> Nota: si el comando `sencha` fallara por cuestiones de permisos en Linux tendremos que reinstalar como un usuario normal en vez de como *root*.

Crear nuestra primera aplicación

Una vez instalado *Sencha Touch* y *Sencha Cmd* ya podemos crear nuestra primera aplicación. Para ello crearemos una nueva carpeta para la aplicación y entraremos en ella:

```
mkdir myapp
cd myapp
```

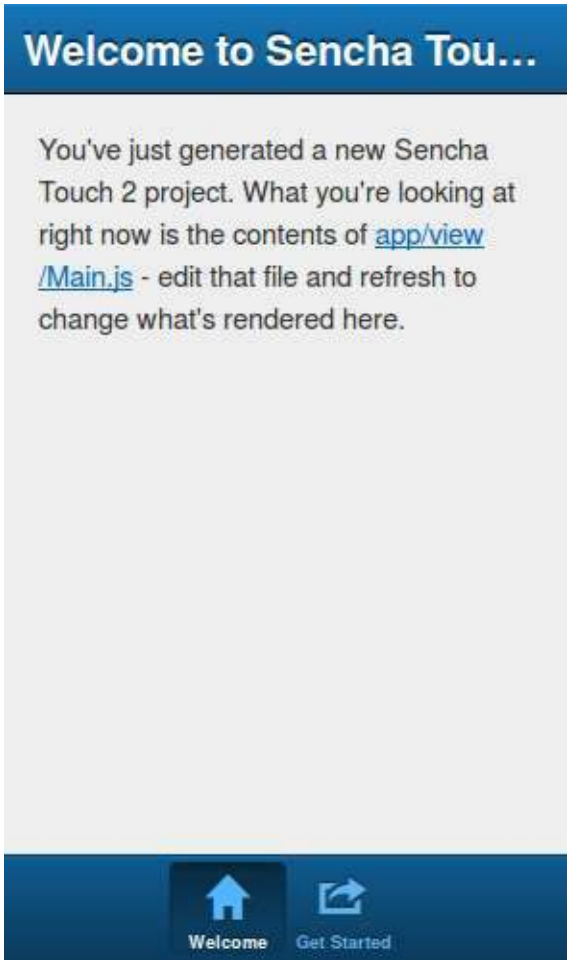
Y a continuación ejecutaremos el comando:

```
sencha -sdk /path/to/touch generate app MyApp .
```

Esto generará el esqueleto de una nueva aplicación de Sencha Touch en el directorio actual, con todas los ficheros que vamos a necesitar incluyendo un documento `index.html` inicial, una copia del SDK de Sencha Touch, las hojas de estilo, imágenes y ficheros de configuración.

Una vez creada la aplicación podemos comprobar que funcione correctamente de varias formas:

- Abriendo directamente el fichero `index.html`, la mayoría de aplicaciones básicas las podremos probar de esta forma.
- Copiando nuestro proyecto a un servidor web y accediendo al proyecto a través de la dirección: http://localhost/MyApp
- Usando el servidor Web que proporciona el Cmd de Sencha. Para activarlo tenemos que ir a la carpeta de nuestra aplicación y en un terminal ejecutar `sencha web start &`. Para acceder en este caso tendremos que usar la ruta: `http://localhost:1841/`



Sencha Touch solo funciona con navegadores basados en WebKit, como son: Safari, Google Chrome, Epiphany, Maxthon o Midori. Si lo probamos en un navegador que no lo soporte, como Firefox o Internet Explorer, solamente veremos una página en blanco o un resultado erróneo. Por lo tanto para probar nuestros proyectos Web tendremos que instalar uno de los navegadores soportados, como Google Chrome (http://www.google.es/chrome) o Apple Safari (http://www.apple.com/es/safari/).

Aunque la mayoría de webs que podemos hacer con Sencha Touch se podrían ejecutar y visualizar directamente sin necesidad de un servidor Web, sí que será necesario su uso si queremos probar nuestros proyectos utilizando algún emulador o un dispositivo móvil real.

En la sección inicial de "Instalación de un servidor Web" se puede encontrar información sobre la instalación de un emulador móvil o la configuración para el acceso externo mediante un dispositivo móvil real.

Estructura de carpetas

Las carpetas y ficheros principales que se generan con un nuevo proyecto son:

- *app* - En este directorio es donde se almacenan las vistas, modelos, controladores y

stores para la aplicación.
- *app.js* - Es el principal script o punto de entrada de nuestra aplicación, el cual ejecutará el código inicial.
- *app.json* - Es el fichero de configuración de nuestra aplicación.
- *index.html* - Fichero HTML principal de la aplicación, el cual se abrirá al iniciarla.
- *packager.json* - Fichero de configuración utilizado por Sencha Cmd para la generación de paquetes nativos de la aplicación.
- *resources* - Contiene todos los recursos o assets de la aplicación, como los CSS, imágenes, etc.

El fichero *index.html* es el primero que se abrirá al iniciar la aplicación. Este fichero únicamente contendrá la carga de javascripts y estilos y después le pasará el control al fichero *app.js*, el cual será el punto de entrada a la aplicación y donde irá todo el código de Sencha Touch.

Código HTML básico de una aplicación

Las aplicaciones de Sencha Touch se crean como un documento HTML5 que contiene referencias a los recursos de JavaScript y CSS. Nuestro fichero `index.html` debe de contener como mínimo el siguiente código:

```
<!DOCTYPE HTML>
<html manifest="" lang="en-US">
<head>
    <meta charset="UTF-8">
    <title>Mi aplicación</title>
    <script id="microloader" type="text/javascript"
        src=".sencha/app/microloader/development.js"></script>
</head>
<body>
</body>
</html>
```

A continuación analizaremos por separado cada una de las partes de este código:

```
<!DOCTYPE HTML>
<html manifest="" lang="en-US">
  ...
</html>
```

La primera línea nos indica que este es un documento del tipo HTML5. Las etiquetas de `<html>` y `</html>` indican el inicio y final del documento HTML y deben de contener en su interior todo el resto del código.

```
<head>
  <meta charset="UTF-8">
  <title>Mi aplicación</title>
  ...
</head>
<body>
</body>
```

Todo documento HTML (y HTML5 también) debe de contener primero una sección de cabecera (`<head>`) y a continuación una sección con el contenido principal o cuerpo del documento (`<body>`). En este caso el cuerpo del documento (`<body>`) se encuentra vacío. Esto se debe a que la librería Sencha Touch crea todo el contenido de la Web, incluidos todos los elementos de la interfaz, mediante código JavaScript.

La cabecera del documento (`<head>`) debe de contener como mínimo los metadatos acerca del tipo de contenido, el conjunto de caracteres usados, y un título que mostrará el navegador en la parte superior de la ventana. Además debe de contener un enlace a la librería de javascript de carga de Sencha Touch:

```
<script id="microloader" type="text/javascript"
        src=".sencha/app/microloader/development.js"></script>
```

La etiqueta `<script></script>` se utiliza para cargar código JavaScript en nuestra página Web. En este caso se carga únicamente el fichero `development.js` , el cual se encarga de cargar todo lo necesario para iniciar nuestra aplicación: ficheros con hojas de estilo, librerías de Sencha Touch de JavaScript y el fichero `app.js` con el código de nuestra aplicación.

El fichero `development.js` comprueba el dispositivo y navegador en el que se ha cargado la aplicación y configura las hojas de estilo, javascripts y otros parámetros para que la aplicación se adapte lo mejor posible al mismo.

Este fichero se utiliza para entornos de desarrollo, pero para producción o testing tendremos que compilar la aplicación mediante Sencha Cmd o generar nuestra propia compilación. En futuras secciones se tratará más en profundidad este tema.

Ahora ya tenemos cargadas las librerías de Sencha Touch y el código de nuestra aplicación para empezar a trabajar. De momento, si lo visualizamos en un navegador solo veremos una página en blanco ya que el código de nuestra aplicación (`app.js`) de momento está vacío.

Compilar un proyecto

Al crear un nuevo proyecto este viene preparado para un entorno de desarrollo, por lo que si queremos obtener la versión final de producción con el código optimizado para su utilización tendremos que ejecutar (dentro de la carpeta del proyecto):

```
sencha app build production
```

Este comando creará una versión de producción de nuestro proyecto y la colocará en la carpeta `build/production/<nombre-app>`. Como esta carpeta también estará en el servidor para probarla podremos acceder a la dirección: `http://localhost/<nombre-app>/production/<nombre-app>`. Para más información sobre este proceso podéis consultar las direcciones:

http://docs.sencha.com/touch/2.4/getting_started/using_creating_builds.html

http://www.sencha.com/blog/getting-started-with-sencha-touch-2-build-a-weather-utility-app-part-3/

Mostrar aviso durante la carga

Mientras que se carga la librería de Sencha Touch podemos mostrar fácilmente un texto o una imagen. Para esto podemos aprovechar el cuerpo del documento (`<body>`) que hasta ahora se encontraba vacío. Todo lo que incluyamos en esta sección se visualizará únicamente durante la carga, posteriormente será ocultado por el contenido de la aplicación. En el siguiente ejemplo se ha creado un cuadro centrado en el que aparece el texto "Cargando Aplicación...".

```
<body>
    <div style="margin:100px auto 0 auto; width:220px; font-size:16pt;">
        Cargando aplicación...
    </div>
</body>
```

Instanciar una aplicación

Para realizar nuestro primer ejemplo vamos a crear una aplicación que muestre en pantalla el mensaje "¡Hola Mundo!", para lo cual abriremos el fichero `app.js` y añadiremos el siguiente código:

```
Ext.application({
    name: 'MyApp',
    launch: function() {
        Ext.create("Ext.Panel", {
            fullscreen: true,
            html: '¡Hola Mundo!'
        });
    }
});
```

A continuación analizaremos por separado cada una de las partes de este código:

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

Con `Ext.application({ ... });` creamos una nueva instancia de Sencha Touch, es decir, este es el constructor de nuestra aplicación. Entre las llaves `{}` le pasaremos la lista de opciones de configuración para crear nuestra aplicación. En primer lugar le damos un nombre `name: 'MyApp'`, con esto automáticamente se crea una variable global llamada `MyApp` junto con los siguientes *namespaces*:

- *MyApp*
- *MyApp.views*
- *MyApp.controllers*
- *MyApp.models*
- *MyApp.stores*

Estos *namespaces* (o espacios de nombres) nos permitirán acceder a atributos de nuestra aplicación de forma sencilla, los iremos viendo en detalle más adelante.

> El nombre de la aplicación (`name: 'MyApp'`) no debe contener espacios y deberá estar formado únicamente por letras mayúsculas y minúsculas, números y algunos símbolos permitidos en la definición de variables en JavaScript.

La función `launch: function() { ... }` solo se ejecuta una vez al cargar la aplicación, y es donde deberemos de colocar el código necesario para definir y cargar nuestra aplicación. En el ejemplo propuesto creamos un panel usando el siguiente código:

```
Ext.create("Ext.Panel", {
    fullscreen: true,
    html: '¡Hola Mundo!'
});
```

Con `Ext.create("Ext.Panel", { ... });` instanciamos un panel para nuestro contenido y se lo asignamos al *viewport* de nuestra aplicación. El *viewport* es la vista principal de la aplicación, dentro de la cual iremos añadiendo todo el contenido.

Entre las llaves `{}` le pasaremos la lista de opciones de configuración que definirán el panel. Con la instrucción `fullscreen: true` le indicamos que debe ocupar toda la pantalla y con `html: 'Hola Mundo!'` el código HTML que tiene que contener.

Con esto ya hemos creado nuestra primera aplicación, un panel que ocupa toda la pantalla con el texto "¡Hola Mundo!".

> Si queremos podemos definir la función `launch` como una función independiente. Dentro del código del panel simplemente tendremos que poner el nombre de la función a llamar, por ejemplo: `launch: iniciarAplicacion` (sin poner los paréntesis de función). Y luego de forma independiente definiríamos la función `function iniciarAplicacion() { ... }` con el resto del código. Esta es una buena práctica para modularizar nuestro código.

Comprobar los resultados

Para comprobar el resultado de nuestra aplicación podemos abrirla en un navegador compatible con WebKit, como Chrome o Safari. Si abrimos el ejemplo de la sección anterior deberíamos obtener algo como:

Recordad que además del acceso directo se puede acceder usando un servidor web (en general será preferible esta 2ª opción ya que en muchas ocasiones el acceso directo puede dar problemas de permisos o de *cross origin request*). Para el servidor tenemos dos alternativas:

- Usar un servidor web propio instalado en nuestro sistema operativo, copiar nuestro proyecto dentro de la carpeta pública del servidor y acceder a través de la dirección: http://localhost/MyApp
- Usar el servidor Web que proporciona Sencha Cmd. Para activarlo tenemos que ir a la carpeta de nuestra aplicación y en un terminal ejecutar `sencha web start &`. Para acceder en este caso utilizaremos la ruta: `http://localhost:1841/`

Al usar un servidor también podemos comprobar el resultado utilizando un emulador de móvil, como en las imágenes inferiores:

Estos emuladores son parte del IDE de Xcode y del SDK de Android. Para poder utilizarlos necesitaremos tener instalados los SDKs además de tener el código en un servidor Web. Para más información consultar la sección inicial "Instalación de un servidor Web" y "Emuladores".

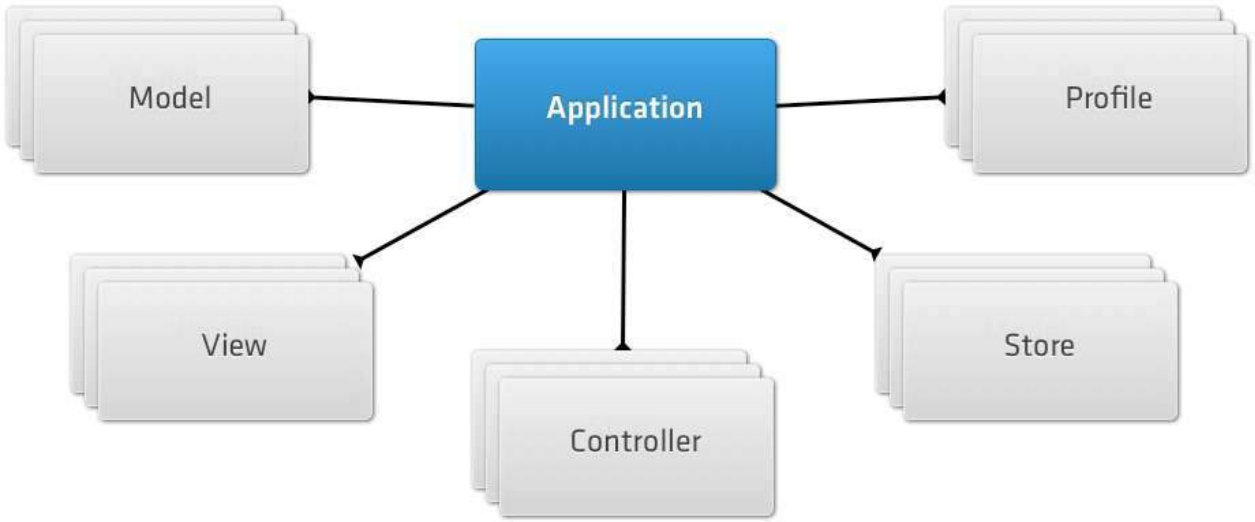
Para depurar nuestras aplicaciones, además de las herramientas para el desarrollador que incorpora el navegador podemos instalar la siguiente extensión especial para Sencha:

https://chrome.google.com/webstore/detail/app-inspector-for-sencha/pbeapidedgdpniokbedbfbaacglkceae?hl=en

Todo en un único fichero vs. MVC (patrón Modelo, Vista, Controlador)

Para facilitar la creación de aplicaciones y proporcionar a la vez una estructura de contenidos sencilla pero potente, Sencha Touch utiliza el patrón de diseño MVC (Modelo Vista Controlador).

Siguiendo este patrón una aplicación se conforma por una lista de Modelos, Vistas, Controladores, Stores (o Almacenes) y Profiles (o Perfiles), además de una serie de metadatos y recursos como el icono, imágenes, etc.



Sin embargo para aplicaciones pequeñas y sencillas no es necesario realizar esta separación del código y se puede escribir todo en un único fichero. Por lo tanto Sencha Touch permite ambos sistemas.

En este curso de introducción no vamos a tratar el modelo MVC, pero todos los conceptos y herramientas que vamos a ver sirven también para el patrón MVC.

Para más información podéis consultar la siguiente dirección:

http://docs.sencha.com/touch/2.4/core_concepts/about_applications.html

Uso de Componentes y contenedores

Qué es un componente

La mayoría de las clases visuales de Sencha Touch son componentes que heredan de `Ext.Component` lo que les da una serie de propiedades:

- Mostrar y ocultarse en cualquier momento.
- Habilitarse y deshabilitarse.
- Centrarse en la pantalla.
- Flotar sobre otros componentes.
- Acoplar y alinear otros componentes dentro del propio componente.
- Acoplarse a otros componentes.

Qué es un contenedor

Las aplicaciones se forman mediante multitud de componentes, normalmente anidados unos dentro de otros. Los contenedores **son un tipo de componente** especial que permite agrupar y organizar otros componentes dentro de si mismos. La mayoría de las aplicaciones tendrán un único contenedor (el *viewport*) que ocupará toda la pantalla, el cual contendrá una serie de componentes hijos. Por ejemplo en una aplicación de correo el contenedor del *viewport* contendrá dos componentes principales, uno para la lista de mensajes y otro para la previsualización de los correos.

Los contenedores proporcionan además la siguiente funcionalidad:

- Añadir componentes hijos e instanciarlos en tiempo de ejecución.
- Eliminar componentes hijos.
- Especificar un tipo de *layout*.

Los *layouts* determinan la disposición de los componentes hijos dentro del contenedor. En la aplicación del ejemplo de correo podríamos utilizar un *layout* horizontal del tipo *HBox* para indicar que la lista se sitúe en la parte izquierda y el panel de previsualización a la derecha ocupando el resto del espacio.

Instanciar componentes

Los componentes se crean igual que el resto de clases en Sencha Touch, utilizando el método `Ext.create`. A continuación se incluye un ejemplo:

```
var panel = Ext.create('Ext.Panel', {
    html: 'Esto es un panel'
});
```

Este trozo de código crea una instancia de un panel, le asigna un contenido HTML básico y lo almacena en la variable `panel`. Un Panel es un tipo de componente que puede contener HTML u otros items o paneles.

En el código de ejemplo simplemente se instancia el panel pero sin llegar a mostrarse, esto es porque al instanciar un elemento no se renderiza (no se hace visible) en la pantalla de forma automática. Esta característica nos permite crear componentes cuando queramos y mostrarlos cuando nos hagan falta, lo cual en general será más rápido que instanciarlos y mostrarlos inmediatamente.

Si queremos mostrar el panel que hemos creado simplemente tendríamos que añadirlo al *viewport* de la aplicación, de la forma:

```
Ext.Viewport.add(panel);
```

Configuración de un componente

Al crear un componente le podemos pasar un objecto con las opciones para configurarlo. Este objecto lo podemos crear directamente en la propia declaración del componente englobando las opciones entres llaves `{}` e indicando las opciones de configuración como pares de clave - valor. Por ejemplo:

```
// Crear la aplicación con sus opciones de configuración
Ext.application({
    name : 'MiApp',
    launch : function() { ; }
});

// Crear un panel y pasarle opciones de configuración
 var panel = Ext.create('Ext.Panel', {
    fullscreen: true,
    html: 'Esto es un panel!'
});
```

Cada componente tiene multitud de opciones de configuración, las cuales las podemos consultar en la documentación de Sencha Touch.

Al instanciar el objeto podemos pasarle tantas opciones de configuración como queramos, las cuales las podremos consultar después o modificar, por ejemplo:

```
Ext.application({
    name : 'MiApp',
    requires: ['Ext.MessageBox'],
    launch : function() {
        // Crear un panel y pasarle opciones de configuración
        var panel = Ext.create('Ext.Panel', {
            fullscreen: true,
            html: 'Esto es un panel!'
        });

        // Actualizar el HTML del panel
        panel.setHtml('Nuevo HTML!');

        // Obtener el HTML del panel y mostrarlo
        Ext.Msg.alert(panel.getHtml());
}});
```

Cada opción de configuración tiene sus métodos tipo *getter* y *setter*, se generan automáticamente a partir del nombre del atributo por lo que siempre siguen el mismo patrón. Por ejemplo, la opción de configuración `html` tiene los métodos `getHtml` y `setHtml`, o por ejemplo la opción de configuración `defaultType` tiene los métodos `getDefaultType` y `setDefaultType`.

Identificadores y referencias

Como ya hemos visto, al crear un elemento lo podemos almacenar en una variable de javascript que posteriormente podemos utilizar para hacer referencia a él. Pero Sencha Touch también permite definir un identificador (`id`) mediante el cual posteriormente podremos hacer referencia a ese elemento. La forma de definirlo, por ejemplo, para un panel es la siguiente:

```
var panel1 = Ext.create('Ext.Panel', {
  id: 'idpanel1',
  html: 'Panel 1'
});
```

Posteriormente desde otro elemento podremos referirnos a este panel de dos formas:

- Mediante la variable `panel1` , por ejemplo: `panel1.setHtml('Nuevo HTML');`
- Mediante su identificador `idpanel1` . Para esto utilizaremos el método `Ext.getCmp('idpanel1')` que nos devolverá una referencia al objeto.

Por ejemplo, para añadir este elemento a un panel contenedor tendríamos que hacer:

```
var panelPrincipal = Ext.create('Ext.Panel', {
  fullscreen: true,
  layout: 'card',
  items: [ Ext.getCmp('idpanel1') ]

  // También podríamos haber usado su nombre de variable, de la forma:
  // items: [ panel1 ]
});
```

Este identificador podemos usarlo con todos los elementos: botones, barras, etc. Es una buena práctica definirlo para todos los elementos que creamos que vayamos a referenciar posteriormente. En este documento, por simplicidad, no lo incluiremos en todos los ejemplos, solamente cuando sea necesario.

Array de *items* y el atributo *xtype*

Algunos componentes incluyen la propiedad "*items*", la cual permite especificar el array de elementos que contiene. Por ejemplo, en un panel nos permitirá indicar los elementos que este contiene o en un grupo de botones los botones a agrupar.

Si los elementos han sido creados previamente podemos usar su nombre de variable para añadirlos de la forma:

```
items: [elemento]
items: [elemento1, elemento2]
```

Pero Sencha Touch también incluye la posibilidad de crear estos elementos en línea, lo cual será mucho más rápido y nos ahorrará código. Para ello tendremos que especificar directamente las opciones del objeto a crear entre llaves `{}` , de la forma:

```
items: [{...}, {...}]
// O si solo queremos crear un elemento:
items: {}
```

Al crear un elemento en línea, además de especificar el resto de sus propiedades, también tendremos que definir su tipo *xtype* (o tipo de objeto), de la forma:

```
items: { xtype: 'toolbar', docked: 'top' }
```

El atributo *xtype* facilita la creación de componentes y sin la necesidad de utilizar su nombre de clase completo. Es especialmente útil para crear componentes dentro de una clase contenedora. A continuación se muestra un ejemplo más completo de uso:

```
Ext.application({
    name : 'MiApp',
    launch : function() {
        Ext.create('Ext.Container', {
            fullscreen: true,
            layout: 'fit',
            items: [
                {
                    xtype: 'panel',
                    html: 'Este panel se ha creado mediante xtype'
                },
                {
                    xtype: 'toolbar',
                    title: 'Mi App',
                    docked: 'top'
                }
            ]
        });
}});
```

Listado de todos los *xtypes* disponibles

A continuación se incluye un listado de todos los *xtypes* disponibles en Sencha Touch:

Componentes generales:

xtype	Class
actionsheet	Ext.ActionSheet
audio	Ext.Audio
button	Ext.Button
component	Ext.Component
container	Ext.Container
image	Ext.Img
label	Ext.Label
loadmask	Ext.LoadMask
map	Ext.Map
mask	Ext.Mask
media	Ext.Media
panel	Ext.Panel
segmentedbutton	Ext.SegmentedButton

sheet	Ext.Sheet
spacer	Ext.Spacer
title	Ext.Title
titlebar	Ext.TitleBar
toolbar	Ext.Toolbar
video	Ext.Video
carousel	Ext.carousel.Carousel
carouselindicator	Ext.carousel.Indicator
navigationview	Ext.navigation.View
datepicker	Ext.picker.Date
picker	Ext.picker.Picker
pickerslot	Ext.picker.Slot
slider	Ext.slider.Slider
thumb	Ext.slider.Thumb
tabbar	Ext.tab.Bar
tabpanel	Ext.tab.Panel
tab	Ext.tab.Tab
viewport	Ext.viewport.Default

Componentes tipo *DataView*:

xtype	Class
dataview	Ext.dataview.DataView
list	Ext.dataview.List
listitemheader	Ext.dataview.ListItemHeader
nestedlist	Ext.dataview.NestedList
dataitem	Ext.dataview.component.DataItem

Componentes para formulario:

xtype	Class
checkboxfield	Ext.field.Checkbox
datepickerfield	Ext.field.DatePicker
emailfield	Ext.field.Email
field	Ext.field.Field
hiddenfield	Ext.field.Hidden
input	Ext.field.Input
numberfield	Ext.field.Number
passwordfield	Ext.field.Password
radiofield	Ext.field.Radio
searchfield	Ext.field.Search
selectfield	Ext.field.Select
sliderfield	Ext.field.Slider
spinnerfield	Ext.field.Spinner
textfield	Ext.field.Text
textareafield	Ext.field.TextArea
textareainput	Ext.field.TextAreaInput
togglefield	Ext.field.Toggle
urlfield	Ext.field.Url
fieldset	Ext.form.FieldSet
formpanel	Ext.form.Panel

Añadir componentes a contenedores

Una vez creado un panel o contenedor también podemos añadirle más elementos en tiempo de ejecución mediante su método `add`. En el siguiente ejemplo se crea un panel que contiene un único panel y posteriormente se le añade otro panel de forma dinámica:

```
Ext.application({
    name : 'MiApp',
    launch : function() {
        var secondPanel = Ext.create('Ext.Panel', {
            html: 'Segundo panel'
        });

        // Este sería el panel principal
        var mainPanel = Ext.create('Ext.Panel', {
            fullscreen: true,
            layout: 'hbox',
            defaults: {
                flex: 1
            },
            items: {
                html: 'Primer panel',
                style: 'background-color: #5E99CC;'
            }
        });

        // Añadimos otro panel al contenedor principal
        mainPanel.add(secondPanel);
}});
```

En este caso le asignamos el layout *hbox* al panel principal contenedor para que los paneles hijos se vayan añadiendo de forma horizontal. Además se utilizan un par de propiedades nuevas: `style`, la cual nos permite escribir código CSS para aplicar estilos a un componente y `defaults`, que nos permite establecer valores por defecto que se aplicarán a todos los componentes que contenta. En este caso al primer panel se le asignará un `flex` de 1, por lo que ocupará todo el ancho, pero al añadir el segundo panel también se le asignará un `flex` de 1 por lo que el ancho se repartirá y cada panel ocupará la mitad del espacio disponible (en la sección de *layouts* se tratará este tema más en profundidad).

Eliminar componentes de un panel

Para eliminar elementos de un contenedor utilizamos su método `remove` y la variable que define el item. Por ejemplo para eliminar el panel que hemos añadido en el ejemplo anterior tendríamos que hacer:

```
mainPanel.remove(secondPanel);
```

> Este método elimina el componente del contenedor, no lo destruye ni libera la memoria ni los eventos asociados.

Mostrar y ocultar componentes

Los elementos también se pueden mostrar u ocultar simplemente llamando a sus métodos `hide()` o `show()`. Continuando con el ejemplo anterior, para mostrar u ocultar el panel principal haríamos:

```
mainPanel.show();

mainPanel.hide();
```

Eventos

Todos los componentes de Sencha Touch lanzan eventos ante determinados cambios, estos eventos pueden ser escuchados y realizar una acción cuando son activados. Por ejemplo, al escribir en un campo de texto este lanza su evento `change`, por lo que podríamos escuchar a dicho evento usando un *listener* como se muestra en el siguiente ejemplo:

```
Ext.create('Ext.form.Text', {
    label: 'Name',
    listeners: {
        change: function(field, newValue, oldValue) {
            // El contenido ha cambiado
        }
    }
});
```

Los componentes de Sencha Touch lanzan multitud de eventos facilitando el control de la aplicación y la programación por eventos. A estos eventos nos podemos suscribir también de forma dinámica una vez que el componente se ha creado. En la documentación de cada clase se puede encontrar un listado con los eventos que lanza y como podemos utilizarlos.

Destruir componentes

Cuando no se va a necesitar más un elemento se recomienda eliminarlo completamente para ahorrar memoria. Hemos de tener en cuenta que en los dispositivos móviles la memoria es un recurso escaso y si nuestra aplicación es muy grande puede llegar a ralentizar el móvil. Por este motivo se ha introducido el método `destroy` que elimina el componente que lo llame:

```
mainPanel.destroy();
```

Este comando eliminaría el *mainPanel* del DOM y además eliminaría todos los *listeners* que estuvieran escuchando a sus eventos. Hemos de tener cuidado ya que también se eliminará todo el contenido del elemento, por ejemplo si es un contenedor se eliminarían sus paneles hijos.

Layouts

Los *layouts* se utilizan para especificar las dimensiones y posicionamiento de los componentes en una aplicación. Por ejemplo, en un aplicación de correo en general se colocarán dos paneles una a continuación del otro en horizontal, el de la izquierda para lista de mensajes ocupando un tercio del ancho y el de la derecha para la previsualización ocupando el resto del espacio.

A continuación se analizarán los diferentes tipos de *layouts* que incorpora Sencha Touch:

- *hbox*
- *vbox*
- *card*
- *fit*

Además también veremos los conceptos de *docking*, *pack* y *align*.

Layout tipo *HBox*

Si queremos que los elementos en un contenedor se dispongan de forma horizontal podemos utilizar el *layout* tipo `hbox`.

Además, para especificar el espacio que han de ocupar los componentes dentro del contenedor podemos utilizar la propiedad `flex`, la cual indica la proporción de espacio que ocupará un componente. Al indicar el espaciado de una serie de componentes con `flex` no es necesario que sumen 100, sino que se sumará el total de las cantidades y ese será el 100% del espacio. Por ejemplo, para conseguir una columna que ocupe 1/3 y otra de 2/3 especificaríamos los siguientes valores para la propiedad *flex*:

Para conseguir una disposición como la de la imagen tendríamos que indicar el *layout* tipo *hbox* al contenedor padre y establecer el atributo *flex* de cada hijo de la forma:

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'hbox',
    items: [
        {
            xtype: 'panel',
            html: 'Columna que ocupa 1/3 del ancho.',
            flex: 1,
            style: 'background-color: #5E99CC;'
        },
        {
            xtype: 'panel',
            html: 'Columna que ocupa 2/3 del ancho.',
            flex: 2,
            style: 'background-color: #759E60;'
        }
    ]
});
```

Layout tipo *VBox*

El layout *vbox* es similar a *hbox* pero creando una disposición vertical en lugar de horizontal. Lo podemos visualizar como un conjunto de cajas apiladas de la siguiente forma:

El código para crear una pantalla de este tipo sería idéntico al utilizado en el ejemplo anterior pero cambiando el tipo de *layout hbox* por *vbox*:

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'vbox',
    items: [
        {
            xtype: 'panel',
            html: 'Fila que ocupa 1/3 del alto.',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'Fila que ocupa 2/3 del alto.',
            flex: 2
        }
    ]
});
```

Layout tipo *Card*

El *layout* tipo *Card* permite asignar muchos componentes o contenedores al mismo *layout* pero en el que solamente se mostrará uno a la vez. Como si fuera una baraja de cartas, los componentes se apilarán y solamente se mostrará uno de ellos ocupando toda la pantalla y ocultando al resto.

En esta imagen el cuadro gris es el contenedor y la caja azul dentro de él es el elemento que actualmente se está mostrando. El resto de elementos que aparecen al lado están ocultos, pero se pueden intercambiar en cualquier momento para mostrarse.

A continuación se incluye un ejemplo de un panel con un *layout* tipo *card* que contiene cuatro tarjetas:

```
var panel = Ext.create('Ext.Panel', {
    layout: 'card',
    items: [
        {
            html: "Primer item"
        },
        {
            html: "Segundo item"
        },
        {
            html: "Tercer item"
        },
        {
            html: "Cuarto item"
        }
    ]
});

panel.setActiveItem(1);
```

Por defecto se muestra la primera tarjeta, pero mediante la llamada a `panel.setActiveItem(1);` indicamos que se muestre la 2ª tarjeta (el número de elemento empieza en cero, por lo que el 1 se refiere a la 2ª posición).

Al añadir las *cartas* del *layout* las podemos crear directamente en el array de *items*, como en el ejemplo, o añadirlas posteriormente con el método `add` del panel (como ya se vio en una sección anterior).

Layout tipo *Fit*

El *layout* tipo *fit* es uno de los más sencillos, simplemente hace que sus componentes hijos ocupen todo el tamaño disponible del contenedor.

Por ejemplo, si tenemos un contenedor de 200px de ancho por 200px de alto y le añadimos un hijo y el *layout* tipo *fit*, el componente añadido será expandido para ocupar el mismo tamaño que el padre:

```
var panel = Ext.create('Ext.Panel', {
    width: 200,
    height: 200,
    layout: 'fit',
    items: {
        xtype: 'panel',
        html: 'Panel del mismo tamaño que el padre.'
    }
});
Ext.Viewport.add(panel);
```

> Si añadimos varios elementos a un contenedor de este tipo solamente será visible el primero, ya que al expandirse para ocupar todo el espacio ocultará al resto. Para mostrar otros elementos en el panel podemos utilizar otros *layouts* (hbox, vbox o card) u ocultar el elemento visible para dejar espacio.

Docking o acoplamiento

Todos los *layouts* tienen la capacidad de acoplar elementos "adicionales" de forma fija en cualquiera de sus laterales. Al acoplar un elemento el resto de contenido del *layout* será redimensionado para adaptarse. Las posiciones en las que se puede acoplar un elemento son: *top*, *right*, *bottom*, o *left*.

Es importante notar que los elementos acoplados son "adicionales" al contenido del *layout*, es decir, si por ejemplo acoplamos un elemento a un contenedor con un *layout* tipo *hbox*, el elemento acoplado no seguirá la alineación horizontal, sino que se pondrá en la posición que se especifique con `docked` de entre la lista de posiciones permitidas.

En la imagen superior tenemos un *layout* tipo *hbox* con dos columnas y un elemento acoplado en la parte superior. A continuación se incluye el código para crear una disposición de este tipo:

```
Ext.create('Ext.Container', {
    fullscreen: true,
    layout: 'hbox',
    items: [
        {
            docked: 'top',
            xtype: 'panel',
            height: 20,
            html: 'Elemento acoplado en la parte superior.'
        },
        {
            xtype: 'panel',
            html: 'Columna izquierda.',
            flex: 1
        },
        {
            xtype: 'panel',
            html: 'Columna derecha.',
            flex: 2
        }
    ]
});
```

En el ejemplo se ha utilizado la característica `docked` para acoplar un panel, pero es mucho más común su utilización con herramientas tipo `toolbar` o `titlebar`.

Podemos acoplar tantos elementos como queramos, los cuales se irán añadiendo en el mismo orden en el que se asignen.

Pack y *Align*

Las características *pack* y *align* del *layout* sirven para controlar la alineación de los elementos hijos dentro de un contenedor:

- *Pack*: es la alineación en el mismo eje de alineación del *layout* utilizado. Por ejemplo, en un *layout* tipo *hbox* sería el horizontal y en uno tipo *vbox* el vertical. Se pueden asignar tres posibles valores: *start*, *center* y *end*.
- *Align*: es la alineación en el eje perpendicular al de la alineación del *layout* utilizado. Por ejemplo, en un *layout* del tipo *hbox* será la vertical y en uno del tipo *vbox* la horizontal. Puede tener cuatro posibles valores: *start*, *center*, *end* y *stretch*.

A continuación se incluye un ejemplo de su utilización:

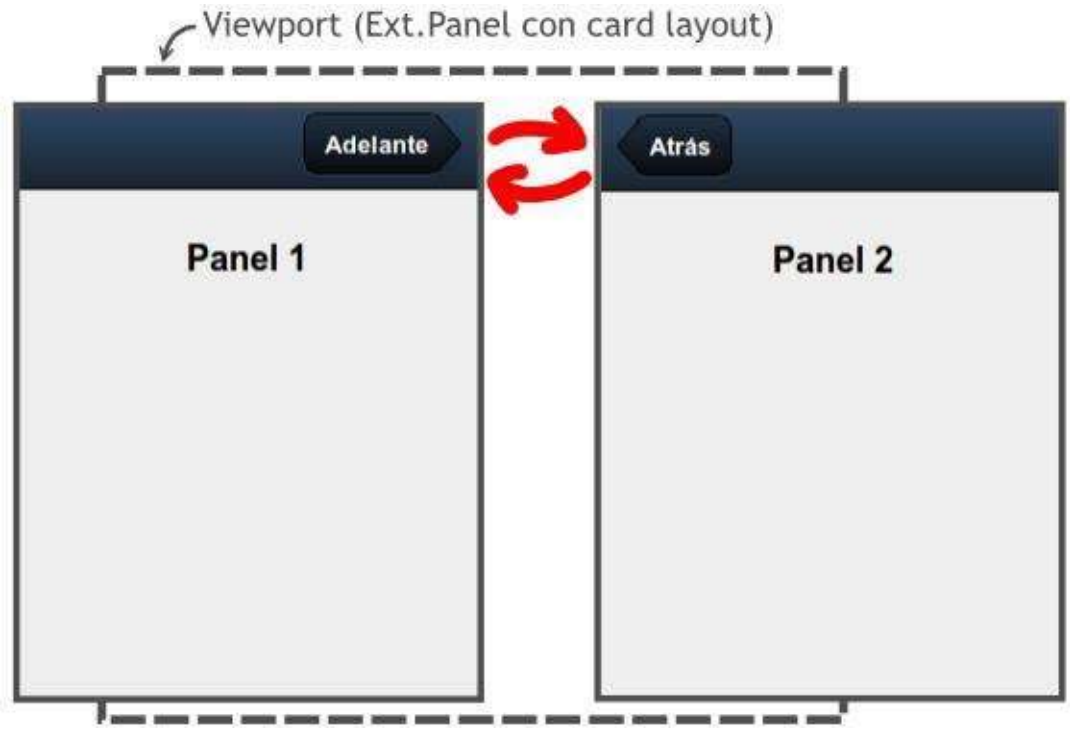
```
Ext.create('Ext.Panel', {
    fullscreen: true,
    layout: {
        type: 'hbox',
        align: 'center',
        pack: 'center'
    },
    items: {
        xtype: 'panel',
        html: 'Contenido centrado.'
    }
});
```

Transiciones de cambio de vista

En este apartado vamos a ver como cambiar entre diferentes paneles. Nuestra aplicación tendrá una vista principal y al apretar sobre algún botón, cambiaremos a una vista o panel diferente. Para hacer esto lo más importante es utilizar un panel base no visible que guardaremos en una variable (en este caso en `panelPrincipal`), el cual contendrá como **"items"** los paneles entre los que queremos cambiar. Además tenemos que establecer el layout del panel base a **"layout: 'card'"**, quedando el código de nuestro panel de la forma:

```
var panelPrincipal = Ext.create('Ext.Panel', {
    fullscreen: true,
    layout: 'card',
    items: [
        panel1,  // Panel visible
        panel2
    ]
});
```

El panel que se verá al principio es el primero que se añade a la lista de items, quedando el otro (o los otros) ocultos. En la siguiente imagen se puede ver un esquema del intercambio de paneles (en nuestro ejemplo con dos paneles). El panel asignado al "viewport" (o contenedor base) queda invisible por detrás y mediante un botón podemos pasar de un panel a otro:



A continuación se incluye el código para el "panel1". Un simple panel con una barra de herramientas en la parte superior que contiene un botón. Lo más importante aquí es la función **"handler"** del botón, en la cual llamamos a `panelPrincipal.setActiveItem` (función explicada a continuación) para cambiar al "panel2". El código para el "panel2" sería exactamente igual, pero cambiando la variable, el html y la función *handler*.

```
var panel1 = Ext.create('Ext.Panel', {
    fullscreen: true,
    layout: 'fit',
    html: 'Panel 1',
    items: [{
        xtype: 'toolbar',
        docked: 'top',
        items: [{
            ui: 'forward',
            text: 'Adelante',
            handler: function() {
                panelPrincipal.setActiveItem( panel2 );
            }
        }] // end items
    }] // end items
});
```

Es importante destacar que en este ejemplo hemos hecho referencia al panel base (`panelPrincipal`) y al panel a cambiar (`panel2`) mediante su nombre de variable, pero también podríamos haberle asignado un identificador y haberlo obtenido con `Ext.getCmp(id)` .

Método *setActiveItem*

La función `setActiveItem( Object/Number item )` permite cambiar entre el panel activo o visible por otro panel indicado. La forma de indicar el panel puede ser mediante su número en el array de paneles del contenedor (empezando por cero) o mediante el propio objeto a mostrar.

Método *animateActiveItem*

La función `animateActiveItem( Object/Number item, atributos_animación )` funciona igual que `setActiveItem` , permite cambiar el panel actual por otro panel indicado mediante su posición en el array de items (empezando por cero) o mediante una referencia al propio objeto. Además, esta función permite definir la animación que se realizará al intercambiar los paneles. Los tipos de animaciones que podemos utilizar son:

- *fade*: difumina el panel actual, fundiéndolo con el panel de destino, hasta completar la transición.
- *pop*: realiza una especie de animación 3D. Escala el panel actual minimizándolo hasta ocultarlo, mientras que aumenta el tamaño del panel a visualizar.
- *slide*: realiza un desplazamiento para intercambiar un panel por otro, podemos indicar una dirección: *left, right, up, down* (por ejemplo: *direction: 'left'*).
- *flip*: realiza una animación 3D para intercambiar los paneles.
- *cube*: realiza una animación 3D para intercambiar los paneles.
- *cover*: realiza una animación 3D para intercambiar los paneles.
- *reveal*: realiza una animación 3D para intercambiar los paneles.
- *scroll*: realiza una animación 3D para intercambiar los paneles.

Para todos ellos podemos definir una duración en milisegundos (por ejemplo *"duration: 2000"*).

Una posible transición que podemos definir es:

```
panelPrincipal.animateActiveItem(
    panel2,
    {type: 'slide', direction: 'up', duration: 2000});
```

> Nota: Las animaciones "*flip*", "*cube*", "*cover*", "*reveal*" y "*scroll*" no funcionan correctamente en versiones antiguas de algunos navegadores, como en Android. En estos casos serán sustituidas por la animación por defecto.

Componentes

En esta sección se introducen los principales componentes que incorpora Sencha Touch y que podemos utilizar para la elaboración de las pantallas de una aplicación. Algunos de estos componentes son:

- *Toolbars*
- Botones
- *TabPanel*
- *Carousel*
- Diálogos
- Formularios

Toolbars

Hasta ahora hemos visto como utilizar paneles y contenedores (`Ext.Panel` y `Ext.Container`), en esta sección vamos a ver como añadir barras de herramientas dentro de estos elementos.

Para añadir barras de herramientas a un panel tenemos dos opciones, igual que para otro tipo de componentes, estas son:

- Crear la barra de herramientas de forma separada usando el constructor " `var toolbar = Ext.create('Ext.Toolbar', { ... });` " y posteriormente añadirla al panel en su atributo " `items` " usando su nombre de variable.
- Crear la barra de herramientas directamente en el atributo " `items` " del panel. Para esto definiremos el tipo de componente usando el `xtype: 'toolbar'` .

En ambos casos, dentro del constructor podemos usar los siguientes atributos:

- **id: 'identificador'**: Atributo opcional para indicar el identificador de la barra.
- **docked: 'top'** o **docked: 'bottom'**: indica que la barra se coloque en la parte superior o en la parte inferior del panel respectivamente.
- **title: 'texto'**: indica el texto que se colocará en el centro de la barra.
- **ui: valor**: atributo opcional que cambia la apariencia de la barra. Por defecto toma el valor "*dark*", pero también podemos usar "*light*" que aplicará unos colores más claros.

En el siguiente ejemplo se crean dos barras de herramientas y se añaden a un panel. La primera se crea de forma separada y después se añaden al panel usando su nombre de variable. La segunda barra se crea en línea usando su *xtype*:

```
var topToolbar = Ext.create('Ext.Toolbar', {
  docked: 'top',
  title: 'Top Toolbar'
});

var panel = Ext.create('Ext.Panel', {
  fullscreen: true,
  layout: 'fit',
  html: 'Contenido central',
  items: [
    topToolbar,
    {
        xtype: 'toolbar',
        dock: 'bottom',
        title: 'Bottom Toolbar'
    }
  ]
});
```

Con lo que obtendríamos un resultado similar a:

Botones

Los botones se añaden a las barras de herramientas (`Toolbar`) u otro tipo de componentes mediante su propiedad `items` . La forma de construir un botón, igual que para otros componentes, son dos:

- Mediante su constructor creamos el botón y lo asignamos a una variable: `var button = Ext.create('Ext.Button', {...});` . Posteriormente podemos asignar dicha variable al atributo `items` de algún contenedor.
- Definir el botón directamente dentro del atributo `items` de algún contenedor mediante el `xtype: 'button'` .

Además tenemos una serie de propiedades que podemos configurar:

- *text*: texto del botón.
- *ui*: tipo o apariencia del botón.
- *iconCls*: icono del botón.

En el siguiente ejemplo se crea una barra de herramientas con dos botones, uno creado de forma *inline* y otro de forma separada:

```
var button1 = Ext.create('Ext.Button', {
  text: 'Mi botón 1'
});

var topToolbar = Ext.create('Ext.Toolbar', {
  dock: 'top',
  title: 'mi barra',
  items: [
        button1,
        {
            xtype: 'button',
            ui: 'action',
            text: 'Mi botón 2'
        }
  ]
});
```

Podemos usar siete tipos **ui** predefinidos de botones, estos son:

- ui: 'normal'
- ui: 'back'
- ui: 'forward'
- ui: 'round'

- ui: 'action'
- ui: 'confirm'
- ui: 'decline'

Además podemos usar los modificadores **"-small"** y **"-round"** sobre los tipos de botón "action", "confirm" y "decline" para obtener botones más pequeños o redondeados:

Si no indicamos un tipo (`ui`) por defecto nos aparecerá el botón tipo "normal".

Si queremos variar el **ancho** de un botón podemos utilizar la propiedad " `width: '200px'` " en píxeles o " `width: '95%'` " indicando porcentajes.

Iconos

También podemos usar algunos iconos predefinidos indicando el nombre del icono mediante la propiedad **"iconCls: 'nombre'"**, de la forma:

```
var button = Ext.create('Ext.Button', {
  iconCls: 'action'
});
```

Los iconos que podemos utilizar son:

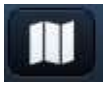
 action  add  arrow_down

 arrow_left  arrow_right  arrow_up

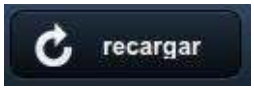
 compose  delete  organize

 refresh  reply  search

 settings  star  trash

 maps  locate  home

Si además usamos la propiedad **"text: 'texto'"** al definir el botón, el texto aparecerá a la derecha del icono:



Opcionalmente podemos cambiar el color de estos botones con iconos mediante los tipos (ui) de 'action', 'decline' o 'confirm', obteniendo:

Imágenes externas

Si queremos usar una imagen externa tenemos que aplicar al botón un estilo CSS. El botón lo definimos de forma normal, pero utilizamos su propiedad `cls` para indicar el nombre del estilo:

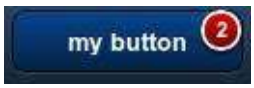
```
items: [ { xtype: 'button', ui: 'normal', cls: 'btnAyuda' } ]
```

El estilo "btnAyuda" lo tendremos que definir indicando la imagen de fondo a usar junto con el ancho y el alto del botón. El tamaño de la imagen que usemos deberá de coincidir con el tamaño aplicado al botón para que no se vea recortado. El tamaño habitual de un botón es de 45x35 píxeles. Además es imprescindible añadir en el CSS la propiedad `!important` al cargar la imagen de fondo. Esto es debido a que Sencha Touch sobrescribe algunos estilos y tenemos que aplicar esta propiedad para que prevalezca el nuestro:

```
.btnAyuda {
    background: url(resources/imgs/ayuda.png) !important;
    width: 45px;
    height: 35px;
}
```

Badges

De forma sencilla podemos añadir una insignia distintiva a los botones para destacar alguna información. Para esto utilizamos la propiedad **"badgeText: '2'"**, que daría como resultado:



Alineaciones

Por defecto los botones aparecerán alineados a la izquierda del contenedor. Para crear otras alineaciones utilizaremos un espaciador **"{ xtype: 'spacer' }"**. Este espaciador es un componente no visible que ocupará todo el espacio libre disponible, por lo tanto si lo asignamos a la izquierda de un botón lo "empujará" a la derecha, y si añadimos dos espaciadores, uno a cada lado de un botón, lo centrará. En el siguiente código podemos ver diferentes ejemplos de alineaciones:

```
// Alineación derecha
items: [
  { xtype: 'spacer' },
  { xtype: 'button', ui: 'normal', text: 'Botón' }
]
// Alineación centrada
items: [
  { xtype: 'spacer' },
  { xtype: 'button', ui: 'normal', text: 'Botón' },
  { xtype: 'spacer' }
]
```

Acciones

Para añadir acciones a los botones tenemos que definir su propiedad "**handler**", a la cual le asignaremos una función. Esta función la podemos definir en línea, de la forma `handler: function () { ... }`, o creando una función independiente para separar mejor el código, como en el ejemplo:

```
function botonPresionado(btn, evt) {
    alert("Presionado " + btn.text);
}

var topToolbar = Ext.create('Ext.Toolbar', {
      items: [
        {
            xtype: 'button',
            ui: 'normal',
            text: 'Botón 1',
            handler: botonPresionado
        },
        {
            xtype: 'button',
            ui: 'action',
            text: 'Botón 2',
            handler: function(btn, evt) {
                alert("Presionado " + btn.text);
            }
        }]
});
```

TabPanel

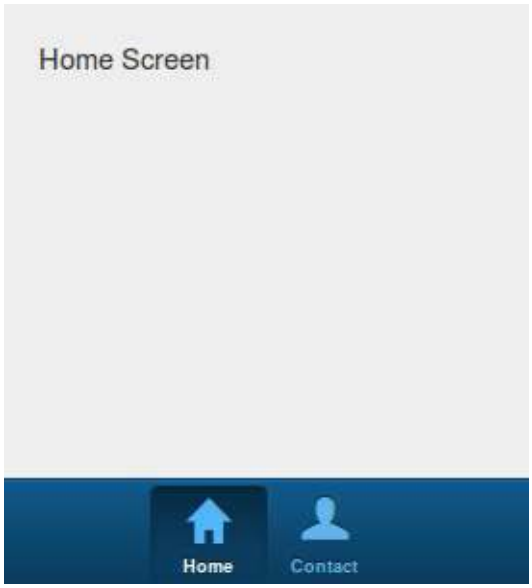
Un *TabPanel* es similar a un panel con un *layout* tipo *card*, pero al que se ha añadido además la funcionalidad de mostrar automáticamente una barra de herramientas con la lista de *tabs* que contiene y que nos permitirá cambiar entre ellos.

Esta barra de *tabs* se puede posicionar en la parte superior (*top*) o en la inferior (*bottom*) del panel, y opcionalmente se le puede asignar un título e iconos a los botones.

En el siguiente ejemplo se muestra un *TabPanel* con los *tabs* en la parte inferior, además se le han añadido iconos:

```
Ext.application({
    name: 'MiApp',
    launch: function() {
        Ext.create('Ext.TabPanel', {
            fullscreen: true,
            tabBarPosition: 'bottom',
            defaults: {
                styleHtmlContent: true
            },
            items: [
                {
                    title: 'Home',
                    iconCls: 'home',
                    html: 'Home Screen'
                },
                {
                    title: 'Contact',
                    iconCls: 'user',
                    html: 'Contact Screen'
                }
            ]
        });
    }
});
```

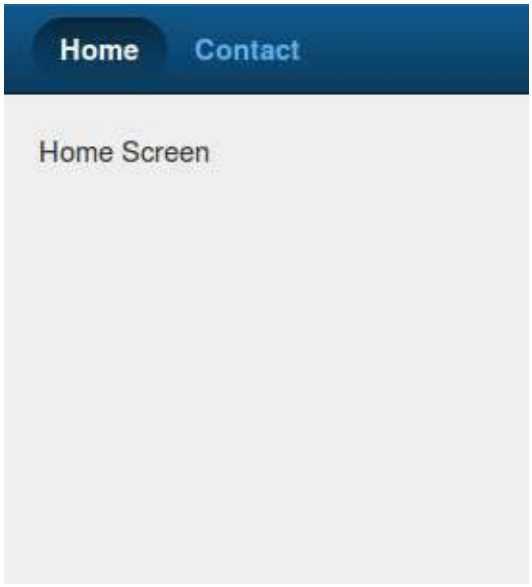
El cual se mostraría como:



Como se puede observar la barra de *tabs* automáticamente se configura con su icono y título, además al pulsar sobre ellos se realizaría una animación para cambiar de panel. Si no se especifica el atributo `tabBarPosition` la barra por defecto se situará en la parte superior, a continuación se incluye un ejemplo:

```
Ext.application({
    name: 'MiApp',
    launch: function() {
        Ext.create('Ext.TabPanel', {
            fullscreen: true,
            defaults: {
                styleHtmlContent: true
            },
            items: [
                {
                    title: 'Home',
                    html: 'Home Screen'
                },
                {
                    title: 'Contact',
                    html: 'Contact Screen'
                }
            ]
        });
    }
});
```

El cual se mostraría como:



Animaciones en un *TabPanel*

Los *TabPanel* realizan el cambio de panel automáticamente (no tenemos que escribir código para esto) y tienen asignada la animación tipo *slide* por defecto. Si queremos podemos cambiarla por cualquier otra (podemos usar las mismas que en un *card layout*) de la forma:

```
Ext.application({
    name: 'MiApp',
    launch: function() {
        Ext.create('Ext.TabPanel', {
            fullscreen: true,
            defaults: {
                styleHtmlContent: true
            },
            layout: {
                type: 'card',
                animation: {
                    type: 'fade'
                }
            },
            items: [
                {
                    title: 'Home',
                    html: 'Home Screen'
                },
                {
                    title: 'Contact',
                    html: 'Contact Screen'
                }
            ]
        });
    }
});
```

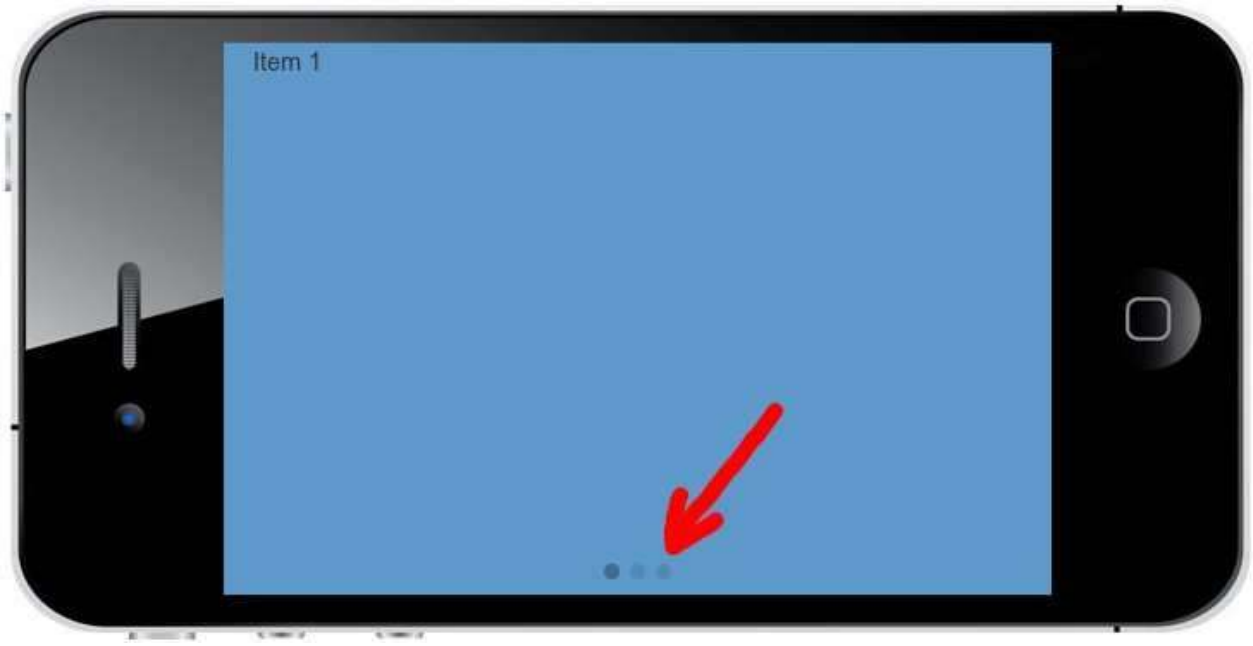
Carousel

El Carousel es un contenedor de paneles que nos permite cambiar entre ellos simplemente arrastrando el dedo. Solo se muestra un panel en cada momento junto con un pequeño indicador con puntos que referencia el número de paneles disponibles.

Es muy sencillo configurarlo, en su sección `items` tenemos que definir cada uno de los paneles. Si queremos que se utilicen los estilos HTML básicos tenemos que activar la opción `defaults: { styleHtmlContent: true }`, como en el siguiente ejemplo:

```
var panelCarousel = Ext.create('Ext.Carousel', {
    fullscreen: true,
    defaults: {
        styleHtmlContent: true
    },
    items: [
        {
            html : 'Item 1',
            style: 'background-color: #5E99CC'
        },
        {
            html : 'Item 2',
            style: 'background-color: #759E60'
        },
        {
            html : 'Item 3'
        }
    ]
});
```

Con lo que obtendríamos un resultado como el siguiente:



Dentro de los items de un carousel podemos añadir cualquier tipo de componente de entre los disponibles en Sencha Touch.

Una opción interesante de configuración es la orientación del panel, que básicamente lo que hace es cambiar la posición de los puntos y la dirección de movimiento de los paneles. Para configurarlo usamos la propiedad `direction: 'horizontal'` (por defecto) o `direction: 'vertical'`.

Diálogos mediante *MessageBox*

Esta clase nos permite generar mensajes emergentes de tres tipos: alertas, confirmación y de campo de texto.

Pero para que funcionen los avisos tenemos que precargar la clase `Ext.MessageBox` en el atributo `requires` de la aplicación (o de la clase o componente que lo utilice):

```
Ext.application({
    name: 'MiApp',
    requires: ['Ext.MessageBox'],
    launch: function() {
        ...
    }
});
```

Esto es necesario ya que de otra forma no funcionarán los avisos. En general se recomienda especificar en cada contenedor las clases que tiene que cargar, en general el propio sistema las carga en el momento en que las necesita, pero en algunas ocasiones (como en esta) si no se precargan pueden aparecer errores.

A continuación veremos los diferentes tipos de aviso que podemos usar.

Alertas

Muestra un mensaje de aviso con un solo botón OK, como podemos ver en la imagen siguiente:

Para crear una ventana de aviso usamos el constructor **`Ext.Msg.alert(title,msg,function)`**, el primer parámetro es el título de la ventana, el segundo es el mensaje de aviso que aparecerá en el centro de la ventana, y el último es la función *callback* que se llamará una vez cerrada la ventana. Por ejemplo:

```
Ext.Msg.alert('Titulo', 'Mensaje de aviso', Ext.emptyFn);
```

En este caso usamos la función vacía `Ext.emptyFn` para que no se ejecute nada. En su lugar podríamos haber puesto directamente el nombre de una función a llamar.

Confirmación

Este mensaje de aviso nos da la opción de aceptar o rechazar, como podemos ver en la siguiente imagen:

En este caso utilizamos el constructor **`Ext.Msg.confirm(title,msg,function)`**, los parámetros serán los mismos: título, mensaje y función. En este caso sí que nos interesa indicar el nombre de una función para poder comprobar si se ha pulsado el botón OK. La función *callback* recibirá un único parámetro cuyo valor será el texto del botón pulsado.

```
function myFunction(btn)
{
  if( btn == "yes" )
    Ext.Msg.alert( "¡Ha pulsado sí! :D" );
  else
    Ext.Msg.alert( "Ha pulsado no :(" );
}

Ext.Msg.confirm( "Confirmation", "Are you sure you want to do that?", myFunction );
```

Prompt, solicitar datos

El diálogo tipo *prompt* sirve para solicitar un dato al usuario, consiste en una pequeña ventana con un campo de texto que se puede aceptar o rechazar:

Utilizamos el constructor `Ext.Msg.prompt(title,msg,function)` con los parámetros: título, mensaje y función. En este caso la función *callback* recibirá dos parámetros: el botón pulsado y el texto introducido.

```
function myFunction(btn, text)
{
  Ext.Msg.alert( btn + ' ' + text );
}

Ext.Msg.prompt('Name', 'Please enter your name:', myFunction );
```

Formularios

Para crear formularios utilizamos el constructor `Ext.create('Ext.form.Panel', { ... });` , el cual se comporta exactamente igual que un panel, pero permitiendo añadir fácilmente en el array " `items` " campos de tipo formulario. En el siguiente ejemplo se crea un formulario que contiene un campo de texto y un área de texto:

```
Ext.application({
    name: 'MiApp',
    launch: function() {
        Ext.create('Ext.form.Panel', {
            fullscreen: true,
            items: [
                {
                    xtype: 'textfield',
                    name: 'title',
                    label: 'Title',
                    required: true
                },
                {
                    xtype: 'textareafield',
                    name: 'narrative',
                    label: 'Narrative'
                }
            ]
        });
    }
});
```

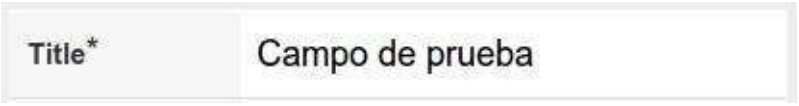
Tipos de campos

Para **todos los campos** podemos especificar un nombre " `name` ", una etiqueta " `label` " y si es requerido " `required: true` " (esta propiedad solo es visual, añade un asterisco (`*`) en el nombre del campo, pero **no realiza ninguna validación**).

El nombre (`name`) se utiliza para cargar y enviar los datos del formulario (como veremos más adelante), y la etiqueta (`label`) se mostrará visualmente en la parte izquierda de cada campo. El valor de todos los campos se encuentra en su atributo " `value` ", el cual también podemos utilizarlo para especificar un valor inicial.

Los principales tipos de campos que podemos utilizar son los siguientes (indicados según su nombre " `xtype` " en negrita):

- **textfield**: campo de texto.



- **textareafield**: área de texto.

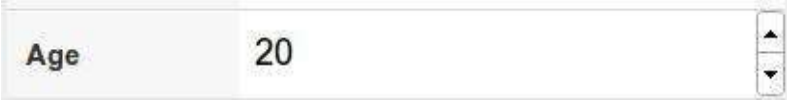
- **passwordfield**: campo de texto para introducir contraseñas. El código es igual que para un *textfield* pero cambiando el valor de " `xtype: 'passwordfield'` ":

- **urlfield**: campo de texto para direcciones Web, incluye validación de URL correcta:

- **emailfield**: campo de texto para introducir e-mails, incluye validación automática:

- **togglefield**: permite seleccionar entre dos valores (0 ó 1). Por defecto se encuentra desactivado, para activarlo por defecto tenemos que añadir " `value:1` " a la definición del campo:

- **numberfield**: campo numérico, permite introducir el número manualmente o mediante las flechas laterales. Inicialmente no contiene ningún valor, pero podemos definir un valor inicial mediante la propiedad " `value: 20` ":



- **spinnerfield**: campo numérico, permite introducir el número manualmente o mediante los botones laterales. Inicialmente su valor es 0. Podemos definir un valor inicial mediante la propiedad " `value: 20` ". También podemos definir un valor mínimo " `minValue: 0` ", un valor máximo " `maxValue: 100` ", el incremento " `incrementValue: 2` " y si se permiten ciclos " `cycle: true` ".

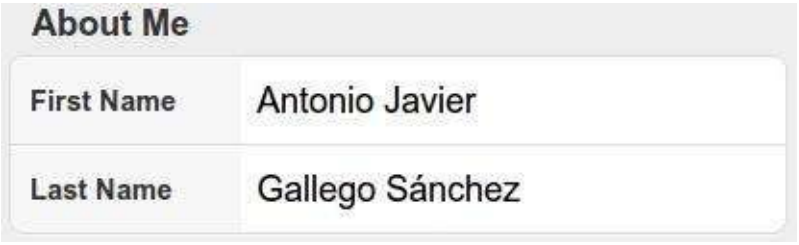
- **sliderfield**: campo numérico modificable mediante una barra o slider. Inicialmente su valor es 0. Podemos definir un valor inicial mediante la propiedad " `value: 50` ". También podemos definir un valor mínimo " `minValue: 0` ", un valor máximo " `maxValue: 100` " y el incremento " `incrementValue: 2` ".

- **datepickerfield**: campo para seleccionar fechas. Al pulsar sobre el campo aparece una ventana en la que podemos seleccionar fácilmente una fecha. Podemos indicarle una fecha inicial utilizando " `value: {year: 1989, day: 1, month: 5}` ":

- **fieldset**: Este elemento en realidad no es un campo de datos, sino un contenedor. No añade ninguna funcionalidad, simplemente pone un título (opcional), y agrupa elementos similares, de la forma:

```
items: [{
xtype: 'fieldset',
title: 'About Me',
items: [
  { xtype: 'textfield', name : 'firstName', label: 'First Name' },
  { xtype: 'textfield', name : 'lastName', label: 'Last Name'}
]
}]
```

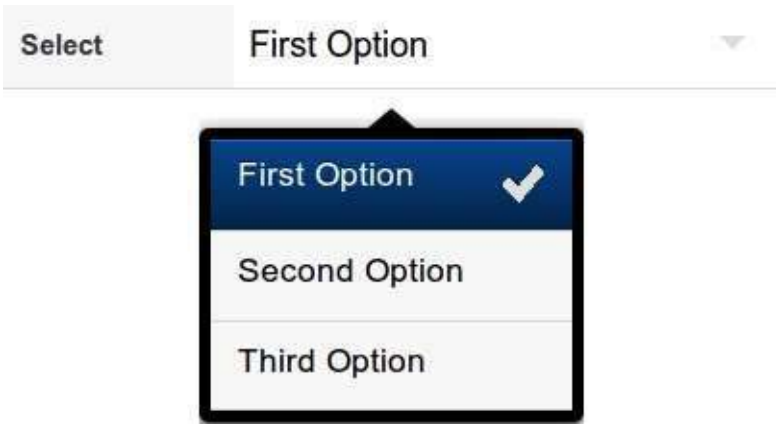
Con lo que obtendríamos un resultado similar a:



- **selectfield**: campo desplegable para seleccionar entre una lista de valores. Las posibles opciones se indican en la propiedad " `options` " como un array. Para cada opción tenemos que indicar sus valores `text` (texto que se mostrará) y `value` (valor devuelto para la opción seleccionada).

```
items:[{
xtype: 'selectfield',
label: 'Select',
options: [
  {text: 'First Option',  value: 'first'},
  {text: 'Second Option', value: 'second'},
  {text: 'Third Option',  value: 'third'}
]
}]
```

  Con lo que obtendríamos un resultados como el siguiente:



- **checkboxfield**: el campo checkbox nos permite elegir uno o varios elementos de una lista. Cada campo de la lista se tiene que declarar como un item independiente, pero todos ellos deben de tener el mismo nombre " `name` " para poder ser agrupados (muy importante para posteriormente poder recoger los datos correctamente). Además podemos utilizar la propiedad " `checked: true` " para que aparezcan marcados inicialmente:

```
items: [
{
  xtype: 'checkboxfield',
  name : 'check_color',  // Nombre del grupo
  value: 'red',
  label: 'Red',
  checked: true
}, {
  xtype: 'checkboxfield',
  name : 'check_color',
  value: 'green',
  label: 'Green'
}, {
  xtype: 'checkboxfield',
  name : 'check_color',
  value: 'blue',
  label: 'Blue'
}
]
```

Con lo que obtendríamos un resultado como el siguiente (en la imagen se han agrupado además dentro de un *fieldset*):

- **radiofield**: el campo de tipo "radio" nos permite elegir **solo un elemento** de una lista. Cada campo de la lista se tiene que declarar como un item independiente, pero todos ellos deben de tener el mismo nombre " `name` " para poder ser agrupados (muy importante para posteriormente poder recoger los datos correctamente). Además podemos utilizar la propiedad " `checked: true` " en uno de ellos para que aparezca marcado inicialmente:

```
items: [
{
  xtype: 'radiofield',
  name : 'radio_color',  // Nombre del grupo
  value: 'red',
  label: 'Red',
  checked: true
}, {
  xtype: 'radiofield',
  name : 'radio_color',
  value: 'green',
  label: 'Green'
}, {
  xtype: 'radiofield',
  name : 'radio_color',
  value: 'blue',
  label: 'Blue'
}
]
```

Con lo que obtendríamos un resultado similar a (en la imagen se han agrupado además dentro de un *fieldset*):

También podemos instanciar los campos de un formulario de forma independiente utilizando su constructor, por ejemplo para el campo de " `textfield` " sería " `Ext.form.Text` ", o para el campo " `togglefield` " sería " `Ext.form.Toggle` ". En general el constructor tendrá el mismo nombre que su tipo " `xtype` " empezando por mayúscula y quitando el sufijo " `field` ".

Almacenamiento

En esta sección vamos a ver las opciones que tenemos para trabajar con los datos de una aplicación. Para esto Sencha Touch pone a nuestra disposición varias herramientas:

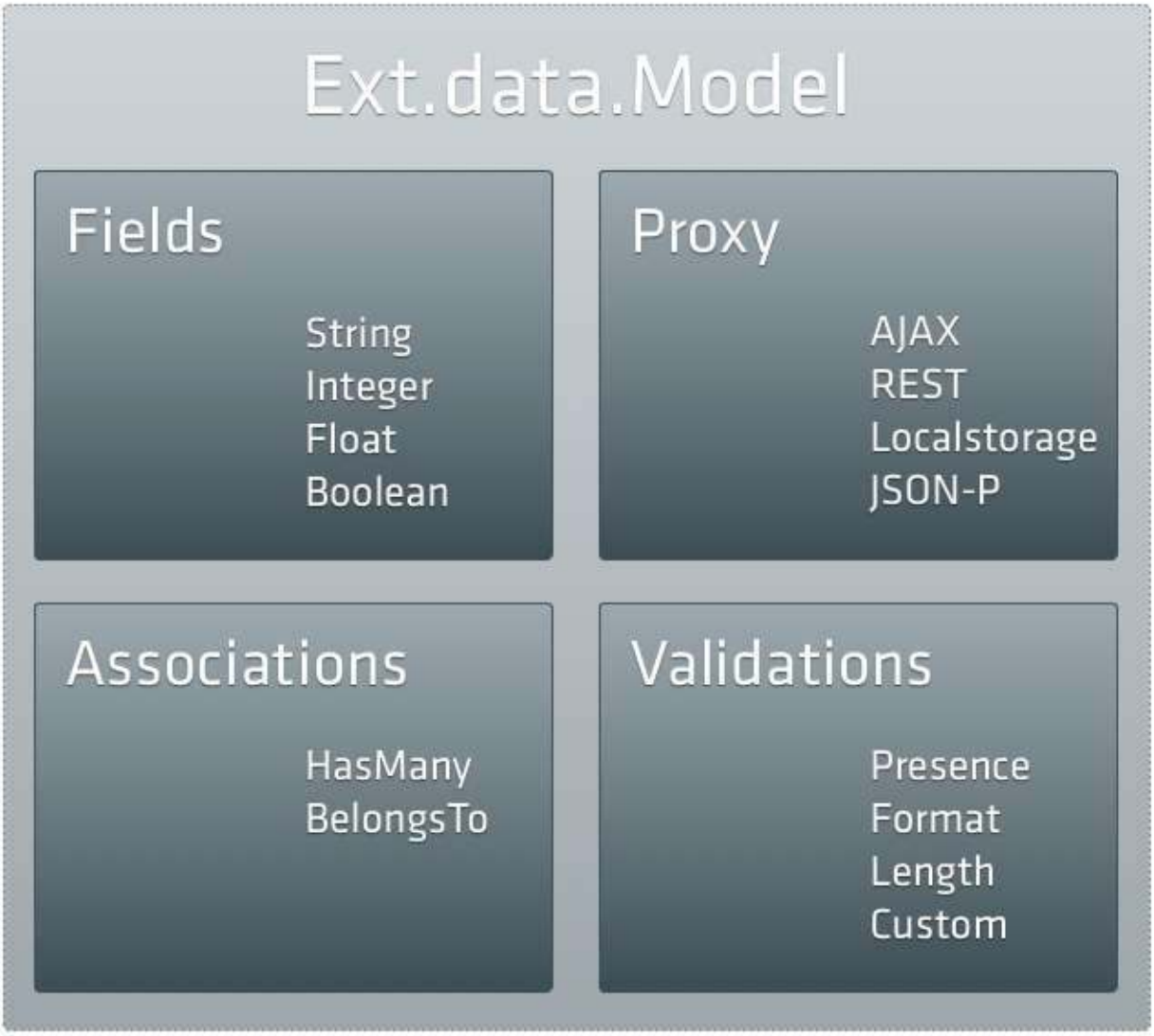
- *Data Model*: Nos permitirá representar las entidades de datos como colecciones de campos con sus datos asociados.
- *Data Store*: Son colecciones de datos de un modelo. Además nos permitirá realizar otro tipo de operaciones más avanzadas como ordenar, filtrar, agrupar o lanzar eventos.
- *Proxy*: Permiten cargar y almacenar los datos desde una fuente de datos remota o local en un *data model* o un *data store*.

Además también se estudiarán varias formas de utilizar estos datos en una aplicación, por ejemplo en listados, *data views* o en formularios. A continuación veremos en detalle cada uno de estos apartados.

Data Model

Los *Data Model* nos permiten representar las entidades de datos, junto con sus validaciones de formato y las relaciones con otros modelos, como si estos fuera objetos o clases. Por comparación podríamos pensar que un *Data Model* es como una clase (en POO) con la definición de los datos que la componen y funciones para poder validar y trabajar con esos datos.

A continuación podemos ver un esquema de todas las opciones que agrupa un *Data Model*, las cuales iremos viendo se las siguientes secciones:



Definir un modelo

Para crear un modelo usamos el constructor `Ext.define('<Model-Name>', { ... })`, en el que como primer parámetro indicamos el nombre del modelo a crear y como segundo parámetro recibe un objeto con la configuración del modelo. En la configuración de modelo tendremos que indicar dos cosas: que hereda de la clase `Ext.data.Model` y los campos que definen al modelo. Por ejemplo:

```
Ext.define('User', {    // Nombre del modelo: User
    extend: 'Ext.data.Model',
    config: {
        fields: [        // Descripción de los campos que lo componen
            { name: 'id', type: 'int' },
            { name: 'usuario', type: 'string' },
            { name: 'nombre',  type: 'string' },
            { name: 'genero',  type: 'string' },
            { name: 'activo',  type: 'boolean', defaultValue: true},
            { name: 'fechaRegistro', type: 'date', dateFormat: 'c' }
        ]
    }
});
```

En este ejemplo hemos creado el modelo `User` con cinco campos. Por defecto, en todos los modelos se asigna como identificador principal el campo `id`, el cual se utiliza para varias cosas (como saber si el modelo ha sido guardado). Si queremos variar el campo identificador podemos hacerlo con `idProperty: 'id'`.

Al heredar de la clase `Ext.data.Model` obtiene una serie de métodos y propiedades que nos van a permitir trabajar con los datos del modelo y almacenarlo de forma permanente.

En la sección `fields` se definen el resto de campos que componen el modelo. Para cada campo podemos utilizar:

- *name*: nombre o identificador del campo.
- *type*: los campos pueden ser de cinco tipos: *string, int, float, boolean, date, auto*.
- *defaultValue*: atributo opcional que nos permite indicar el valor por defecto de un campo.
- *dateFormat*: atributo opcional mediante el cual podemos indicar el formato de la fecha (ver http://docs.sencha.com/touch/2.4/2.4.1-apidocs/#!/api/Ext.Date).

Cuando se asigna el tipo *auto* o no se asigna ningún tipo el campo podrá obtener cualquier tipo de valor y no realizará ninguna validación de formato.

Crear una instancia de un modelo

Para crear una instancia de un modelo utilizamos también el constructor `Ext.create('<Model-Name>', { ... })` especificando el nombre del modelo del cual queremos crear una instancia y un objeto con los valores a asignar a la instancia. Por ejemplo:

```
var ed = Ext.create('User', {
    usuario: 'ajgallego',
    nombre: 'Javier Gallego',
    genero: 'Masculino',
    activo: true
});
```

Validaciones

Los modelos de datos incluyen soporte para realizar validaciones, las cuales las deberemos de incluir dentro de la misma clase a continuación del campo `fields`. Por ejemplo:

```
Ext.define('User', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            // ...
        ],
        validations: [
            { type: 'presence',  field: 'nombre' },
            { type: 'length',    field: 'nombre', min: 5 },
            { type: 'format',    field: 'usuario', matcher: /([a-z]+)[0-9]{2,3}/},
            { type: 'inclusion', field: 'genero', list: ['Masculino', 'Femenino'] },
            { type: 'exclusion', field: 'usuario', list: ['admin'] }
        ]
    }
});
```

Los tipos de validaciones que podemos utilizar son los siguientes:

- *presence*: Indica que el campo es requerido.
- *length*: Valida la longitud del campo. Podemos indicar como atributos un mínimo (*min*) y/o un máximo (*max*).
- *exclusion*: Valida que el valor del campo no se encuentre entre la lista de valores especificados en *list*.
- *inclusion*: Valida que el valor del campo se encuentre entre la lista de valores especificados en *list*.
- *format*: Permite especificar una expresión regular (mediante el atributo *matcher*) para validar el campo.

Además, definiendo el valor de la propiedad *message*, podemos cambiar el mensaje de error que se produciría si una validación no es correcta:

```
{ field: 'titulo', type: 'presence',
  message: 'Por favor, introduzca un título' }
```

Cuando trabajemos con un formulario y queramos comprobar estas validaciones lo tendremos que hacer manualmente llamando a la función `validate()`. A continuación se incluye un ejemplo:

```
var newUser = Ext.create('User', {
    nombre: 'Javi',
    usuario: 'admin',
    genero: 'género no válido'
});

var errors = newUser.validate();

console.log('Válido?', errors.isValid()); // 'false' si hay errores
console.log('Errores:', errors.items); // array con los errores encontrados
console.log('Errores en género:', errors.getByField('genero'));
```

Además, como se puede ver en el ejemplo también disponemos de las funciones `isValid()` para comprobar si han habido errores, `.items` para obtener el array de errores encontrados y `getByField('<nombre-del-campo>')` para obtener los errores de un campo dado. En la sección de formularios se verá como integrar la validación de un modelo con un formulario.

Relaciones con otros modelos

Los modelos de datos también soportan la creación de relaciones con otros modelos de datos, del tipo "`hasMany`" y "`belongsTo`". Estos contenidos se quedan fuera de este capítulo de introducción por lo que para obtener más información podéis consultar la web:

http://docs.sencha.com/touch/2.4/core_concepts/data/models.html

Uso de modelos de datos en una aplicación

Al crear un modelo lo que estamos haciendo en realidad es crear una nueva definición de una clase (y no una instancia de una clase). Se puede observar la diferencia en el constructor, ya que usamos `Ext.define('<nombre-modelo>', { ... });` en lugar de `Ext.create`.

En estos casos, en los que creemos nuevas definiciones, estaremos obligados a cargarlos por separado. Para ello Sencha Touch incorpora un potente sistema que nos permite modularizar nuestro código siguiente el patron MVC muy fácilmente. Solamente tendremos que seguir los siguientes pasos:

- Crear una fichero con el mismo nombre que el modelo dentro de la carpeta `app/model`. Si por ejemplo nuestro modelo se llama `User` el fichero se tendrá que llamar `User.js`.
- El contenido del fichero del modelo seguirá la sintaxis que hemos visto hasta ahora pero añadiendo el espacio de nombres `<nombre-app>.model.<nombre-modelo>` al nombre del modelo, por ejemplo si nuestra aplicación se llamase `MyApp` y el modelo a crear `User`, el modelo tendría que quedar como el siguiente:

```
Ext.define('MyApp.model.User', {    // Nombre del modelo con espacio de nombres
    extend: 'Ext.data.Model',
    config: {
        fields: [
            { name: 'id', type: 'int' },
            { name: 'name', type: 'string' }
        ]
    }
});
```

- Por último tendremos que decirle a nuestra aplicación que cargue la definición del modelo para que podamos utilizarla. Para esto simplemente tenemos que hacer:

```
Ext.application({
    name: 'MyApp',
    models:['User'],
    launch: function() { ... }
});
```

A la hora de **cargar** el modelo podemos indicarlo usando solamente el nombre del modelo o usando el espacio de nombres completo (`MyApp.model.User`). Pero a la hora de **utilizarlo** (por ejemplo, para crear una instancia del modelo o asociarlo a un *store*) tendremos que usar la ruta del espacio de nombres completa.

Data Store

Los almacenes de datos (*data store*) se utilizan para encapsular o almacenar una colección de instancias de un modelo determinado. Además disponen de funciones para ordenar, filtrar y consultar los datos. De forma opcional podemos indicar que utilicen un *proxy* para sincronizar estos datos con un almacén local o remoto.

En esta sección nos vamos a centrar en las características para gestionar un *store*: añadir, ordenar, filtrar, buscar y eliminar. En la siguiente sección sobre *proxies* veremos como hacer persistentes estos datos.

Crear un *Data Store*

Crear un almacén de datos es fácil, utilizaremos el constructor `Ext.create('Ext.data.Store', {...})` y como segundo parámetro le pasaremos las opciones de configuración indicando el nombre del modelo:

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'User'
});
```

Añadir datos

Podemos añadir datos directamente junto a la definición de un *Store*, solo tenemos insertarlos como un array a través de su propiedad *"data"*. Suponiendo que el modelo "User" solo tuviera dos campos (*id*, *name*), podríamos añadir datos de la forma:

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'User',
    data: [
        {id: 1, name: 'Javier Gallego'},
        {id: 2, name: 'Fran García'},
        {id: 3, name: 'Boyan Ivanov'},
        {id: 4, name: 'Miguel Lozano'}
    ]
});
```

O también podemos añadir datos posteriormente llamando a la función " `add` " del objeto:

```
myStore.add({id: 5, name: 'Javier Aznar'},
            {id: 6, name: 'Pablo Suau'});
```

Ordenar y Filtrar elementos

Para ordenar y filtrar los datos usamos las propiedades " `sorters` " y " `filters` ". En el siguiente ejemplo se ordenan los datos de forma descendente por nombre de usuario (también podría ser `ASC`) y se realiza un filtrado por género (los filtros también admiten expresiones regulares).

```
Ext.create('Ext.data.Store', {
    model: 'User',
    sorters: [
        { property: 'usuario', direction: 'DESC' }
    ],
    filters: [
        { property: 'genero', value: 'Femenino' }
    ]
});
```

Buscar registros

En algunos casos antes de añadir un registro será necesario comprobar si el registro está repetido. Para esto podemos utilizar el método `findRecord(campo, valor)` del *Store*, el cual devuelve el registro encontrado o *null* en caso de no encontrar ninguna coincidencia. En el siguiente ejemplo se compara el campo *id* de los datos del *Store*, con el campo *id* del registro a añadir:

```
if (myStore.findRecord('id', registro.data.id) === null)
{
    myStore.add( registro );
}
```

Otra opción para buscar registros es la función `find(campo, valor)` la cual devuelve el índice del registro encontrado (o -1 en caso de no encontrarlo), y posteriormente podríamos llamar a `getAt(index)` para obtener los datos.

Eliminar registros

Para eliminar un registro de un *Store* usaremos la función `remove(registro)` , por ejemplo:

```
myStore.remove( registro );
```

Es recomendable comprobar si existe el registro a eliminar, para esto usaremos la función `findRecord()` . Normalmente el *Store* estará asignado a algún panel que nos permita ver los datos (como un listado). Si quisiésemos eliminar un registro de este listado, primero

tendríamos que obtener el *Store* usado, a continuación comprobar si existe el registro y si es así eliminarlo. Por último habría que sincronizar los datos para que se actualicen, de la forma:

```
var store = miListado.getStore();

if( store.findRecord('id', registro.data.id) )
{
    store.remove( registro );
}

store.sync();
miListado.refresh();
```

También podemos utilizar la función `removeAt(index)` para eliminar los registros de un *Store* a partir de su índice.

Proxy

Los *proxies* se utilizan para definir la forma de leer y escribir la información. Dependiendo del proxy que utilicemos podremos almacenar los datos de forma local o de forma remota. Además, estos pueden ser definidos tanto en el *data store* como en el *data model*.

Los posibles *proxies* u opciones de almacenamiento que podemos usar son:

- Almacenamiento en local:
    - Memoria (*memory*): almacenar los datos en memoria.
    - Local Storage (*localstorage*): almacenar los datos usando la nueva característica de HTML5 de almacenamiento en local.
    - Session Storage (*sessionstorage*): almacenar los datos usando la nueva característica de HTML5 de almacenamiento en local pero por sesión (al cerrar la sesión se borrarán los datos).
    - SQL (*sql*): almacenar los datos usando la nueva característica de HTML5 de almacenamiento en una base de datos local de tipo WebSQL.
- Almacenamiento en remoto:
    - Ajax (*ajax*): Genera peticiones ajax para cargar o enviar los datos a un servidor.
    - JsonP (*jsonp*): JSONP o JSON con padding es una técnica de comunicación utilizada para realizar llamadas asíncronas a dominios diferentes y de esta forma evitar los errores por peticiones *cross-domain*.
    - Rest (*rest*): Especialización del *proxy* ajax para realizar peticiones a un servidor RESTful.

A continuación se tratarán las posibles formas de definición de un *proxy* y algunos ejemplos de uso usando los tipos *ajax*, *localstorage* y *rest*.

Definir el *proxy* en el *Data Store*

En un *Data Store*, además de definir el modelo asociado, podemos definir el *proxy* a utilizar. En el siguiente ejemplo se configura un *proxy* de tipo *ajax*, además se le proporciona la `url` con la dirección para acceder a los datos y la propiedad `reader` para especificar el tipo de datos con los que tiene que trabajar.

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'User',
    proxy: {
        type: 'ajax',
        url : 'users.json',
        reader: 'json'
    },
    autoLoad: true
});
```

Además hemos añadido la propiedad `autoLoad: true` para que se carguen los datos al inicio desde el proxy indicado. Si no lo hiciéramos así el *store* inicialmente estaría vacío, aunque también podríamos cargarlos utilizando el método `myStore.load()`.

La propiedad `reader` especifica la forma de codificar / decodificar los datos. En este caso se ha especificado el tipo *json*, pero Sencha Touch admite también el tipo *xml*.

Al cargar la aplicación con el *store* del ejemplo se auto-descargarían los datos desde la URL indicada. A continuación, y según tiene definido el *store*, espera recibir un objeto JSON con un array de datos con los mismos campos que el modelo *User* asociado. El formato de los datos en JSON debería ser similar al siguiente:

```
{
    success: true,
    users: [
        { id: 1, name: 'Greg' },
        { id: 2, name: 'Seth' }
    ]
}
```

En caso de que la raíz de los datos del array fuese distinta podríamos indicarlo configurando el `reader` de la forma:

```
    ...
    proxy: {
        type: 'ajax',
        url : 'users.json',
        reader: {
            type: 'json',
            root: 'usersList'
        }
    }
```

Definir el *proxy* en el *Data Model*

El *proxy* también se puede definir directamente en el modelo de datos, lo cual tiene dos beneficios. Primero, si el modelo se utiliza en varios *stores* solo se tendrá que especificar una vez la configuración del *proxy*. Y segundo, podremos cargar datos y guardarlos sin necesidad de referenciar el *store*.

En el siguiente ejemplo se puede ver como definir un *proxy* directamente en el modelo (las opciones de configuración son exactamente las mismas que si lo hiciéramos en el *store*):

```
Ext.define('User', {
    extend: 'Ext.data.Model',
    config: {
        fields: ['id', 'name', 'age', 'gender'],
        proxy: {
            type: 'rest',
            url : 'data/users',
            reader: {
                type: 'json',
                root: 'users'
            }
        }
    }
});

// Definimos el store y lo asociamos, el cual ya tendrá configurado el proxy
var myStore = Ext.create('Ext.data.Store', {
    model: 'User'
});
```

En este otro ejemplo se muestra como guardar y cargar datos directamente desde el modelo que hemos creado en el ejemplo anterior:

```
// Crear un usuario
var item = Ext.create('User', {
    name: 'Bilbo Bolsón',
    age : 111
});

// Almacenar el usuario creado. Esto enviará una petición tipo POST
// a la ruta "/users", ya que tiene definido un proxy tipo rest.
item.save({
    success: function(item) {
        console.log("El usuario se ha guardado con ID: "+ item.getId());
    }
});

// Obtener una referencia al modelo User
var User = Ext.ModelMgr.getModel('User');

// Cargar el usuario con id = 1
User.load(1, {
    success: function(user) {
        console.log("Usuario 1 con nombre: " + user.get('name'));
    }
});
```

Almacenamiento en local con *localstorage*

Para este tipo de almacenamiento, el proxy utiliza la nueva funcionalidad de HTML5 de almacenamiento en local. Esta opción es muy útil para almacenar información sin la necesidad de utilizar un servidor. Sin embargo solo podremos guardar pares clave-valor, no soporta objetos complejos como JSON (*JavaScript Object Notation*).

Al usar almacenamiento en local es muy importante que el modelo de datos tenga un "id" único, que por defecto tendrá ese nombre de campo (*id*); en caso de utilizar otro lo podríamos indicar mediante la propiedad " `idProperty: 'myID'` " del modelo.

Para que el proxy utilice el almacenamiento en local simplemente tendremos que indicar el tipo (`type: 'localstorage'`) y un identificador (`id` , usado como clave para guardar los datos). En el siguiente ejemplo se crea un almacén de datos para el modelo "User" que hemos definido en las secciones anteriores.

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'User',
    proxy: {
        type: 'localstorage',
        id: 'my-store-id'
    },
    autoLoad: true
});
```

Componentes asociados a datos

En este apartado veremos algunos de los componentes avanzados que incorpora Sencha Touch que nos permitirán mostrar los datos de un almacenamiento. Estos componentes incorporan propiedades que nos van a permitir enlazarlos directamente con un *store* o almacenamiento dado y que nos facilitarán la sicronización de los datos.

En primer lugar analizaremos las "plantillas", las cuales nos permitirán indicar la disposición (o vista) de los datos de un almacenamiento, y seguidamente veremos tres componentes que podremos utilizar para mostrar los datos de un almacenamiento, estos son:

- DataViews
- Listados
- Formularios

Plantillas

Las plantillas se utilizan para describir la disposición y la apariencia visual de los datos de nuestra aplicación. Nos proporcionan funcionalidad avanzada para poder procesarlos y darles formato, como: auto-procesado de arrays, condiciones, operaciones matemáticas, ejecución de código en línea, variables especiales, funciones, etc.

Para instanciar un template utilizamos el constructor " `Ext.XTemplate( template )` ", donde *template* será una cadena con la definición del template a utilizar. Posteriormente podemos utilizar la función " `overwrite(elemento, datos)` " del *template* para aplicar un template con unos datos sobre un elemento dado. En la sección de "Visualización de datos" se detalla otra forma de aplicar un template en un panel.

Auto-procesado de arrays

Para crear un template que procese automáticamente un array se utiliza la etiqueta `<tpl for="variable">plantilla</tpl>` , teniendo en cuenta que:

- Si el valor especificado es un array se realizará un bucle por cada uno de sus elementos, repitiendo el código de la "plantilla" para cada elemento.
- La "plantilla" puede contener texto, etiquetas HTML y variables o elementos del array a sustituir.
- Las variables a sustituir se indican de la forma " `{nombre_variable}` ", donde " `nombre_variable` " debe de corresponder con el nombre de uno de los elementos del array iterado.
- Mediante la variable especial `{#}` podemos obtener el índice actual del array.
- En la sección `for="variable"` de la plantilla se debe de indicar el nombre de la variable que contiene el array a procesar, de la forma:
  - Con `<tpl for=".">...</tpl>` se ejecuta un bucle a partir del nodo raíz.
  - Con `<tpl for="foo">...</tpl>` se ejecuta el bucle a partir del nodo "foo".
  - Con `<tpl for="foo.bar">...</tpl>` se ejecuta el bucle a partir del nodo "foo.bar"

Si por ejemplo tenemos el siguiente objeto de datos:

```
var myData = {
  name: 'Tommy Maintz',
  drinks: ['Agua', 'Café', 'Leche'],
  kids: [
    { name: 'Tomás', age:3 },
    { name: 'Mateo', age:2 },
    { name: 'Salomón', age:0 }
  ]
};
```

Podríamos mostrar un listado con el contenido de `myData.kids` indicando en la sección `for="variable"` que procese a partir de la raíz del array `myData.kids` :

```
var myTpl = new Ext.XTemplate(
  '<tpl for=".">',
    '<p>{#}. {name}</p>',
  '</tpl>' );

myTpl.overwrite(myPanel.element, myData.kids);
```

Si por ejemplo indicamos que se procese la variable "*myData*" y queremos obtener el mismo resultado, tendríamos que modificar el *template* para que se procese a partir del nodo "*kids*", de la forma `<tpl for="kids">...</tpl>` .

```
var myTpl = new Ext.XTemplate(
  '<tpl for="kids">',
    '<p>{#}. {name}</p>',
  '</tpl>' );

myTpl.overwrite(myPanel.element, myData);
```

Si el array solo contiene valores (en el objeto de datos de ejemplo, sería el array "*drinks*"), podemos usar la variable especial `{.}` dentro del bucle para obtener el valor actual:

```
var myTpl = new Ext.XTemplate(
  '<tpl for="drinks">',
    '<p>{#}. {.}</p>',
  '</tpl>' );

myTpl.overwrite(myPanel.element, myData);
```

Condiciones

Para introducir condiciones en las plantillas se utiliza la etiqueta `<tpl if="condicion">plantilla </tpl>`. Hemos de tener en cuenta que: si utilizamos símbolos como las "`<`", "`>`" o las "comillas" deberemos escribirlos codificados: `<`, `>` o `&quot;`, y que si usamos los operadores "`elseif`" o "`else`" tendremos que cerrar la plantilla al final.

Ejemplos:

```
<tpl if="age &lt; 10">
    Niño
<tpl elseif="age &gt= 10 && age &lt; 18">
    Adolescente
<tpl else>
    Adulto
</tpl>
<tpl if="name == &quot;Javi&quot;">¡Hola Javi!</tpl>
```

Visualización

Para renderizar el contenido de una plantilla sobre un panel (u otro elemento que lo soporte, como veremos más adelante), podemos usar la función "`tpl.overwrite(elemento, datos)`" que ya hemos usado en los ejemplos anteriores. O usar la propiedades "`tpl`" junto con "`data`", de la forma:

```
var myPanel = Ext.create('Ext.Panel', {
    data: myData,
    tpl: myTpl
});

// O también:
// myTpl.overwrite(myPanel.element, myData);
```

Data Views

Los *Data Views* nos permiten mostrar datos de forma personalizada mediante el uso de plantillas y opciones de formato. Principalmente se utilizan para mostrar datos provenientes de un *store* y aplicarles formato utilizando las plantillas " `Ext.XTemplate` ", como hemos visto en la sección anterior. Además también proporcionan mecanismos para gestionar eventos como: *click, doubleclick, mouseover, mouseout*, etc., así como para permitir seleccionar los elementos mostrados (por medio de un "*itemSelector*").

En el siguiente ejemplo vamos a crear un *DataView* para mostrar el contenido de un *store* definido por separado y mediante la utilización de una plantilla también previamente definida usando `Ext.XTemplate` :

```
var myDataView = Ext.create('Ext.DataView', {
    fullscreen: true,
    store: myStore,
    tpl: myTpl
});
```

En este otro ejemplo creamos un *DataView* para mostrar el contenido de un *store* que incluimos dentro de la propia clase con un array de datos interno. Además utilizamos la propiedad `itemTpl` el lugar de `tpl` , lo que nos permite indicar el *template* de cada elemento o *item* del array directamente.

```
var myDataView = Ext.create('Ext.DataView', {
    fullscreen: true,
    store: {
        fields: ['name','age'],
        data: [
            {name:'Manuel', age:21},
            {name:'Pedro', age:56},
            {name:'Javi', age:36},
            {name:'Laura', age:57},
            {name:'Alfredo', age:11},
            {name:'María', age:12}
        ]
    },
    itemTpl: '{name} tiene {age} años'
});
```

Esta "vista de datos" podemos mostrarla en nuestra aplicación tal cual la hemos creado o también podemos añadirla a la sección `items` de un panel:

```
var myPanel = Ext.create('Ext.Panel', {
    fullscreen: true,
    items: [ myDataView ]
});
```

Listados

Permiten mostrar datos en forma de listado a partir de una plantilla por defecto de tipo lista. Estos datos se obtienen directamente de un "*store*" y se mostrarán uno a uno en forma de listado según la plantilla definida en " `itemTpl` ". Además incorpora funcionalidades para gestionar eventos como: *itemtap*, *itemdoubletap*, *containertap*, etc.

Utilizarlo es muy simple, solo tenemos que definir el "*store*" que queremos utilizar y la plantilla para cada uno de los elementos con " `itemTpl` ", por ejemplo:

```
var myList = Ext.create('Ext.List', {
    fullscreen: true,
    store: {
        fields: ['firstName', 'lastName'],
        data: [
            {firstName: 'Pedro', lastName: 'García'},
            {firstName: 'Jorge', lastName: 'Sánchez'},
            {firstName: 'Sandra', lastName: 'Gallego'},
            {firstName: 'María', lastName: 'Pérez'}
        ]
    },
    itemTpl: '{firstName} {lastName}'
});
```

En el ejemplo anterior hemos creado el *store* a utilizar directamente dentro de la lista con un array de datos interno, pero también podríamos crear el *store* de datos por separado y conectarlo alguna fuente de datos local o remota (como se vio en la sección correspondiente):

```
var myList = Ext.create('Ext.List', {
    fullscreen: true,
    store: myStore,
    itemTpl: '{firstName} {lastName}'
});
```

El código del ejemplo generaría una aplicación como la siguiente:

Es muy importante diferenciar " `itemTpl` " de la propiedad " `tpl` " que ya habíamos visto (en las que usábamos los `XTemplate`). En " `itemTpl` " se procesa cada elemento del listado individualmente. Otra diferencia es que tenemos que utilizar como separador para la concatenación el símbolo de unión "+" y no la coma ",".

Esta lista ya la podemos mostrar directamente, sin la necesidad de incluirla en un contenedor, sin embargo, también podemos añadirla dentro de un panel en su sección `items` de la forma:

```
var myPanel = Ext.create('Ext.Panel', {
    layout: 'fit',
    items: [ myList ]
});
```

> Nota: es posible que al asignar un listado a un panel no se visualice correctamente. En este caso tenemos que asegurarnos de eliminar la propiedad `fullscreen` tanto del listado como del panel y de asignar el `layout: 'fit'` al panel (y ningún layout al listado).

En el "*store*" debemos de utilizar la propiedad " `sorters` " para ordenar el listado, pues sino nos aparecerá desordenado. Por ejemplo, podríamos indicar (en el "*store*") que se ordene por el apellido " `sorters: 'lastName'` ".

Obtener datos de la lista

Para obtener el almacén de datos asociado a un listado utilizamos su método `getStore()` :

```
var notesStore = myList.getStore();
```

Una vez obtenido ya podemos realizar operaciones sobre él como añadir, modificar o eliminar algún registro (consultar sección correspondiente).

Actualizar datos

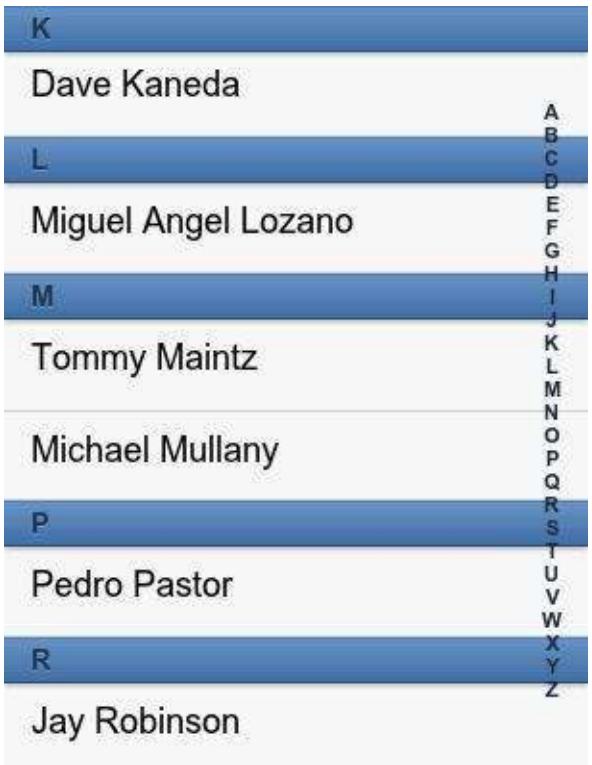
Si modificamos el almacén de datos asociado con el listado tendremos que actualizarlo para que se visualicen correctamente los nuevos datos en el listado. En primer lugar llamaremos al método `sync()` del *store* para sincronizar los cambios. A continuación, si es necesario, ordenamos los datos (pues el registro se habrá añadido al final). En el ejemplo se ordenan de forma descendente por fecha. Por último llamamos al método `refresh()` del listado para actualizar la vista.

```
notesStore.add( registro );

notesStore.sync();
notesStore.sort([{ property: 'date', direction: 'DESC'}]);
myList.refresh();
```

Agrupar elementos

Una propiedad muy útil que nos ofrecen los listados es la posibilidad de agrupar los elementos (como podemos ver en la imagen inferior). Para esto activaremos la propiedad " `grouped: true` " del listado y opcionalmente podremos indicar que se muestre una barra lateral de navegación " `indexBar: true` ".



Pero para que esta propiedad funcione correctamente tendremos que indicar dentro del *store* la forma de agrupar los elementos. Tenemos dos opciones:

- `groupField: 'campo'` - para agrupar por un campo (por ejemplo: elementos de género masculino y femenino).
- `getGroupString: function(instance) {...}` - para agrupar usando una función. Esta opción es mucho más avanzada y nos permitirá agrupar, por ejemplo, usando la primera letra del apellido (como se muestra en la imagen de ejemplo).

Para obtener el resultado de la imagen de ejemplo anterior, el código quedaría como el siguiente:

```
var myStore = Ext.create('Ext.data.Store', {
    model: 'User',
    proxy: {
        type: 'localstorage',
        id: 'my-store-id'
    },
    autoLoad: true,
    sorters: 'apellido',
    getGroupString: function(instance) {
        return instance.get('apellido')[0];
    }
});

var myList = Ext.create('Ext.List', {
    fullscreen: true,
    store: myStore,
    grouped : true,
    indexBar: true,
    itemTpl: '{nombre} {apellido}'
});
```

Acciones

Para añadir acciones al presionar sobre un elemento del listado tenemos varias opciones:

- `itemtap` : permite realizar una acción al presionar sobre un elemento de la barra. Este evento lo debemos definir dentro de la sección " `listeners` " de nuestro `Ext.List` , de la forma:

```
listeners: {
  itemtap: function(view, index, target, record) {
      alert( "tap on" + index );
  }
}
```

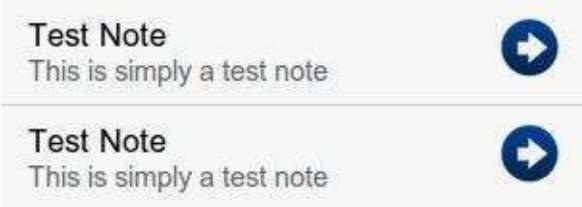
Donde el parámetro *record* representa el objeto sobre el que se ha pulsado. Este valor lo podríamos aprovechar para cargarlo directamente en un formulario o realizar alguna operación con él.

- `itemdoubletap` : permite realizar una acción al presionar dos veces consecutivas sobre un elemento. Este evento lo debemos definir dentro de la sección " `listeners` " de nuestro `Ext.List` , de la forma:

```
listeners: {
  itemdoubletap: function(view, index, target, record){
      alert("doubletap on "+index);
  }
}
```

Donde el parámetro *record* representa el objeto sobre el que se ha pulsado.

- `onItemDisclosure: Boolean / Función` - esta propiedad admite diferentes valores. Si le indicamos el valor booleano "true" simplemente añadirá un icono con una flecha a la derecha de cada elemento (como podemos ver en la imagen inferior).



En lugar de un valor booleano, podemos indicarle una función. En este caso, además de añadirse el icono en cada elemento, también se ejecutará la función cada vez que se presiones sobre dicho icono. Solo se capturará cuando se presione sobre el icono, no sobre toda la barra (como en el caso de *itemtap*). A continuación se incluye un ejemplo de uso:

```
onItemDisclosure: function (record) { alert( "item disclosure" ); }
```

Donde el parámetro *record* representa el objeto sobre el que se ha pulsado. En el siguiente código, al pulsar sobre un elemento de la lista se cargan los datos del elemento pulsado en un formulario (como veremos más adelante), y se cambia la vista para visualizar ese panel.

```
onItemDisclosure: function( record )
{
  myFormPanel.setRecord( record );
  panelPrincipal.animateActiveItem(
          myFormPanel,
          { type: 'slide', direction: 'left' });
}
```

Formularios

En esta sección vamos a ver como podemos cargar, guardar y validar los datos de un formulario.

Cargar datos en un formulario

Para insertar datos de un modelo en un formulario utilizamos el método " `setRecord( data )` " de la clase " `Ext.form.Panel` ". En el siguiente ejemplo se crea un formulario con dos campos llamados "*title*" y "*text*" y a continuación se cargan los datos del registro "*note*".

```
var noteEditor = Ext.create('Ext.form.Panel', {
  id: 'noteEditor',
  fullscreen: true,
  items: [ {
      xtype: 'textfield',
      name: 'title',
      label: 'Title',
      required: true
    }, {
      xtype: 'textareafield',
      name: 'text',
      label: 'Text'
    } ]
});

noteEditor.setRecord( note );
```

El método " `setRecord( data )` " recibe como parámetro una instancia de un modelo de datos (ver sección *Data Model*), del cual cargará solamente los campos cuyos nombre coincidan con los establecidos en el formulario. En este formulario tenemos dos campos: " `name: 'title'` " y " `name: 'text'` ", si cargamos una instancia de un modelo de datos como el descrito a continuación, solamente se cargarían los dos campos que coinciden.

```
Ext.define('Note', {
    extend: 'Ext.data.Model',
    config: {
        fields: [
            { name: 'id', type: 'int' },
            { name: 'date', type: 'date', dateFormat: 'c' },
            { name: 'title', type: 'string' },
            { name: 'text', type: 'string' }
        ]
    }
});
```

En la sección de "*Data Model*" ya hemos visto que podemos usar la función " `Ext.create('<nombre-modelo>', {lista-valores})` " para crear instancias de un modelo de datos. En el siguiente ejemplo se crea una instancia del modelo anterior y se añade al formulario "noteEditor" que habíamos definido anteriormente. En este caso, al asignar los datos al formulario, solo se cargarían los campos "*title*" y "*text*".

```
var note = Ext.create('Note', {
    id: 1,
    date: new Date(),
    title: 'titulo',
    text: 'texto'
});

noteEditor.setRecord( note );
```

A continuación se incluye un ejemplo un poco más avanzado: Creamos una barra de herramientas a la que añadimos un botón con el texto "Cargar datos". Para este botón definimos su función " `handler` ", de forma que al pulsar el botón se crea una instancia del modelo 'Note' y posteriormente se carga en el formulario 'noteEditor'.

```
var myToolbar = Ext.create('Ext.Toolbar', {
    docked: 'top',
    title: 'Mi barra',
    items: [
        {
            xtype: 'button',
            ui: 'action',
            text: 'Cargar datos',
            handler: function() {
                var now = new Date();
                var noteId = now.getTime();
                var note = Ext.create('Note', {
                    id: noteId,
                    date: now,
                    title: 'titulo',
                    text: 'texto'
                });
                noteEditor.setRecord( note );
            }
        }
    ]
});
```

Guardar los datos de un formulario

Para guardar los datos de un formulario en general tendremos que seguir cuatro pasos:

- En primer lugar llamamos al método `getRecord()` del formulario para obtener su modelo de datos asociado, el cual devolverá un objeto del tipo `Ext.data.Model` con la definición de los campos utilizados en el formulario, pero no sus valores.
- A continuación llamamos a la función `updateRecord(objeto)` del mismo formulario para transferir los valores introducidos a la instancia del modelo que hemos obtenido antes.
- En tercer lugar tenemos que realizar el proceso de validación de los datos (explicado en el siguiente apartado).
- Y por último guardar los datos en el *store* correspondiente. Si tenemos una instancia del almacén de datos creada (ver sección *Data Store*) podemos añadir los datos llamando a su función `add` , de la forma:

```
notesStore.add(currentNote);
```

En el siguiente código de ejemplo se resumen los cuatro pasos que habría que seguir para cargar los datos del formulario 'noteEditor' y guardarlos en el almacén 'notesStore'.

```
// Cargar el modelo de datos
var currentNote = noteEditor.getRecord();

// Actualizar el modelo con los datos del formulario
noteEditor.updateRecord(currentNote);

// Realizar validaciones
// (ver siguiente apartado)

// Guardar los datos
notesStore.add(currentNote);
```

Más comúnmente nuestro almacén estará asociado con algún elemento que nos permita visualizar los datos (como un listado o un *Data View*, ver secciones correspondientes). Si por ejemplo fuera un listado deberíamos de obtener la instancia del almacén de datos llamando a su método `getStore()` y posteriormente añadir los datos que habíamos obtenido del formulario de la forma:

```
var notesStore = notesList.getStore();
notesStore.add( currentNote );
```

Opcionalmente podemos comprobar si los datos a añadir están repetidos. Para esto tenemos que utilizar el método `findRecord()` del *store* (ver sección *Data Store*).

```
var notesStore = notesList.getStore();

if( notesStore.findRecord('id', currentNote.data.id) === null)
{
   notesStore.add( currentNote );
}
```

Para terminar con el ejemplo del listado, una vez añadidos los datos tendremos que sincronizar su *store*, ordenarlo (si fuese necesario) y por último actualizar o refrescar la vista del listado:

```
notesStore.sync();
notesStore.sort([{ property: 'date', direction: 'DESC'}]);

notesList.refresh();
```

Comprobar validaciones

Para comprobar las validaciones de un formulario lo tenemos que hacer de forma manual llamando a la función `validate()` del modelo de datos asociado. Para esto tienen que estar definidas estas validaciones en el modelo. Continuando con el ejemplo del modelo de datos "Note", vamos a añadir que los campos `id`, `title` y `text` sean requeridos:

```
Ext.define('User', {
    extend: 'Ext.data.Model',
    fields: [
        { name: 'id', type: 'int' },
        { name: 'date', type: 'date', dateFormat: 'c' },
        { name: 'title', type: 'string' },
        { name: 'text', type: 'string' }
    ],
    validations: [
        { type: 'presence', field: 'id' },
        { type: 'presence', field: 'title',
          message: 'Introduzca un título para esta nota' },
        { type: 'presence', field: 'text',
          message: 'Introduzca un texto para esta nota' }
    ]
});
```

Los pasos a seguir para realizar la validación de un formulario son los siguientes:

- Obtener el modelo de datos asociado a un formulario (`getRecord()`) y rellenarlo con los datos introducidos por el usuario (`updateRecord()`).
- Llamar a la función `validate()` del modelo de datos. Esta función comprueba que se cumplan todas las validaciones que estén definidas en dicho modelo, devolviendo un objeto del tipo `Errors` .
- A continuación usaremos la función `isValid()` del objeto `Errors` para comprobar si ha habido algún error. Esta función devuelve un valor *booleano* indicando si existen errores o no.
- En caso de que existan errores tendremos que mostrar un aviso con los errores y no realizar ninguna acción más.
- Dado que pueden haber varios errores (guardados en el array `items` del objeto `Errors`), tenemos que iterar por los elementos de este array usando la función `Ext.each(array, funcion)` . Esta función recibe dos parámetros: el primero es el array sobre el que va a iterar y el segundo la función que se llamará en cada iteración. Esta función recibe a su vez dos parámetros: el item de la iteración actual y el índice de ese item en el array.
- Para mostrar el aviso con los errores podemos usar un `MessageBox` (ver sección correspondiente).

A continuación se incluye un ejemplo completo de la validación de un formulario:

```
// Cargar el modelo de datos
var currentNote = noteEditor.getRecord();

// Actualizar el modelo con los datos del formulario
noteEditor.updateRecord(currentNote);

// Realizar validaciones
var errors = currentNote.validate();

if(!errors.isValid())
{
    var message="";

    Ext.each( errors.items, function(item, index) {
        message += item.getMessage() + "<br/>";
    });

    Ext.Msg.alert("Errores", message, Ext.emptyFn);

    return;   // Terminamos si hay errores
}

// Almacenar datos
notesStore.add(currentNote);
```

También podríamos haber creado un bucle para iterar entre los elementos del array de errores, o haber llamado a la función `errors.getByField('title')[0].getMessage()` para obtener directamente el mensaje de error de un campo en concreto.

Más información

Podemos consultar principalmente tres fuentes de información cuando tengamos alguna duda:

- Los **tutoriales y la sección de FAQ** en la página Web de Sencha Touch: http://www.sencha.com/
- La **documentación API** Online:
  - http://docs.sencha.com/touch/2.4/2.4.1-apidocs/
  - http://www.sencha.com/learn/touch/
  - También disponible de forma local accediendo en la dirección "/docs" del SDK descargado.
- Los **foros** en la página web de Sencha Touch.

Además en la carpeta "touch-sdk/examples" podemos encontrar aplicaciones de ejemplo.

Ejercicios 1

En la sección de ejercicios de Sencha Touch vamos a practicar creando una pequeña aplicación móvil que nos permita gestionar un listado de notas. La pantalla principal contendrá el listado de notas, con botones para crear nuevas notas y modificarlas. Al crear una nueva nota nos aparecerá un formulario en el que podremos introducir los datos y añadirlos al listado. Este mismo formulario también se utilizará para modificar las notas existentes. Además desde la pantalla principal podremos abrir la ayuda con información sobre la aplicación y el autor.

> En todos los ejercicios se utilizará el mismo proyecto de Sencha Touch, el cual se irá completando ejercicio tras ejercicio hasta tener la aplicación completa. Para la entrega solamente será necesario enviar el resultado final.

Ejercicio 1 - Estructura de la aplicación (0.5 puntos)

En este primer ejercicio vamos a crear el proyecto de Sencha Touch que utilizaremos en todos los ejercicios y la función de inicialización del mismo.

Si ya tenemos descargado e instalado el SDK y el Cmd de Sencha Touch procedemos a crear una carpeta llamada `misnotas` y a generar un proyecto dentro de esta llamado `MisNotas`.

El contenido de los ficheros principales será el siguiente:

- El documento *index.html* no será necesario modificarlo ya que al generar el proyecto ya se incluye la estructura básica, la carga de los recursos necesarios (hojas de estilo y javascript) e incluso un aviso de carga que aprovecharemos para nuestro proyecto.
- En *app.js* es donde vamos a crear nuestra aplicación y los paneles que necesitamos. Borramos el código que se incluye de ejemplo para empezar nuestra aplicación desde cero, siguiendo estos pasos:
    - En primer lugar creamos la instancia de la aplicación a la que pondremos como nombre `MisNotas`. Aquí deberemos definir también la función `launch`, la cual destruirá el *loading* y asignará el panel principal al *viewport*:

```
Ext.application({
    name: 'MisNotas',
    launch: function() {
        Ext.fly('appLoadingIndicator').destroy();
        Ext.Viewport.add( /*TODO: referencia al panel principal*/ );
    }
});
```

- De forma separada creamos el panel principal. En este panel de momento solo indicaremos que ocupe toda la pantalla y que muestre el texto HTML "Mis Notas".

```
var panelPrincipal = Ext.create('Ext.Panel', {
    fullscreen: true,
    html: 'Mis Notas'
});
```

No es necesario que la declaración esté dentro de la función *launch* pero sí que es **importante** que se ejecute después (ya que es necesario que el SDK esté cargado). Para esto podemos crear una función (definida antes de `Ext.application({...});`) que instancie el objeto y lo devuelva o que lo guarde en una variable global (esta segunda opción será mejor ya que después tendremos que hacer referencias a estos objetos). Una posible estructura a seguir es:

```
var panelPrincipal = null;

function crearPanelPrincipal() {
    panelPrincipal = Ext.create('Ext.Panel', { ... });
    return panelPrincipal;
}

Ext.application({
    name: 'MisNotas',
    launch: function() {
        Ext.fly('appLoadingIndicator').destroy();
        Ext.Viewport.add( crearPanelPrincipal() );
    }
});
```

Nota: En las plantillas de los ejercicios se incluye el fichero `app.js` con un esqueleto de la estructura a seguir.

Ejercicio 2 - Creación de paneles (0.5 puntos)

En este ejercicio vamos crear los paneles principales de la aplicación, para esto partiremos del código del ejercicio anterior.

Como ya hemos explicado la aplicación consta de un panel para listar las notas, otro para editar las notas y un tercero para mostrar la ayuda. Además vamos a necesitar un cuarto panel que se utilizará como contenedor. A continuación se explican los pasos que debemos seguir:

En *app.js* modificaremos el panel principal para asignarle un *layout* tipo *card* y una sección `items` con referencias a los tres paneles de la aplicación mediante su nombre de variable, a los que llamaremos `panelContenedorLista`, `panelFormulario` y `panelAyuda`.

Estos tres paneles los definiremos de la misma forma: un panel con *layout* tipo *fit* y un texto *html* de prueba que utilizaremos para comprobar que el panel se visualiza correctamente.

Por último deberemos comprobar que todo funciona correctamente. Para visualizar cada uno de los paneles de momento podemos cambiar su orden en la sección `items` del panel contenedor o llamar al método `panelPrincipal.setActiveItem(1);` del panel.

Ejercicio 3 - Barras de herramientas (1 punto)

Este ejercicio continua con el anterior, al cual vamos a añadir las barras de herramientas necesarias para cada panel.

Las barras las instanciaremos como un objeto separado y después las añadiremos a los paneles:

- En el panel `panelContenedorLista` añadiremos una barra en la parte superior con el título "Mis Notas".
- En el panel `panelFormulario` añadiremos dos barras de herramientas. Una en la parte superior con el título "Editar nota" y una segunda en la parte inferior sin ningún título.
- Por último para el panel `panelAyuda` añadiremos una barra en su parte superior con el título "Ayuda".

Ejercicio 4 - Botones (1 punto)

En este último ejercicio vamos a añadir los botones de cada una de las barras de herramientas que hemos definido en el ejercicio anterior. Los botones los podemos declarar directamente de forma *inline* en la sección `items` de cada barra de herramientas mediante su `xtype`.

- En el panel `panelAyuda` vamos a poner un único botón en su barra de herramientas alineado a la izquierda, en el que colocaremos el icono predefinido de tipo 'home'.
- En el panel `panelContenedorLista` vamos a colocar dos botones en su barra de herramientas. El primer botón estará alineado a la izquierda y contendrá una imagen externa (la imagen `inf.png` de la plantilla). Para añadir esta imagen tendremos que asignar un estilo al botón, llamado "btnAyuda" (ver sección "Botones"). Este estilo lo podemos añadir directamente a los estilos del documento "*index.html*" e indicar que cargue la imagen indicada (recordad usar `!important`) con un ancho de 45 píxeles y un alto de 35 píxeles.
  El segundo botón del panel `panelContenedorLista` estará alineado a la derecha, con el tipo 'action' y el texto "Nueva Nota".
- Por último en el panel `panelFormulario` teníamos dos barras de herramientas. En la barra superior colocaremos dos botones. El primero de ellos estará alineado a la izquierda y con el icono predefinido de tipo 'home'. El segundo botón de esta barra estará alineado a la derecha, será de tipo 'action' y con el texto "Guardar". En la barra inferior colocaremos otro botón alineado a la derecha y con el icono predefinido de tipo 'trash'.

Ejercicios 2

En esta sesión vamos a continuar con el ejercicio del editor de notas de la sesión anterior, al cual añadiremos el contenido de la ayuda, el panel con el formulario y definiremos las transiciones entre paneles.

Ejercicio 1 - Transiciones (1 punto)

Partiendo del código del último ejercicio vamos a añadir las acciones de transición que nos permitirán movernos entre paneles. Todas las transiciones que apliquemos serán del tipo '*slide*' y con una duración de 1 segundo (1000ms).

- Al presionar el botón del panel `panelAyuda` realizaremos una transición hacia arriba para cambiar al panel `panelContenedorLista`.
- En el panel `panelContenedorLista` tenemos dos botones en su barra de herramientas. Para el primer botón alineado a la izquierda definimos una transición hacia abajo que cargue el panel `panelAyuda`. El segundo botón cargará el panel `panelFormulario` con una transición hacia la izquierda.
- Por último, para los tres botones del panel `panelFormulario` asignaremos la misma transición hacia la derecha para cambiar al panel `panelContenedorLista`.

Nota: para obtener las referencias a los paneles para crear las transiciones tenemos dos opciones:

- Tener almacenados los paneles en variables globales y usar directamente estas variables.
- Asignar un identificador a cada panel mediante su atribudo `id` y posteriormente obtener una referencia mediante la función: `Ext.getCmp('id-del-panel')`

Ejercicio 2 - Contenido de la ayuda (1 punto)

En este ejercicio vamos a completar el panel de ayuda. En primer lugar modificamos su constructor para que sea del tipo Carousel y definimos su sección `items` para que contenga dos paneles (además del *toolbar* que ya teníamos). También tenemos que quitar los campos `layout: fit` y el texto HTML que teníamos puesto de prueba.

El contenido HTML de cada uno de estos paneles del Carousel lo definiremos en una variable independiente (llamadas `var htmlAyuda1` y `var htmlAyuda2`), que posteriormente asignaremos al elemento `html` del panel correspondiente. Recordad que para concatenar cadenas tendremos que usar el símbolo más (+).

Para el HTML del primer panel del Carousel definiremos una capa (DIV) a la que asignaremos el estilo ".ayuda". Dentro de esta capa colocaremos el título "*Mis Notas*" (de tipo H1), seguido por un par de párrafos con el texto "*Aplicación Web para la gestión de notas realizada con **Sencha Touch**. Máster en Desarrollo de Aplicaciones para Dispositivos Móviles*". Y por último abajo colocaremos una imagen (logo.png) con un ancho de 100 píxeles (subcarpeta "imgs" de la plantilla).

Para el segundo panel utilizaremos también una capa (DIV) con la clase ".ayuda", en la que colocaremos el titular "Autor", seguido por unos párrafos con los datos del autor.

Por último añadiremos los estilos CSS que hemos definido dentro de la sección de estilos del fichero *index.html*:

- *.ayuda*: tendrá un ancho del 100% y el texto centrado.
- *.ayuda h1*: la etiqueta H1 (cuando esté dentro de la clase "ayuda") la definiremos con un color azul oscuro (*color: navy*), un tamaño de letra de 18 puntos y el estilo "uppercase" para que aparezca siempre en mayúsculas (*text-transform: uppercase*).
- *.ayuda p*: la etiqueta P la definiremos para que tenga color gris (gray).

Ejercicio 3 - Panel con Formulario (1 punto)

En este último ejercicio vamos a crear el formulario. Para esto editamos el panel `panelFormulario` que ya teníamos hecho y le cambiamos el constructor de panel por uno de tipo formulario. Además tenemos que quitar también los campos `layout: fit` y el texto HTML que teníamos puesto de prueba. En este formulario vamos a añadir dos campos al array de `items` (además de las dos barras de herramientas que ya teníamos):

- Un campo de texto con el nombre `title`, con la etiqueta "Título:" y activaremos la opción de requerido.
- Un área de texto con nombre `text`, etiqueta "Texto:" y que también sea requerido.

Ejercicios 3

En esta sesión vamos a continuar con los ejercicios anteriores del editor de notas. Añadiremos los elementos necesarios para poder crear, editar, guardar y borrar notas, así como visualizarlas en un listado.

Ejercicio 1 - Modelo y Almacén de datos (0.8 puntos)

En este ejercicio vamos a crear el modelo de datos a utilizar y el almacén donde se van a guardar.

Nuestro modelo de datos llamado ' `Nota` ' tendrá cuatro campos:

- '*id*' de tipo '*int*'
- '*date*' de tipo '*date*' y formato '*c*'
- '*title*' de tipo '*string*'
- '*text*' de tipo '*string*'

Además deberemos definir las siguientes validaciones: los campos 'id', 'title' y 'text' serán requeridos, y para los campos 'title' y 'text' modificaremos el mensaje de error por defecto por "Introduzca un título/texto".

> Los modelos de datos se tienen que guardar por separado en la carpeta designada para ello (`app/model`), además en la aplicación se tendrá que indicar al inicio los modelos que tiene que cargar (revisar el apartado "Uso de modelos de datos en una aplicación").

A continuación crearemos un almacén de datos (*Data Store*), lo guardaremos en la variable '*storeNotas*' y le indicaremos que tiene que usar el modelo ' `Nota` ' (recuerda que para los modelos hay que indicar todo el espacio de nombres). Como proxy vamos a usar el **almacenamiento en local**, indicando como identificador '*misNotas-app-localstore*'. Además lo configuraremos para que se ordenen los datos de forma descendente (DESC) por fecha (campo '*date*') y que se carguen los datos del repositorio al inicio (`autoLoad: true`).

De forma temporal y para poder ver los resultados vamos a insertar datos en el *store*, añadiendo las siguientes líneas al mismo:

```
data: [
    { id: 1, date: new Date(), title: 'Test 1', text: 'texto de prueba' },
    { id: 2, date: new Date(), title: 'Test 2', text: 'texto de prueba' },
    { id: 3, date: new Date(), title: 'Test 3', text: 'texto de prueba' },
    { id: 4, date: new Date(), title: 'Test 4', text: 'texto de prueba' }
]
```

Ejercicio 2 - Listado (0.6 puntos)

En este ejercicio vamos a añadir el listado donde se visualizarán los datos. En primer luegar creamos el listado en la variable " `panelLista` " y le indicamos que utilice el *store* 'storeNotas' que hemos definido previamente.

En su sección `itemTpl` indicamos que muestre el campo `{title}` dentro de una capa tipo DIV con el estilo CSS "*list-item-title*", y que el campo `{text}` lo muestre a continuación en otra capa tipo DIV con el estilo CSS "*list-item-text*".

Este listado ('panelLista') lo tendremos que añadir a nuestro panel 'panelContenedorLista' en su sección `items` para poder visualizarlo.

Ahora tenemos que añadir esos estilos al fichero *app.css*. Para ambas clases ('*list-item-title*' y '*list-item-text*') definiremos los mismos estilos (ver código siguiente), salvo para el "*list-item-text*" que además tendrá el texto en color gris.

```
.list-item-title {  /* Igual para: list-item-text */
    float:left;
    width:100%;
    font-size:80%;
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
    /* Para list-item-text añadimos 'color: gray;' */
}
```

Ejercicio 3 - Crear y Editar notas (0.8 puntos)

Vamos a modificar la función `handler` del botón "Nueva nota" para que al pulsarlo, **antes de realizar la transición**, cree una nueva nota y la asigne al formulario. Para esto vamos a añadir una llamada a la función `crearNuevaNota()` (la cual crearemos de forma separada).

En la función `crearNuevaNota()` en primer lugar nos guardaremos la fecha actual `var now = new Date();` , y a continuación obtendremos el identificador único del registro a crear `var noteID = now.getTime();` (*con esta función transformamos la fecha en milisegundos, con lo*

que obtenemos un número que no se repite que podemos usar como ID). A continuación creamos un registro del modelo '*MisNotas.model.Nota*' y lo cargamos en nuestro formulario (`panelFormulario.setRecord( note );`).

Por último vamos a añadir la funcionalidad de editar las notas creadas. Para esto vamos hasta el 'panelLista', y definimos su función `onItemDisclosure` . Esta función recoge un parámetro (*record*) que tenemos que cargar en el 'panelFormulario' (`panelFormulario.setRecord( record );`). A continuación realizaremos una transición de tipo '*slide*' hacia la izquierda y con una duración de 1 segundo, para mostrar el 'panelFormulario'.

Ejercicio 4 - Guardar y Borrar notas (0.8 puntos)

En este último ejercicio vamos a añadir las acciones de guardar y borrar nota. En las funciones `handler` de los botones "Guardar" y "Borrar" añadiremos llamadas a las funciones `guardarNota()` y `borrarNota()` respectivamente. Las llamadas a estas funciones las deberemos de incluir antes de realizar la transición. A continuación definiremos las acciones a realizar en estas funciones (que estarán definidas de forma separada).

En la función `guardarNota()` realizaremos los siguientes pasos:

- En primer lugar tendremos que cargar los datos introducidos en el formulario (usaremos la función `getRecord()` sobre la variable que contiene el formulario), y a continuación actualizaremos la instancia del formulario (ver sección "Guardar los datos de un formulario").
- En segundo lugar comprobaremos si hay algún error de validación y mostraremos los errores (ver apartado "Comprobar validaciones" de la sección "Formularios").
- Una vez validado el registro obtenido procederemos a guardar los datos. Obtenemos el *store* usado por el listado (`panelLista.getStore()`) y añadimos el registro solo si este no está repetido (función `findRecord()`).
- Por último actualizamos el *store* (`sync()`), lo reordenamos por fecha y refrescamos el listado (`refresh()`).

La función `borrarNota()` es más sencilla:

- Obtenemos el registro actual del formulario (`getRecord()`) y el *store* usado por el listado (`getStore()`).

- A continuación comprobaremos si existe el registro actual en el *store* (usando su "id") y si es así lo eliminaremos (ver apartado "Eliminar registros" de la sección "*Data Store*").
- Por último actualizaremos los datos del *Store* (`sync()`) y refrescamos el listado (`refresh()`).