

Aspectos avanzados de Sencha Touch

Índice

1 Data Model.....	2
2 Data Store.....	3
3 Plantillas.....	6
4 Data Views.....	7
5 Listados.....	8
6 Formularios.....	12
7 Más información.....	21

1. Data Model

Los modelos de datos nos permiten representar objetos, junto con sus validaciones de formato y las relaciones que estos tienen con otros objetos. Por comparación podríamos pensar que un Data Model es como una clase con la definición de los datos que la componen y funciones para poder validar y trabajar con esos datos.

Para usarlos tenemos que crear una instancia de “**Ext.regModel**”, en la que primero se indica el nombre del modelo o identificador (en el ejemplo siguiente sería “Users”), y a continuación los campos que lo componen (fields):

```
Ext.regModel('Users', {
  idProperty: 'id',
  fields: [
    { 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' },
  ]
});
```

También es importante indicar el campo que se va a utilizar como identificador único (idProperty: 'id'). Es recomendable indicarlo siempre porque para algunos tipos de almacenamiento es necesario.

Como se puede ver en el ejemplo, los campos de la sección `fields` se definen primero por su nombre (`name`) y después por su tipo (`type`), que puede ser básicamente de cinco tipos: `string`, `int`, `float`, `boolean`, `date`. Además podemos usar algunos atributos opcionales, como el valor por defecto (`defaultValue`) o el tipo de fecha (`dateFormat`, ver “<http://docs.sencha.com/touch/1-1/#!/api/Date>”).

Validaciones

Los modelos de datos incluyen soporte para realizar validaciones, las cuales las deberemos de incluir dentro de la misma clase `regModel`, a continuación del campo “fields”. Continuando con el ejemplo anterior:

```
fields: [ ... ],
validations: [
  { field: 'id', type: 'presence' },
  { field: 'usuario', type: 'exclusion', list: ['Admin', 'Administrador'] },
  { field: 'usuario', type: 'format', matcher: /([a-z]+)[0-9]{2,3}/ },
  { field: 'nombre', type: 'length', min: 2 },
  { field: 'genero', type: 'inclusion', list: ['Masculino', 'Femenino'] }
]
```

Otra opción interesante es cambiar el mensaje de error que se produciría si una validación no es correcta, esto lo podemos hacer definiendo el valor de la propiedad `message`:

```
{ 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()`, esta opción la veremos más en detalle en la sección de formularios.

Crear de un registro

Para crear registros utilizamos el constructor “`Ext.ModelMgr.create`”:

```
var instance = Ext.ModelMgr.create({
    id: 1,
    usuario: 'javierGallego',
    nombre: 'Antonio Javier Gallego Sánchez',
    genero: 'Male',
    fechaRegistro: new Date()
}, 'Users');
```

Más información

Para más información sobre validaciones: [“<http://docs.sencha.com/touch/1-1/#!/api/Ext.data.validations>”](http://docs.sencha.com/touch/1-1/#!/api/Ext.data.validations).

Los modelos de datos también soportan la creación de relaciones con otros modelos de datos, del tipo “has-many” y “belongs-to”. Para más información consultar la API: [“<http://docs.sencha.com/touch/1-1/#!/api/Ext.data.Model>”](http://docs.sencha.com/touch/1-1/#!/api/Ext.data.Model).

2. Data Store

Los almacenes de datos se utilizan para encapsular los datos (de un modelo dado) en la caché del cliente. Estos datos se cargan y escriben utilizando un proxy. Además disponen de funciones para ordenar, filtrar y consultar los datos. Crear un almacén de datos es fácil, simplemente tenemos que indicarle el modelo de datos a usar y el proxy para cargar y guardar los datos. Dependiendo del proxy que utilicemos podremos almacenar los datos de forma local o de forma remota:

Almacenamiento en local

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. 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'`”.

Para indicar que el proxy ha de utilizar el almacenamiento en local simplemente

tendremos que indicar el tipo (*localStorage*) y un identificador (usado como clave para guardar los datos). En el siguiente ejemplo se utiliza el modelo de datos “Users” que hemos creado en la sección anterior:

```
var myStore = new Ext.data.Store({
    model: 'Users',
    autoLoad: true,
    proxy: {
        type: 'localStorage',
        id: 'notes-app-localstore'
    }
});
```

Además hemos añadido la propiedad `autoLoad: true`, muy importante para que se carguen los datos al inicio. Si no lo hiciéramos así el Store inicialmente estaría vacío.

Una forma alternativa de instanciar un Store es: `Ext.regStore('myStore', { ... })`; de forma similar a como lo hacíamos con los Data Models.

Almacenamiento en remoto

Para configurar un proxy para que utilice **JSON** lo podemos hacer de la forma:

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

En este ejemplo el proxy utiliza AJAX para cargar los datos desde la dirección “./users.json”. La propiedad “reader” indica al proxy como debe decodificar la respuesta que obtiene del servidor a través de la consulta AJAX. En este caso espera que devuelva un objeto JSON que contiene un array de objetos bajo la clave 'users'. Para más información sobre JSON: <http://docs.sencha.com/touch/1-1/#!/api/Ext.data.JsonReader>

Añadir datos

Es muy sencillo añadir datos directamente a un Store, solo tenemos insertarlos como un array a través de la propiedad “data”. Suponiendo que el modelo “Users” solo tuviera dos campos (firstName, lastName), podríamos añadir datos de la forma:

```
var myStore = new Ext.data.Store({
    model: 'Users',
    data: [
        {firstName: 'Ed',    lastName: 'Spencer'},
        {firstName: 'Tommy', lastName: 'Maintz'},
        {firstName: 'Aaron', lastName: 'Conran'},
    ]
});
```

```
        {firstName: 'Jamie', lastName: 'Avins'}  
    ]  
});
```

O también podemos añadir datos posteriormente llamando a la función “**add**” del objeto:

```
myStore.add({firstName: 'Ed',    lastName: 'Spencer'},  
            {firstName: 'Tommy', lastName: 'Maintz'});
```

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).

```
var myStore = new Ext.data.Store({  
    model: 'Users',  
    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()` del *Store*. En el 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, 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, y por último sincronizar los datos para que se visualicen, de la forma:

```
var store = miListado.getStore();
if( store.findRecord('id', registro.data.id) )
{
    store.remove( registro );
}
store.sync();
miListado.refresh();
```

Más información

Para más información sobre los almacenes de datos consultar: [“http://docs.sencha.com/touch/1-1/#!/api/Ext.data.Store”](http://docs.sencha.com/touch/1-1/#!/api/Ext.data.Store).

3. 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.

Auto-procesado de arrays

Para procesar 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.
- 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”
- Podemos usar la variable especial `{#}` para obtener el índice actual del array.

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 “data.kids” de las siguientes formas:

```
var myTpl = new Ext.XTemplate(
```

```
'<tpl for=".">',
'<p>{#}. {name}</p>',
'</tpl>');
myTpl.overwrite(panel.body, myData.kids);

// Otra opción:
var myTpl = new Ext.XTemplate(
'<tpl for="kids">',
'<p>{#}. {name}</p>',
'</tpl>');
myTpl.overwrite(panel.body, myData);
```

Si el array solo contiene valores (en nuestro objeto de ejemplo, 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(panel.body, myData);
```

Condiciones

Para introducir condiciones en las plantillas se utiliza la etiqueta '`<tpl if="condicion"> plantilla </tpl>`'. Pero hemos de tener en cuenta que: si utilizamos las “comillas”, deberemos escribirlas codificadas ("), y que no existe el operador “else”, si lo necesitamos deberemos de utilizar un condición “if” adicional.

Ejemplos:

```
<tpl if="age > 1 && age < 10">Child</tpl>
<tpl if="age >= 10 && age < 18">Teenager</tpl>
<tpl if="name == &quot;Tommy&quot;">Hello!</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()`” que ya hemos usado en los ejemplos anteriores. O usar la propiedades “`tpl`” junto con “`data`”, de la forma:

```
MyApp.views.viewport = new Ext.Panel({
  data: myData,
  tpl: myTpl
});
```

Más información

Para más información sobre las plantillas (operaciones matemáticas, ejecución de código en línea, variables especiales, funciones, etc.) consultar directamente la API de Sencha Touch: <http://docs.sencha.com/ext-js/4-0/#/api/Ext.XTemplate>

4. 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 “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).

Continuando con los ejemplos anteriores, vamos a crear un DataView para mostrar el contenido de “myStore” por medio de plantilla “myTpl”:

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

Esta “vista de datos” tendríamos que añadirla a un panel para poder visualizarlo en nuestra aplicación:

```
MyApp.views.viewport = new Ext.Panel({
    items: [ myDataView ]
});
```

5. Listados

Permiten mostrar datos usando una plantilla por defecto de tipo lista. Estos datos se obtienen directamente de un “store” (Ext.data.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 tendremos que definir el “store” que queremos utilizar, y la plantilla para cada uno de los elementos con “itemTpl”:

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

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, y además, utilizaremos como separador para el salto de línea el símbolo de unión “+” y no la coma “,”.

Una vez creado el listado solo nos faltará añadirlo a un panel para poder visualizarlo:


```
MyApp.views.viewport = new Ext.Panel({
    items: [ myList ]
});
```

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 la propiedad `getStore`:

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

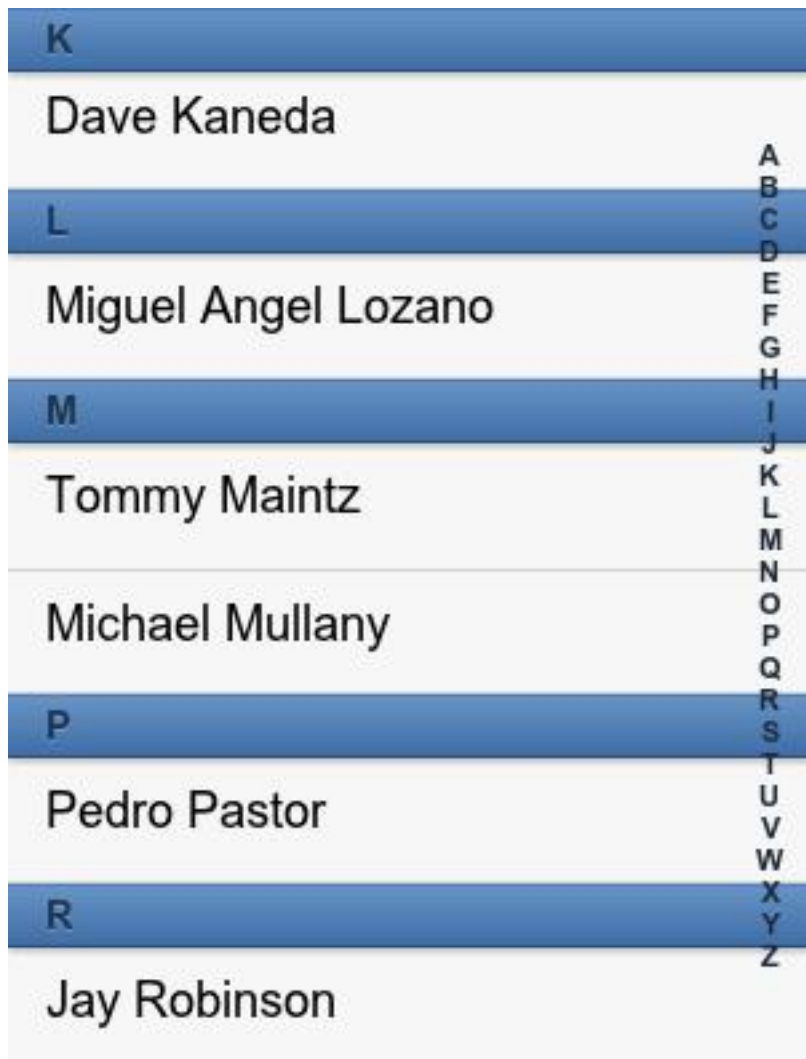
Actualizar datos

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.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” y opcionalmente podremos indicar que se muestre una barra lateral de navegación “indexBar: true”.



Pero para que esta propiedad funcione dentro del “store” tenemos que indicar 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, el código quedaría:

```
var myStore = new Ext.data.Store({
  model: 'Users',
  sorters: 'apellido',
  getGroupString: function(instance) {
```

```
        return instance.get('apellido')[0];
    }
});

var myList = new Ext.List({
    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(record, index) { alert( "tap on" + index ); }
}
```

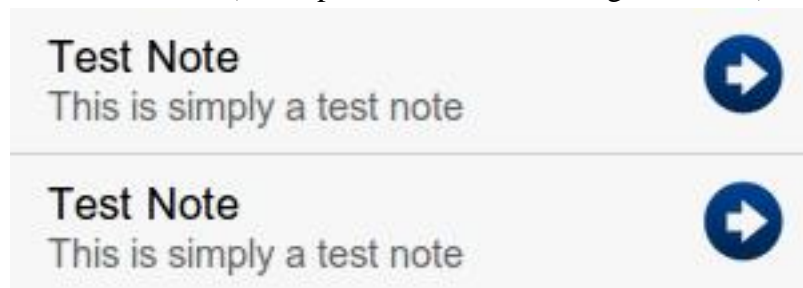
Donde el parámetro *record* representa el objeto sobre el que se ha pulsado. Este valor lo podríamos aprovechar para cargarlo directamente en un formulario o realizar alguna operación con él.

- **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(record, index){ alert( "doubletap 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.

- **onItemDisclosure**: Boolean / Función - esta propiedad admite varios 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 se añadirá también el icono en cada elemento, y además 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`). Un ejemplo de código:

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

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.

Por ejemplo, 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 )
{
    MyApp.views.myFormPanel.load( record );
    MyApp.views.viewport.setActiveItem(
        'myFormPanel',
        { type: 'slide', direction: 'left' });
}
```

6. Formularios

Para crear formularios utilizamos el constructor “`Ext.form.FormPanel`”, que se comporta exactamente igual que un panel, pero en el que podemos añadir fácilmente campos insertándolos en el array “`items`”:

```
var noteEditor = new Ext.form.FormPanel({
    id: 'noteEditor',
    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 se utiliza para cargar y enviar los datos del formulario (como veremos más adelante), y la etiqueta se mostrará visualmente en la parte izquierda de cada campo. El valor de todos los campos se encuentra en su atributo “`value`”, podemos utilizarlo para especificar un valor inicial.

Los principales tipos de campos que podemos utilizar son los siguientes (indicados según su nombre “xtype”):

- **textfield**: campo de texto (el código utilizado lo podemos ver al principio de esta sección):

Title*	Campo de prueba
--------	-----------------

- **textareafield**: área de texto (el código utilizado lo podemos ver al principio de esta sección):

Narrative	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'” :

Password:
-----------	-------

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

Website	http://www.ua.es
---------	------------------

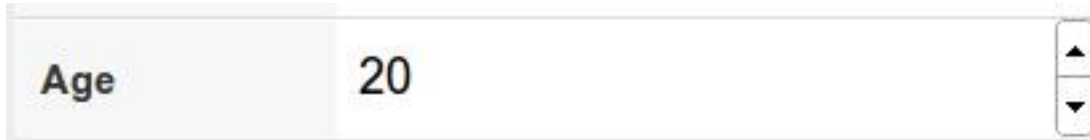
- **emailfield**: campo de texto para introducir e-mails, incluye validación automática:

Email	jgallego@dlsi.ua.es
-------	---------------------

- **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:

Activado:	
-----------	---

- **numberfield**: campo numérico, permite introducir el número manualmente o mediante las flechas laterales. Inicialmente no contiene ningún valor, 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}”:

April		1988
May	1	1989
June	2	1990

- **fieldset:** Este elemento en realidad no es un campo de datos, sino un contenedor. No añade ninguna funcionalidad, simplemente poner un título (opcional), y agrupar 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:

About Me	
First Name	Antonio Javier
Last Name	Gallego Sánchez

- **selectfield:** campo desplegable para seleccionar entre una lista de valores. Los valores disponibles se indican mediante la propiedad “options” como un array:

```
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, y todos ellos deben de tener el mismo nombre “name” para poder ser agrupados (muy importante para poder recoger los datos correctamente). Además podemos utilizar la propiedad “checked: true” para que aparezcan marcados inicialmente:

```

items: [
  {
    xtype: 'checkboxfield',
    name: 'check_color',
    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:



Select - Colores	
Red	<input checked="" type="checkbox"/>
Green	<input checked="" type="checkbox"/>
Blue	<input type="checkbox"/>

- **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 ítem independiente, y todos ellos deben de tener el mismo nombre “name” para poder ser agrupados (muy importante para 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',  
    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:

Radio - Colores	
Red	<input checked="" type="radio"/>
Green	<input type="radio"/>
Blue	<input type="radio"/>

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” pero quitando el sufijo “field”.

Cargar datos en un formulario

Para insertar datos en un formulario utilizamos el método “load(data)” de la clase “FormPanel”. Por ejemplo, en el formulario que hemos creado al principio de esta sección:

```
noteEditor.load( note );
```

Este método recibe como parámetro una instancia de un modelo de datos (ver sección Data Model), y cargará solamente los campos cuyos nombre coincidan. En el ejemplo del inicio de esta sección tenemos dos campos cuyo nombre son “name: 'title'” y “name: 'text'”, si cargamos una instancia de un modelo de datos como el siguiente, solamente cargaría los dos campos que coinciden.

```
Ext.regModel('Note', {
    idProperty: 'id',
    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.ModelMgr.create” para crear instancias de un modelo de datos. Por lo que, por ejemplo, en la función “handler” de un botón, podríamos crear una instancia de este modelo y cargarlo en el formulario, de la forma:

```

{
  text: 'Cargar datos',
  ui: 'action',
  handler: function () {
    var now = new Date();
    var noteId = now.getTime();
    var note = Ext.ModelMgr.create(
      {
        id: noteId,
        date: now,
        title: 'titulo',
        text: 'texto' },
      'Note');
    noteEditor.load( note );
  }
}

```

Guardar los datos de un formulario

En primer lugar llamaremos al método `getRecord()` del formulario para obtener un objeto con los datos introducidos. A continuación deberemos llamar a la función `updateRecord(objeto)` del mismo formulario para transferir estos valores a su instancia interna del modelo (tenemos que diferenciar entre los datos introducidos en los campos del navegador y el objeto interno de esa instancia). Si no llamásemos a este método la instancia del formulario estaría vacía. Después de esto deberíamos de validar los errores en los datos introducidos (ver siguiente apartado) y por último guardar los datos.

```

var currentNote = noteEditor.getRecord();
noteEditor.updateRecord(currentNote);
// Realizar validaciones (ver siguiente apartado)
// Guardar los datos

```

Si tenemos una instancia del almacén de datos creada (ver sección Data Store) podemos añadir los datos simplemente llamando a la función `add`:

```

notesStore.add(currentNote);

```

Más comúnmente tendremos nuestro almacén de datos asociado con algún elemento que nos permita visualizar los datos (como un listado o un Data View, ver secciones correspondientes). En el ejemplo del listado deberíamos de obtener la instancia del almacén de datos llamando a su método `getStore()` y posteriormente añadir los datos:

```

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

```

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

```

var notesStore = notesList.getStore();

```

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

Una vez añadidos los datos, y por terminar el ejemplo del listado, tendremos que sincronizar el Store, ordenarlo (si fuese necesario) y por último actualizar la vista del listado:

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

Comprobar validaciones

Para comprobar las validaciones de un formulario lo tendremos que hacer de forma manual llamando a la función `validate()`. La cual comprobará que se cumplen todas las validaciones que hayamos creado para el modelo de datos. Si continuamos con el ejemplo del modelo de datos “Note” podríamos añadir que los campos `id`, `title` y `narrative` son requeridos.

```
Ext.regModel('Note', {
    idProperty: 'id',
    fields: [
        { name: 'id', type: 'int' },
        { name: 'date', type: 'date', dateFormat: 'c' },
        { name: 'title', type: 'string' },
        { name: 'narrative', type: 'string' }
    ],
    validations: [
        { type: 'presence', field: 'id' },
        { type: 'presence', field: 'title',
            message: 'Introduzca un título para esta nota' },
        { type: 'presence', field: 'narrative',
            message: 'Introduzca un texto para esta nota' }
    ]
});
```

En primer lugar, para validar un formulario tenemos que cargar los datos introducidos en el formulario (con la función `getRecord()`) y a continuación actualizar la instancia del formulario con esos datos. Si no lo hacemos así al obtener los datos del formulario aparecería vacío.

Para validar utilizamos la función `validate()`, la cual devuelve un objeto tipo `Errors`. Podemos usar la función `isValid()` del objeto `Errors` para comprobar si ha habido algún error. En este caso tendremos que mostrar un aviso y no realizar ninguna acción más. Para mostrar el aviso usaremos un `MessageBox` (ver sección correspondiente). Además, dado que pueden haber varios errores (guardados en el array `items` del objeto `Errors`), vamos a iterar por los elementos de este array usando `Ext.each()` y concatenarlos:

```
var currentNote = noteEditor.getRecord();
```

```
noteEditor.updateRecord(currentNote);
var errors = currentNote.validate();
if (!errors.isValid())
{
    var message="";
    Ext.each(errors.items,function(rec,i){
        message += rec.message+"
";
    });
    Ext.Msg.alert("Validate", message, Ext.emptyFn);
    return; // Terminamos si hay errores
}
// Almacenar datos...
```

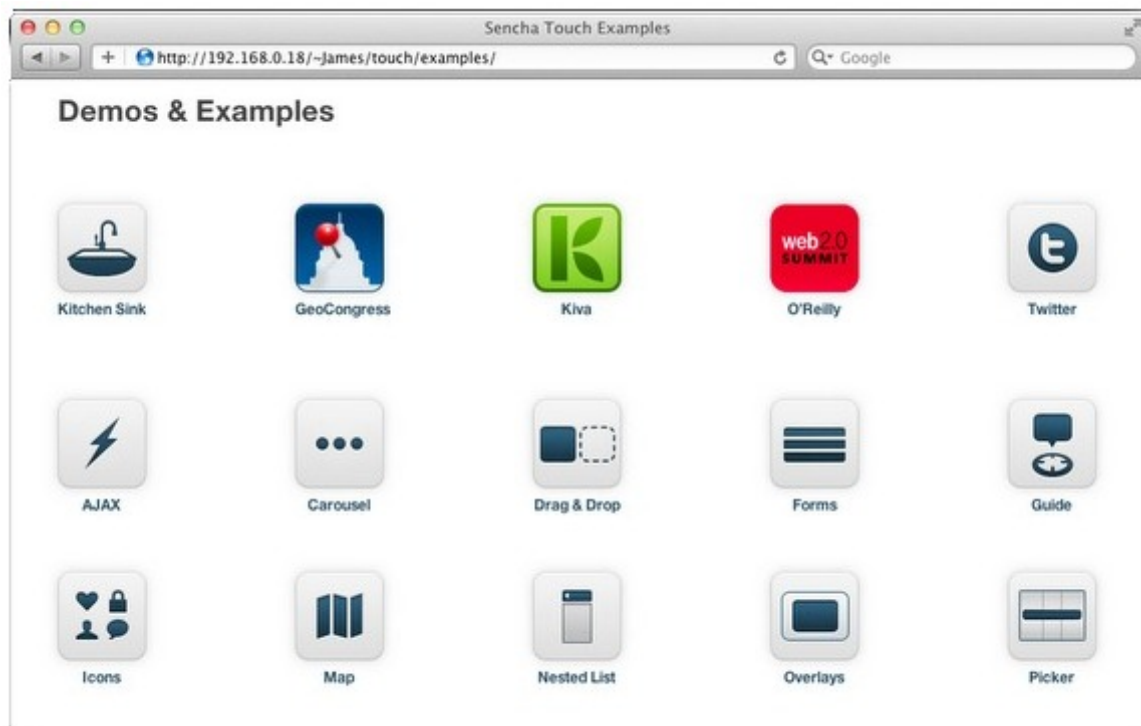
También podríamos haber creado un bucle para iterar entre los elementos del array, o haber llamado a la función `errors.getByField('title')[0].message` para obtener directamente el mensaje de error de un campo en concreto.

7. 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/1-1/>
 - <http://www.sencha.com/learn/touch/>
 - También disponible de forma local accediendo en la dirección “/docs” de tu SDK, por ejemplo:
<http://localhost/touch/docs/>
- Los **foros** en la página web de Sencha Touch:
<http://www.sencha.com/forum/forumdisplay.php?56-Sencha-Touch-Forums>

Además en la carpeta “touch/examples” podemos encontrar aplicaciones de ejemplo. Estos ejemplos también están disponibles de forma online en “<http://localhost/touch/examples/>”:



En GitHub también podemos encontrar numerosas aplicaciones de ejemplo bien documentadas, estas las podemos encontrar en la dirección “<https://github.com/senchalearn/>”.

