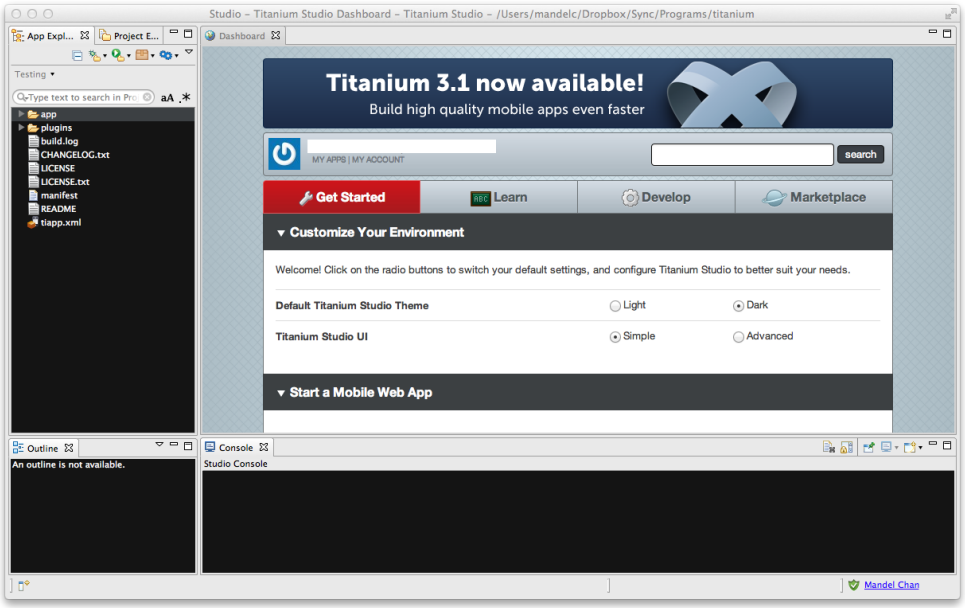
SIGIS'13: Building an App like Building a Website with Titanium Alloy

Welcome to the Mobile App Workshop!! Today, we will introduce an exciting software development kit (SDK) called **Titanium**, which allows programmers / beginners to develop mobile apps on both iOS and Android environments. **Both Mac and Windows versions can be freely downloaded** from the official website of Titanium, meaning that you can still practice Titanium programming even if you are more familiar with the Windows environment.

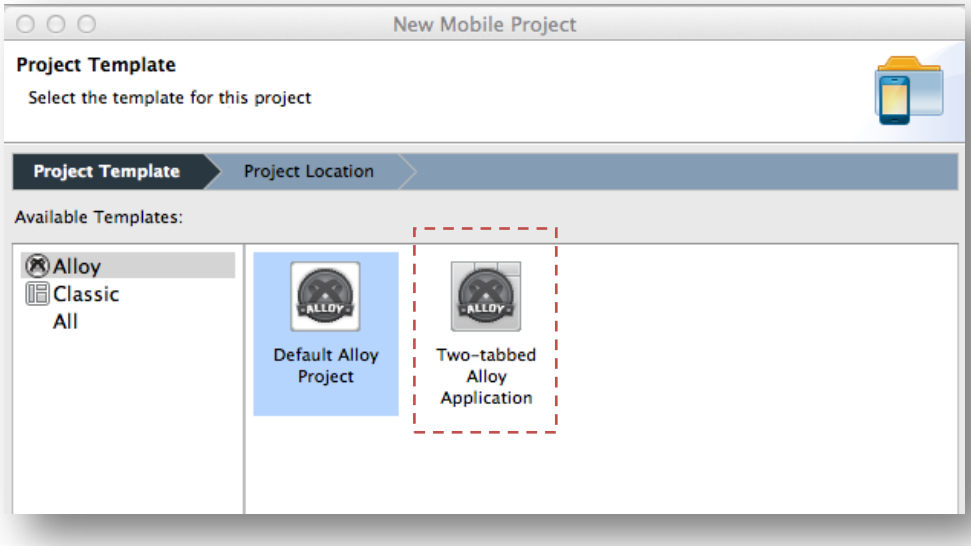
Titanium Official Website: http://www.appcelerator.com/

Titanium Studio

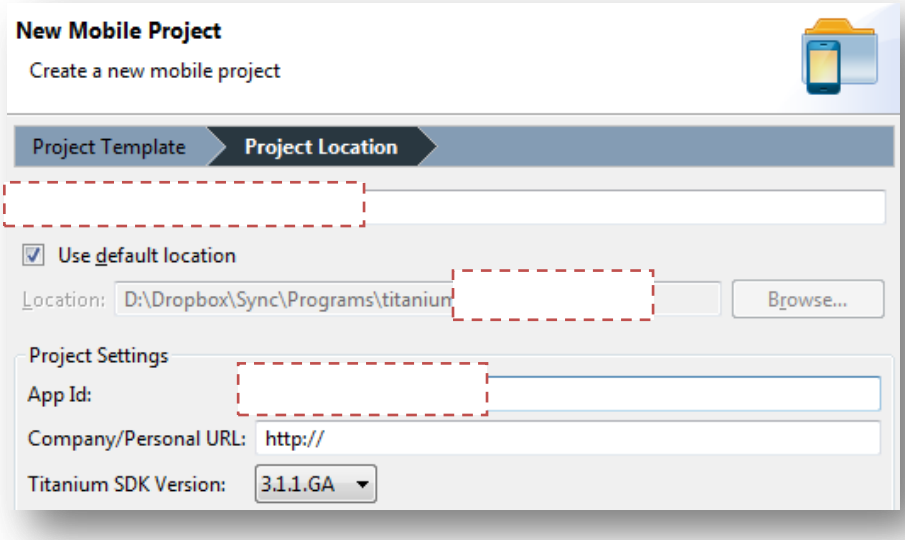
Go to **Launchpad** to find "**TitaniumStudio**" on the second page. You will be greeted by a login window. Please register using the login ID (***hkbu.itcamp@gmail.com***) and the password (***itcamp2013***). You should see the Titanium Studio page shown below.



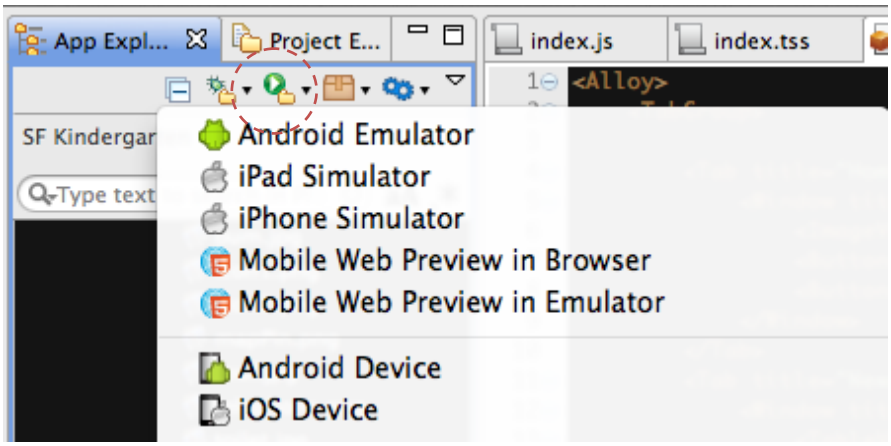
To create a mobile application, please locate "**File**" on the top menu bar and choose "**New**" → "**Mobile Project**". On the template selection screen, you need to select "**Alloy**" on the left panel and then select "**Two-tabbled Alloy Application**". After that, click "**Next**" to continue.



On the next page, you need to provide an **application name** (*i.e. HKBU App*) as well as the **App id** (*i.e. hk.edu.hkbu.comp.sig13*), as shown below.



It is time to see what we've got now. Please press the **Run** button (*circled on the screenshot below*) and then select "**iPhone Simulator**".



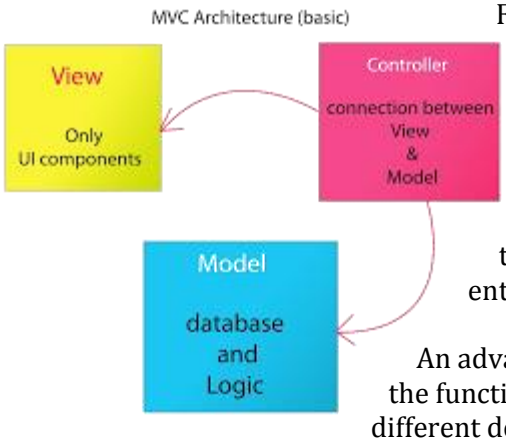
The first time you do this, it may take quite a long time to run it, especially if you are using the Android emulator. Don't worry. It will be much faster later.

Here's the first result: our app with basically two empty tabs.

Titanium Alloy[1]?

Starting from Titanium 3.0, it is recommeded to build app using Alloy. Alloy utilizes the model-view-controller (MVC) paradigm, which separates the application into three different components:

- **Models** provide the business logic, containing the rules, data and state of the application.
- **Views** provide the GUI components to the user, either presenting data or allowing the user to interact with the model data.
- **Controllers** provide the glue between the model and view components in the form of application logic.

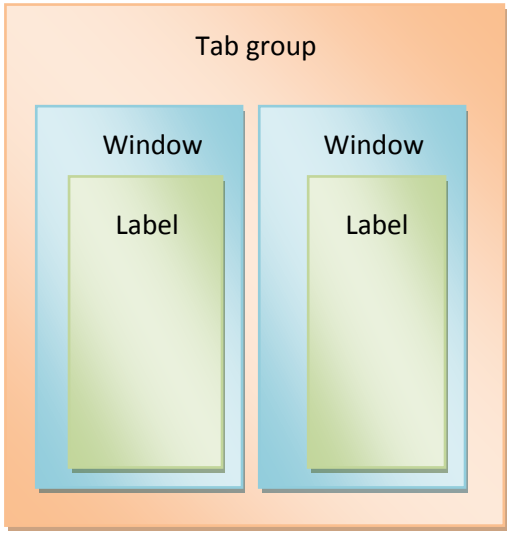


For example, in a calendar application, the models include events, reminders, invitations and contacts. The views present the calendar data and reminders to the user or allow the user to add events. For reminders, the controller checks the model data and launches a 'reminder' view to the user. For adding events, the controller opens an 'add event' view, then adds the event into the model data once the user entered the data.

An advantage of MVC is the ability to reuse code by separating the functionality. For example, you can have specific views for different devices, while keeping the controller code relatively the same and the model data unchanged.

The View

Under Titanium Alloy, the view can be constructed simply by editing an XML file. Find the **index.xml** file under the "**views**" folder on the left panel and let's examine the code. Basically, the app is a tab group with two tabs. For each tab, we have a window attached on it, and each window carries a label.



Let's add one more tab to our tabgroup. To add one more tab, simply make a copy of the following lines and paste it just above the line "**</TabGroup>**":

[1] http://docs.appcelerator.com/titanium/latest/#!/guide/Alloy_Concepts-section-34636240_AlloyConcepts-Model-View-Controller

```
<Tab title="Tab 2" icon="KS_nav_views.png">
        <Window title="Tab 2">
                <Label>I am Window 2</Label>
        </Window>
</Tab>
```

Then, modify the titles for the windows and the tabs. We will have menu, events, followed by map.

The Tabs

The first tab of our app is a dashboard, showing big icons for users to click. Let's do so by adding a DashboardView with the following code. Comment out the label and place the code within the window of the first tab.

```
<DashboardView platform='ios'>
        <DashboardItem image="appicon.png" />
        <DashboardItem image="appicon.png" />
        <DashboardItem image="appicon.png" />
        <DashboardItem image="appicon.png" />
</DashboardView>
```

A DashboardView can give at most 3×3 icons on each page, with their positions automatically arranged. A very nice feature, but only available in iOS. Thus we add platform='ios' to indicate this view will only be shown on iOS devices. You will have time to change the icon images in the pm session.

The second tab is a TableView, showing some faculties in HKBU.

```
<TableView>
        <TableViewRow title="University-wide Activities" cat="UWIDE"/>
        <TableViewRow title="Faculty of Science" cat="SCI"/>
</TableView>
```

Titles are the words on each row of the table. Cats are hidden fields of the rows, we will need them later.

The third tab is our map. Simply add a map view there and define some properties.

```
<View ns="Ti.Map" id = "mapView" animate="true" regionFit="true" userLocation="true"
mapType="Ti.Map.STANDARD_TYPE"/>
```

Also Ti.Map.SATELLITE_TYPE, and Ti.Map.HYBRID_TYPE.

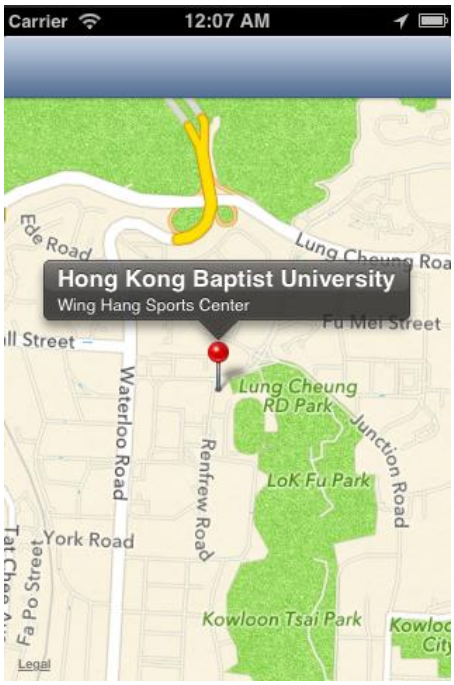
Whether to capture user's current location.

So now we have a dashboard with inactive button, and a map showing the wrong location.

The Controller

Notice that we have given an **id** (mapView) to this view, which allows us to reference it in the controller. Locate **index.js** under the **controller** folder. Provide the latitude, longitude, and zoom level there.

```
 $.mapView.region = {
        latitude : 22.33876,
        longitude : 114.18220,
        latitudeDelta : 0.01,
        longitudeDelta : 0.01
};
```



To find the **latitude** and **longitude** of a particular place, we can use Google Maps. Just right-click on the map and select "**What's here?**". The coordinates will automatically pop up in the search box. Please note that `latitudeDelta` and `longitudeDelta` determine the zoom level.

We can use an annotation to specify a particular building. To do so, please add the following lines to the **index.xml** file, within the map view tags.

```
 <Annotation id="campus" latitude="22.33876"
longitude="114.18220" title="Hong Kong Baptist
University" subtitle="Wing Hang Sports Center" />
```

Map route should be similar to annotation, I'm leaving this as a self-learning exercise in the afternoon session.

Callback Function

This is how to enable the functionality of a GUI element. Let's try this out on a button by adding this line below the map view.

```
<Button title="Move" top="10dp" onClick="btClick" />
```

Top is simply the position of the button, while onClick is expecting a function (btClick in our example) to be called once this button has been clicked. We MUST provide this function in the corresponding controller file (index.js under controllers).

```
function btClick(e) {
        $.mapView.region = {
                latitude : 24.33876,
                longitude : 115.18220,
                latitudeDelta : 0.01,
                longitudeDelta : 0.01
        };
};
```

It's time to define callback functions for the other GUI elements that we have constructed. Here I simply show you the extra codes for the index controller and you should modify the XML accordingly.

```
function dashClick1(e) {
        alert("Hihi");
}

function dashClick2(e) {
        $.index.setActiveTab(2);
}
```

```
function dashClick3(e) {
	console.log("dash3 clicked");
}

function dashClick4(e) {
	console.log("dash4 clicked");
}

function tableClick(e) {
	console.log("table clicked");
}
```

Passing Parameters from View to View

If your application only needs one level, then you can live happily forever with your index view-controller. But what if you want to show another page once the tableview has been clicked?

Locate the **controllers** folder on the left panel, right click on it and select **New -> Alloy Controller**. Name it as **department**. A department.js file is added to the folder. Also say hi to the **department.xml** under **views** and also **department.tss** under **styles**. (We will discuss style later)

Modify the **department.xml** by placing a window there.

```
<Alloy>
	<Window id="win">

	</Window>
</Alloy>
```

To bring up the department page when the tableview is clicked, modify the tableClick() function as below

```
function tableClick(e) {

	var departmentController = Alloy.createController('department');

	$.eventTab.open(departmentController.getView());
}
```

departmentController is simply a local variable, you can use any name here. The string '**department**', on the other hand, refers to the department view-controller that we just defined. Please also give **id**s (**mainTab**, **eventTab**, **mapTab**) to our three tabs in **index.xml**.

To pass parameters from view to view, we have to make changes in both controller files.

In index.js, the createController line becomes

```
function tableClick(e) {

	var departmentController = Alloy.createController('department', {
		cat : e.row.cat,
		index: e.index
	});
...
```

In department.js,

```
var args = arguments[0] || {}
var cat = args.cat || {};

$.win.title = cat;
```

First, **e** is a click event, and this event captures much useful information, like which row (**e.row**) has been clicked. Once we can identify the row, we can retrieve the related information (**e.row.cat**) of this row.

{cat} is the sole argument passing to department.js, represented by **arguments[0]**. However, it can carry more than one fields.

Exercise: Try to display the row index on the Department page.

The Model - Database

Let's download some extra materials from our website. Put the **myapp3.sqlite**, a sqlite database file, in the assets folder.

We have to setup a model to connect to this database file. Let's right click on the models folder and new an **Alloy Model**. Name the model as "**units**".

Next, modify the **units.js** file in order to specify the database file.

```
adapter: {
        type: "sql",
        collection_name: "units",

        "db_file" : "/myapp3.sqlite",
        "idAttribute" : "id",
}
```

The first two lines are already generated for you. Obviously, this is a sql-type adapter to work with database file. **Collection_name** tells which table you read from the db file. **idAttribute** tells the primary key of this table.

We will populate the data from **model** to the **view** (department) with the help of **Collection**. Let's see the changes needed.

Firstly, the **department.xml:**

```
<Alloy>
        <Collection src="units" />
        <Window id='win'>
                <TableView dataCollection="units">
                        <TableViewRow title = "{name}" />
                </TableView>
        </Window>
</Alloy>
```

Secondly, in the **department.js**, add the following lines there.

```
Alloy.Collections.units.fetch();

$.win.addEventListener("close", function(){
   $.destroy();
})
```

This line **<Collection src="units" />** creates a globally-accessible collection (**also named "units"**) and set up the connection with the **"units" model.**

Next, this line Alloy.Collections.units.fetch(); fetch the data file from the model to the collection.

Finally, the collection data are displayed in the tableview through the call of <TableView dataCollection="units"> and <TableViewRow title = "{name}" />. Only the name field is displayed.

It is advised to put down the destroy sentence to avoid memory leak if you follow this approach.

Some Simple Filtering

Add the following callback function in department.js

```
function filterFunction(collection) {
   return collection.where({cateID:cat});
}
```

Next, register this function in the tableview of department.xml

```
<TableView dataCollection="units" dataFilter="filterFunction">
```

Exercise: Explore the dataTransform callback.

Style

Try running our app on android emulator, the font size may look too small in those high-resolution devices. To fix this problem, we can modify the Titanium Style Sheet files.

For example, we can add the following lines to **index.tss**

```
"TableView[platform=android]": {
        backgroundColor: "#111"
},
"TableViewRow[platform=android]": {
        height: "50dp",
        font: {
                fontSize: "20dp"
        }
}
```

For cross-platform app development, we can use the "**platform**" parameter to specify the settings for a specific platform – iOS (iPhone), Android, or others.

Exercise: Practice the use of styling on a particular id or class.

The Model – Web Content

Up to now, everything in our app is static. This means that the content won't update. Thus, we try to get some dynamic content from the web, and then we use a **TableView** to display the content. Similarly, we will use the model and collection approach. But this time, we are not using the sql-type adapter. Please move the **restapi.js** to /assets/alloy/sync/.

Then, create a new model named, say "**restful**" and apply the following changes

```
config : {

	"URL" : "http://cs6067.comp.hkbu.edu.hk/~imt334/server/clients",

	"adapter" : {
		"type" : "restapi",
		"collection_name" : "restful",
	},
}
```

In index.js, add

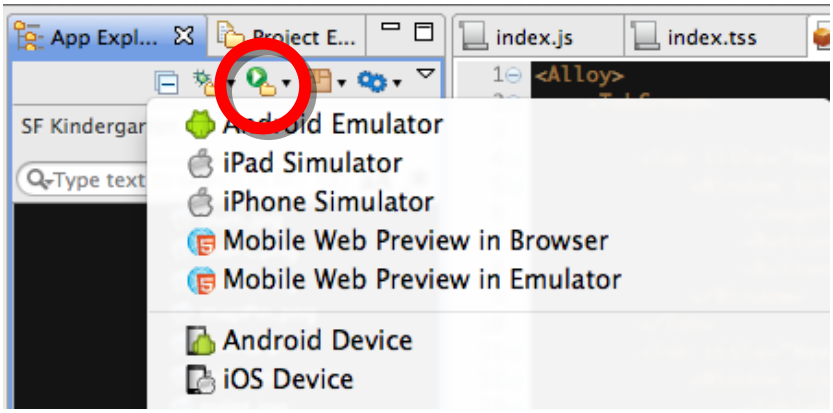
```
function dashClick3(e) {

	Alloy.Collections.restful.fetch({

		success : function() {

			var newsController = Alloy.createController('news');
			$.mainTab.open(newsController.getView());

		},
		error : function() {
			alert("Something Wrong");
		}
	});

}
```

Exercise: Develop the "**news**" view-controller to display the web content.

Export Your Work

Now, it's time to load your program to the mobile devices.

Connect your mobile phone to the computer. Then, press the **Run** button and select "**iOS Device**" if you use iPhone. Note that deploying apps to iPhone requires an Apple Developer Account. Please let us know if you want to transfer our app to your iPhone.



Further Reading

You may also try the following:

ImageView
ScrollView
CoverflowView (ios only)
WebView
NavigationView (ios only)

~ End ~