Elements of ListView

Something smooth this way comes

Thomas D Wilkinson, Appcelerator, Inc.
@CapnAjax

Abstract

ListView is a bit daunting to anybody who hasn't used it before. Its structure is completely different from TableView or any other Titanium UI element. However, anybody who's used TableView knows that when a table gets too large, it slows down, it consumes memory, it scrolls in fits and starts, and the device becomes less responsive.

ListView was designed to solve this problem.

The biggest mental hurdle with ListView is that you no longer create the elements the ListView displays, ListView does that for you. Instead of views you have templates, and you provide the data necessary to fill these templates. The ListView controls the views, and recycles them as they roll off the screen, so you never have more views than are visible.

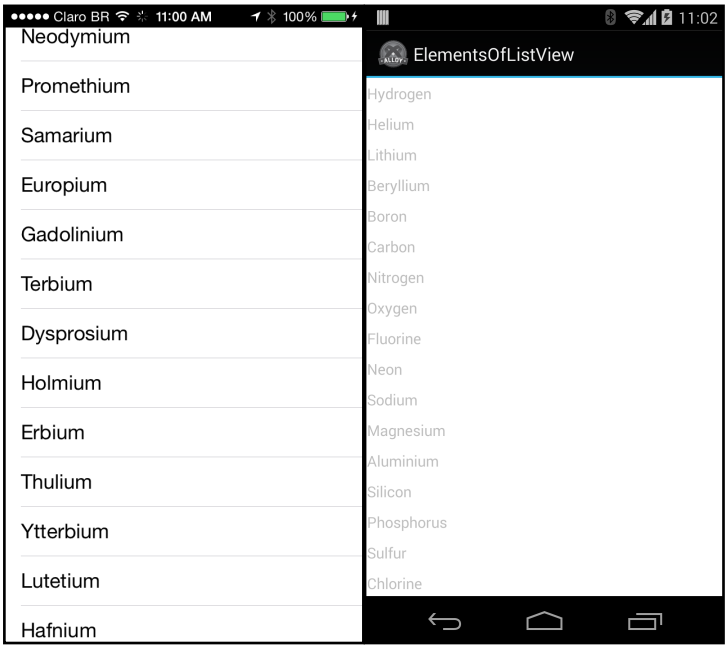
This white paper describes how to handle the data and templates. We start with a basic list using the default template, creating your own template, feeding the ListView with live data, and event handling. When you are done, you'll be a master of one of the most powerful features of the Titanium SDK.

The example project used in this document is in bitbucket at this address:
https://github.com/appcelerator-services/ElementsOfListView/
In this tutorial, you will follow the development of a real-world cross-platform application that is centered around a ListView. For your reference, each chapter has a branch in this repository, check out the branch that corresponds to each chapter to follow along.

The Hello World ListView

➡Git Branch: 01-HelloWorld

We start with the most basic ListView possible — a list of words.



The default ListView on iOS and Android

In your view xml, the ListView looks like this.

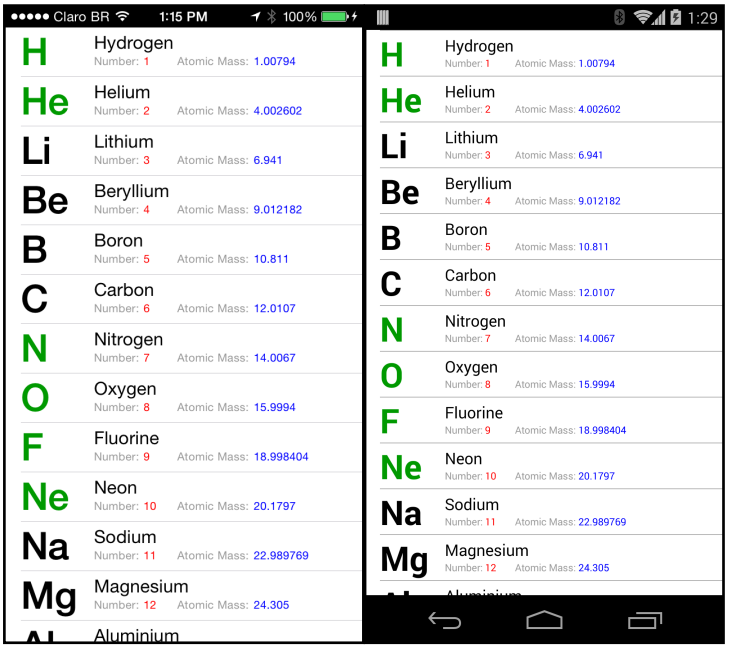
```
<ListView id="elementsList">
    <ListSection name="elements">
        <ListItem title="Hydrogen"/>
        <ListItem title="Helium"/>
        <ListItem title="Lithium"/>
            ...
    </ListSection>
</ListView>
```

At this point, it doesn't look any different from any other Alloy element. It's a `ListView`, with a `ListSection` that contains a series of `ListItem`s. These three elements alone implement a ListView that scrolls. The appearance of these lists is highly platform-specific.

Customizing the Layout with Templates

➡Git Branch: 02-ListTemplate

In many cases the default look and feel above is sufficient, but if we want to customize the appearance of `ListItem`s or create a consistent look and feel across platforms, we will need to build an `ItemTemplate`.



A ListView with a customized template

An `ItemTemplate` is looks very much like a code snippet of an Alloy view; it's a view with elements inside it. There is one notable difference though - the addition of a `bindId`. This `bindId` is what is used to connect the list data with components.

```
<ListView id="elementsList">
    <Templates>
        <ItemTemplate name="elementTemplate">
            <Label bindId="symbol" id="symbol" />
            <View id="atomProperties">
                <Label bindId="name" id="name" />
                <View id="secondLine">
                    <Label class="line2 fieldLabel"
                           text="Number: " />
                    <Label class="line2" bindId="number"
                           id="number" />
                    <Label class="line2 fieldLabel"
                           text="Atomic Mass: " />
                     <Label class="line2" bindId="mass"
                           id="mass" />
```

```
                    </View>
                </View>
            </ItemTemplate>
        </Templates>
        <ListSection name="elements"/>
    </ListView>
```

The elements in the template can be styled the same way as other elements, using the .tss style sheet using the id or class fields it identify elements for styling.

Once you have created your template, you must specify which template to use, or else your `ListView` will still look like it did in the previous section. In the style for the `ListView`, add the name of the template as `defaultItemTemplate`:

```
    "#elementsList": {
            backgroundColor:"white",
            defaultItemTemplate:"elementTemplate",
            separatorColor: "#999"
},
```

In cases where you have multiple templates, you may specify which template to use by setting the `template` attribute of the `ListItem`.

To fill this template, the `ListItem`s need to contain some additional information.

```
    <ListItem symbol:text="Ti" name:text="Titanium" number:text="22"
            mass:text="47.867"/>
```

Each attribute in this `ListItem` refers to a `bindId` in the template. So what does `:text` mean? Each item in this example template that has a `bindId` is a `Label`. The text of a label is the text attribute of the Label element.

```
    <Label text="Titanium" />
```

The `ListItem` is not limited to setting text of labels. It can set the value of any attribute of any Ti.UI component. For example, in the screen shots above, you may notice that gasses have a symbol coloured green. The `ListItem` for Hydrogen sets `symbol:color` to green.

```
    <ListItem symbol:text="H" symbol:color="#090"
            name:text="Hydrogen" number:text="1"
            mass:text="1.00794"/>
```

In fact, you can set values on more than just `Label`s. For example, if you wanted a `View` to have a shaded background you can give the view a `bindId` (say, `wrapperView`) in the template, and, in the `ListItem` set the attribute `wrapperView:backgroundColor="#e8e8ff"`.

Updating Content

➡Git Branch: 03-LiveData

The careful reader will note that in the example above, the `ListItem`s are placed directly in the view xml. Besides that this does a poor job of separating data from presentation, they don't represent live data. We need to update the list programmatically.

`ListItem`s are not views like `TableViewRow`. The actual view that displays the data in a `ListItem` are transient, that is, they can be created and destroyed at any time as the user scrolls. We can only influence the `ListItem` via the data set we feed into the `ListView`.

Let's load these elements into the `ListView`:

```
elements.table = [
   {“name”:”Hydrogen","number":1,"symbol":"H","mass":1.00794},
   {"name":"Helium","number":2,"symbol":"He","mass":4.002602},
   {“name”:”Lithium”,”number”:3,”symbol”:"Li","mass":6.941},
   ...
]
```

This list has to be massaged a bit for the `ListView` to work with it. The format is pretty simple; it's the `bindId` of the view in the template mapped to all the properties of that view to set.

```
var items = _.map(elements.table, function(element) {
        return {
            symbol: {text: element.symbol},
            name: {text: element.name},
            number: {text: element.number},
            mass: {text: element.mass}
        };});
```

And finally to add it to the list, you `setItems` in the list section.

```
$.elementsList.sections[0].setItems(items);
```

Let's turn up the heat!

In the example above, we only set the text of labels, all symbols appear black. It's easy enough to see how to colour them, simply add `color` as a property, ie `symbol: {text: element.symbol, color: “#090”}`, but now we want to update the colors in real time.

In this example, I am adding a slider to the screen to change the temperature. This will cause elements to change between solid, liquid and gas. `ListView` does not add instrumentation to

its dataset to change on the fly, I have to check the data items out, make the changes and check them in.

```
var changeColor = function(itemIndex, color) {
    var listSection = $.elementsList.sections[0];
    var listItem = listSection.getItemAt(itemIndex);
    listItem.symbol.color = color;
    listSection.updateItemAt(itemIndex, listItem);
};
```

There is an important hazard to this however - this is not a cheap process. In this example, when I updated 118 elements one-by-one every time the slider fires a change event, I created a 10-15 seconds of backlog of events on my fastest Android device (HTC One M8).

By throttling the updates to 250ms, and only updating the elements that change, the lag was usually less than a second, depending on how many elements I had to update.

```
var changeColor = function(itemIndex, color) {
   var listSection = $.elementsList.sections[0];
   var listItem = listSection.getItemAt(itemIndex);
   if(listItem.symbol.color != color) {
      listItem.symbol.color = color;
      listSection.updateItemAt(itemIndex, listItem);
   }
};
```

However, by completely replacing the data in the list with setData, the lag time dropped to 100ms on my Android device, and usually less than 10ms on the iPhone 5S.

For this reason, I recommend replacing the entire list data set instead of updating piecemeal if there are more than a couple items to update.

In the example code, I have included comments to describe the three algorithms: updating all items, updating only changed, and then replacing the data set. Enable and disable them by changing `if(false)` to `true` and vice versa in the `changeTemperatureAction` method to see the performance characteristics of each algorithm.

The Marker Event

➡Git Branch: 04-MarkerEvent

Let's make this app even faster.

There are many cases where the data available to display will be extremely large or even infinite, but loading them is slow and it doesn't make sense to load them all at once. Normally, in a TableView, you would monitor scroll events, calculate your position, and at some point decide it's time to load more data.

`ListView` doesn't have scroll events, but it does have a marker. The marker is like a tripwire, the first time the user scrolls past it, the ListView fires an event.

In the previous section, the elements data was in a JavaScript file. In this example, I'm going to break it out into several JSON files, and load them one at a time. Even though I'm not loading from a web service, there's still a good use case for splitting the load. Parsing JSON can be expensive if the data gets too long. Right now the elements file isn't very big, but I could potentially add information about elements in the future that could blow the file up. To keep the startup quick, we want to defer loading as much information as we can until we absolutely need it.

Set up the marker event.

```
<ListView id="elementsList" onMarker="markerReached">
    ...
</ListView>
```

Set up the trigger point for the marker.

```
$.elementsList.setMarker({sectionIndex:0,itemIndex:15});
```

Note this event will only fire once each time a marker is set, so in the event handler, you must set the marker to ahead a little.

```
var markerReached = function() {
    addData();
    $.elementsList.setMarker({
        sectionIndex:0,
        itemIndex: ($.elementsList.sections[0].items.length - 10)
    });
};
```

In earlier sections of this tutorial, we used `setItems()` to add data to the table. When `setItems()` increases the size of the list, the iPhone will use an animation to grow the list. When you just want new items on the bottom, it looks odd. In this case it's better to use the `appendItems()` method to add the new items on the bottom.

```
var addData = function() {
   ...
   $.elementsList.sections[0].appendItems(newData);
};
```

Searching in the table

➡Git Branch: 05-Search

On the surface this is very easy to do. Add `searchableText` to the `ListItem` and add a `SearchBar` to the `ListView`, and everything you need is done for you.

In the view:

```
<ListView id="elementsList" onMarker="markerReached">
    <Templates>
        ...
   </Templates>
   <SearchBar id="searchBar" />
   <ListSection name="elements" />

</ListView>
```

And add searchable text to the list item. Note that this is inside and object called `properties`. The properties object is for ListItem properties while the other fields are for binding to template items.

```
var preprocessForListView = function(rawElements) {
    return _.map(rawElements, function(element) {
        return {
            properties: {
                searchableText: element.name + ' ' +
                        element.symbol + ' ' +
                        element.number.toString()
            },
            symbol: {text: element.symbol, ... },
            name: {text: element.name},
            number: {text: element.number.toString()},
            mass: {text: element.mass.toString()}
        };
    });
};
```

However, there are some limitations to this. The first limitation is the annoying Android behavior of always automatically focusing on any available text fields when a window opens, which in this case is the Search bar. The second is that you can only search things that are already loaded in the list.

There is no really elegant way to prevent Android from automatically focusing on a text field. There are many kluges however.

The example sources show the Android version using a search bar that is outside the `ListView`. To get around Android's focus issue, we're setting up the `SearchBar` only when the user is actually searching. The critical element here is `$.elementsList.searchText`. This sets the search text when the `ListView`'s search view isn't defined.

For the second issue, our example app loads all the remaining items into the list as the user clicks into the `SearchBar`. This is only one of a number of acceptable ways to handle this issue. If loading the entire dataset is impractical, one can attach events to the `SearchBar` and use them to send queries to the web service so the additional items can be added to the list. You can do this automatically, or allow the user to click a button to say "continue this search on the server". Whatever makes the most sense depends on the circumstances of the app.

Clicking on Rows

➡Git Branch: 06-EventsOnRowsAndElements

Unlike TableViews, there are no events on ListItems. You must capture itemclick events in the ListView, then find the item the user clicked on. There are two ways of doing this: by sectionIndex and itemIndex, or by itemId.

The previous section actually has an itemclick which used the idemId to handle the Android search feature. The ListItem that displayed the message “Click to search” was given the itemId “clickToSearch”.

```
var clickToSearch = {
   template: Titanium.UI.LIST_ITEM_TEMPLATE_DEFAULT,
    properties: {
        height: 50,
        title: "Click to search",
        color: "#009",
        itemId: "clickToSearch"
    }
};
```

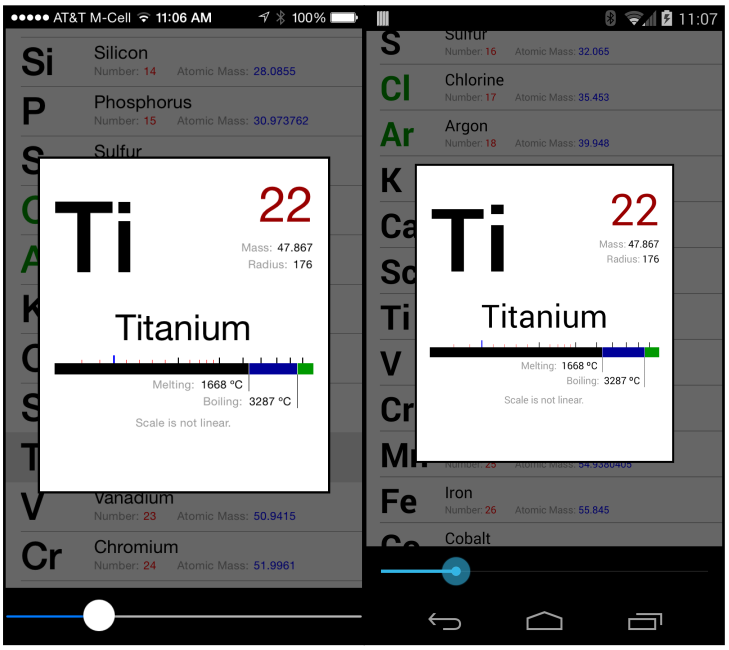
Then the handler for the itemclick event checked the itemId of the event.

```
$.elementsList.addEventListener('itemclick', function(e) {

    if(!$.searchBar.visible && "clickToSearch" == e.itemId) {
        // start search
        ...
    }
});
```

Events on Elements

In this section, we're going to add two new features to the app. When I click on a ListItem, I will show the Wikipedia page for the app, but when I click on the element symbol, I will show a "quick-view" detail of the element in an overlay.



The QuickView feature on iOS and Android

To support the Wikipedia feature, I can use the itemclick described in the previous section, but for the quick view feature, I need to capture the click event on the symbol.

As we already covered the itemclick event, I'm not going to go into create detail on the Wikipedia feature. However, the quick-view feature requires some attention. To make this work we need to do three things:

1. Determine that we clicked on the element symbol,
2. Prevent the symbol's click from trigging an itemclick event,
3. Still be able to capture itemclick events that aren't on the symbol.

The views that display the ListItems only really exist when they are on the screen. They are only created when they scroll into view and they disappear when they scroll out of view. This is why ListView is so fast, but it also means ListView cannot give out references to the views of ListItem. However, in the template we can set event attributes, such as onClick like this:

```
<ItemTemplate name="elementTemplate" >
    <Label bindId="symbol" id="symbol" onClick="symbolClick" />
    ...
</ItemTemplate>
```

One point to note though, handler must be declared with a **Function Declaration**, not a **Function Expression** or the handler will never be called.

```
// Function Declaration
function symbolClick(e) {
    ... // works
};

// Function Expression
var symbolClick = function(e) {
    ... // fails quietly
};
```

For for details on what **Function Declarations** vs **Function Expressions** are, go to:
http://kangax.github.io/nfe/

This takes care of the first problem. We have captured a click on a symbol. Now we have to prevent the click on the symbol from triggering the itemclick without disabling the itemclick altogether.

We can't use cancelBubble to stop the click from becoming and itemclick because those a two different events. There are, however, a few things we can do.

One way is to adjust the template and put a click event on a wrapper view. Then we can have the handler for that wrapper's click event do what we were originally using the itemClick to do, which is, in this case, launching wikipedia.

Another way is to drop a flag. The click is processed before the itemclick, we we can simply set a variable to say the click was just fired, and the itemclick checks that variable before doing its own work.

```
// the flag
var symbolClicked = false;

// the click handler
function symbolClick(e) {
    ...
    symbolClicked = true;
};

// the itemclick handler
var elementClick = function(e) {
    if(symbolClicked) {
        symbolClicked = false;
        return;
    }
    ...
```

```
};
```

The event `e` also carries the itemId of the `ListItem` as a convenience.

Capturing events this way can give you access to the actual view objects that display the ListItem's content.

CAUTION: The views that display the `ListItem`s get recycled as the user scrolls through the list. Manipulating views directly can lead to unintended consequences. For example, setting a view to `visible=false` to make a label disappear can also make other labels in the list disappear. Therefore, it is important that, when changing the content of a `ListItem`, you only do so by changing the the data you feed it.

For example:

```
// breaks things
function symbolClick(e) {
    e.source.visible = false;
}

// safe
function symbolClick(e) {
    // get the itemId, which is conveniently included in the
    // click event
    var itemId = e.itemId;
    // then scan the list to get the correct data item
    var items = $.elementsList.sections[0].getItems();
    for(var i = 0; i < items.length; i++) {
        if(items[i].properties.itemId == itemId) {
            // found the item, now update it
            items[i].symbol.visible = false;
            $.elementsList.section[0].updateItemAt(i, items[i]);
            break;
        }
    }
}
```

In conclusion

The final warning in the previous section is actually a glimpse into the very source of the ListItem's power. By constantly recycling its constituent views, the ListView saves a lot of trips over the the Titanium native bridge and keeps a lot of objects out of memory. This is what makes ListView scroll smoothly. Also, by taking over the object creation, it's much easier to translate data into content and put it on the screen fast.

The troubles are obvious, events and some manipulations can be more effort than if you just had the views at your disposal. However, these troubles are easily outweighed by the ease of getting great performance and small memory footprints with limitless datasets. The ListView represents a strong separation of model view and controller, and the rewards are clear to any programmer to embrace it.

Code strong.