Development - Titanium

Aptean Ltd

Contents

1 Development - Titanium

1.1 Android Development Environment Setup

1.1.1 Titanium

- Download and install Titanium.

- Once dpkg file is downloaded, run it, then drag titanium to applications.

- Double-click the downloaded application and follow the instructions.

- Run the Titanium application.
- Select the default workspace, always.
- Login using your supplied Titanium login details.
- Install 2.4 SDK through the Help/Install Titanium SDK/Install from Update stream/Release/2.1.4.GA

 **Note:** The latest updates of Titanium no longer offer the option of downloading SDK version 2.1.X. To get this version:

- Navigate to http://builds.appcelerator.com.s3.amazonaws.com/index.html#2_1_X
- Find the latest version, right-click on the link and copy it.
- Go to Help/Install Titanium SDK.../Install from URL and paste in the link.
- Click Finish and wait for the installation to be complete.

 **Note:** If this is necessary, the TiApp.xml Titanium SDK Build Property on your local project will need to be changed to point at your 2.1.X version.

- Suggested Titanium theme is Light, UI is Advanced.
- Set up SVN within Titanium as follows:
  - https://wiki.appcelerator.org/display/tis/Subversion
- First time you use the SVN repository (for example, when you import the project), you will be prompted to set it up and select a connector. Choose the top one (SVN Kit 1.2.3)
-  **Note:** downloading: always accept the license agreement. Always restart when prompted.
- Assumes you have access to the SVN repository on \\10.43.0.21\Projects\EPOD_Development\SVN\epod

- Set up a Server share to 10.43.0.21/Projects and enter the correct username and password.
- Do this by choosing Add Server, entering smb://10.43.0.21/Projects
- Then Import a project, and follow the wizard.
- The SVN URL should be: file:///Volumes/Projects/EPOD/_Development/SVN/epod
- Enter the username and password you want.

- The SVN URL should be: P:/EPOD/_Development/SVN/epod or file://P:/EPOD/_Development/SVN/epod
- Enter the username and password you want.

- Choose to search for projects under this folder and import separately.
- But realistically, you are only interested in ePODAndroid.
- You can also install RapidSVN.

1.1.2 Titanium Module setup

two ways:

- Use the Dashboard window to install
- Click on My Modules
- Click on install on:
    - ♦ barcode
    - ♦ paint
    - ♦ imageFactory

OR

- Follow instructions at:
- https://wiki.appcelerator.org/display/tis/Using+Titanium+Modules
- Install the following modules:
    - ♦ barcode
    - ♦ paint
    - ♦ imageFactory

- Note: EXTRACT the zip files into the already-existent Modules directory, don't just leave the zips there. Watch out for Apple overwriting the entire folder rather than merging the two folders.

1.1.3 Android SDK

- Download.

- Install by extracting and copying the folders into the Applications folder.
- Add to the PATH env variable:
- from Terminal:
- vi ~/.bash_login
- Add:
    - ♦ export PATH=$PATH:/Applications/sdk/Platform-tools:/Applications/sdk/tools

- Install as normal

1.1.4 Android Dev Setup

Run the ADK setup utility (called Android in Platform-tools) Download all info from 4.03 to 2.2, skipping all 3.X variants (no-one really needs Honeycomb).

1.1.5 XCODE v4.5

 MAC ONLY

- Download
- Login to developer.apple.com with the supplied credentials.
- Click iOS Dev Center
- Click Looking for an older version of Xcode? Download Change search to Xcode 4.5 Click link to Xcode 4.5.1, around 1.5 Gb

- Install
- Double-click the downloaded file, then double-click the Xcode app inside.
- Agree to everything.

1.2 Android Programming Guide

The entire program set is automatically documented from the source code. This is stored in

```
P:\Product\EPOD\_Development\Documentation\Android Client\index.html
```

1.2.1 Program Folder Hierarchy

Under ePODAndroid:

- *CHANGELOG.txt* - All versions and changes made for the application. Keep this file updated whilst changes are being made. A formal new version will be added when files are checked in. Check in all files with descriptive notations. tiapp.xml should be checked in with the fill comment from this file.
- *README.md* - A descriptive header for the automatically produced source documentation.
- *tiapp.xml* - the main controller of the Titanium project. Contains settings, version control, extensions and parameters.
- *i18n* - multi-lingual files - see following section
- *platform* - platform-specific files - see following section
- *Resources* - the main coding area. Here are all the files that a developer would typically maintain or affect. See following sections.

1.2.1.1 i18n folder

Folders in here are organised per system language.

Each folder contains a 'strings.xml' file, that contains XML tags that have an ID and a piece of text or message. These text snippets should be used for every piece of displayed text within the Android Client application, such as:

- Labels
- Button titles
- All messages

These labels are accessed through specific functions written for this purpose, wrapping the standard Titanium Multilingual classes, as follows:

- getLabel(strLabelID) - This function returns the multi-lingual text for the label requested, plus ': '
- getMessage(strLabelID, pValues) - This function returns the multi-lingual text for the message requested, with the values inserted into the message string at the appropriate point.
- getString(strLabelID) - This function returns the multi-lingual text for the label requested

These functions are part of the Ti.App.style class.

It is imperative that each text item is placed into each of these files. Then translation can be added for different languages as and when required.

1.2.1.2 Platform folder

This folder contains the preferences or settings applicable to each specific platform. The standard Apple, Android and Titanium documentation should be consulted to ascertain the usage and how these are maintained.

1.2.1.3 Resources folder

- *Android* - Android-specific images, segregated by:
  - ♦ Resolution (hdpi, mdpi, ldpi)
  - ♦ orientation (land, port)
  - ♦ wide-screen ratio (long, notlong)

- *Database* - The DAL (Data Access Layer). These are functions and objects that are used to directly affect or access data from the database. All objects are documented as part of the Namespace Database and in specific classes for each DAL object.

- *DBConnection*
- *PDA_CONTAINER*
- *PDA_DEVICE_TYPE*
- *PDA_JOB*
- *PDA_JOB_GROUP*
- *PDA_LOAD*
- *PDA_PHOTO*
- *PDA_PRODUCT*
- *PDA_REASON_CODE*
- *PDA_SERVICE*
- *PDA_SERVICE_ACTIVITY*
- *EPOD_SERVICE_PRODUCT_MASTER*
- *PDA_SERVICE_ACTIVITY_MASTER*
- *PDA_SERVICE_PRODUCT*
- *PDA_SITE*
- *PDA_USER*
- *PDA_VEHICLE*
- *PENDING_DATA*
- *XF_AUDIT*

- *images* - Generic images are added here then referenced in classes or styling information.
- *System* - Reusable objects - documented as part of Namespaces System, UI and Webservices.
- *ui* - main screen objects - documented as part of Namespace UI.
- app.js - This is the start point for the entire application.

1.2.2 Application Hierarchy

This section covers how the application calls each screen in a standard flow. Unless specified differently, each element is part of the UI namespace. All of these items are documented as part of the Project documentation in file:///spekefs2012/Projects/Product/EPOD/_Development/Documentation/Android%20Client/index.html - paste this link into your browser window to begin exploring the documentation.

App Start (app.js) - create database if required and call login

|

Login (Login.js) - user logs in with username, password and vehicle.

|

Vehicle Checks (VehicleChecks.js) - if required, vehicle checks are requested here.

|

Job List (JobList.js) - this retrieves Loads assigned to the user, shows a list of the jobs and starts timers.

|

Cancel a Job (Cancellation.js) - Cancel a job and return to the Job List Job Detail (JobDetail.js) - Show the information of a job and start/travel/arrive.

|

Cancel a Job (Cancellation.js) - Cancel a job and return to the Job List Collection (Collection.js) - the collection process Delivery (Delivery.js) - the delivery process Service (SM_Service.js) - the service process

|||

Signature Capture (Signature.js) - all processes above request customer and possibly driver signature. Job Photo (DocumentPhoto.js) - optional photo taken of the completed job.

|

Send Message Update and return to Job list (through Webservices)

1.2.3 Messaging To and From Client

Messaging to and from the PDA Client is always initiated by the client, in the form of a request to the Server-side Calidus_ePOD webservice. The service will always respond with a response message.

The message processors always check to see if there is a data connection before attempting to send.

Additionally, the function these processes call (Webservices.SendWaitXML) handle errors returned from the connection attempt.

So, there are several return statuses:

- Success! - process the response message (Webservices.ProcessResponse) and return the result of that
- Fail - NAK received - return and callback -2
- Fail - other (XML Document returned is null, has no child nodes or is invalid in some way, or the send actually failed, or the HTTP connection received an error or timeout) - return and callback -1

Each of these statuses are handled by the calling routine.

Messages need not be sent immediately, they could be placed onto a Pending queue, for sending in the background through a timed process (see later).

The following is a list of all Request messages, the tags used to send them, the response tags expected and the usage.

The Request messages that can be sent (by their various DAL objects) at the time of writing that are placed on the queue are:

- Job Update - tag JOB_UPDATE, expecting JOB_UPDATE_RESPONSE - used on completion of a Job (either confirmation or cancellation)
- GPS Status - tag GPS_STATUS_REQUEST, expecting GPS_STATUS_RESPONSE - used when sending a Vehicle Location Update.
- Photos - tag PHOTO_REQUEST, expecting PHOTO_RESPONSE - used when sending images associated to a Job Update.
- Load Update - tag LOAD_UPDATE_REQUEST, expecting LOAD_UPDATE_RESPONSE - used when sending back Metric information obtained about a Load.
- Vehicle Check - tag VEHICLE_CHECK_UPDATE, expecting VEHICLE_CHECK_RESPONSE - used when updating the server with Vehicle Check results.

Messages that are never queued (i.e. they work immediately or are discarded) are:

- Logon request - tag LOGON_REQUEST, expecting LOGON_RESPONSE - used when logging on, expecting to receive back all standing data that has changed since last logon.
- Grace logon request (1st time logon) - tag LOGON_REQUEST, expecting LOGON_RESPONSE - used solely when logging on for the first time on a device, or after a database update. Works as per Logon Request.
- Load request - tag LOAD_REQUEST, expecting LOAD_RESPONSE - used when requesting work, expecting to receive a load with a series of jobs.
- Auto-update request - tag AUTO_UPDATE_REQUEST, expecting AUTO_UPDATE_RESPONSE - used from the AutoUpdate Timer to refresh any changes to a load or jobs being completed.

Possibly queued or immediate, depending on circumstances (whether we have a data connection):

- Job Lock Request - tag JOB_LOCK_REQUEST, expecting JOB_LOCK_RESPONSE - used to set a job as in progress by this driver on the server. If immediately sent, expects an update of the job being locked, in case there have been changes.
- Job Arrival Request - tag JOB_ARRIVAL_REQUEST, expecting JOB_LOCK_RESPONSE - used to mark a job as arrived at by this user on the server. If immediately sent, expects an update of the job, in case there have been changes.

The timed processes are started from the Job List screen and control sending messages in the background. They are:

- Auto-Update
- GPS Tracking
- Pending Messages

1.2.3.1 Auto-Update

Fully documented in class Ti.App.AutoUpdateTimer, in namespace Webservices.

When a Load is picked up for the user (from Job List) a timer checks every few minutes with the server to see if there have been any changes to the jobs assigned on this load. This process sends the message to the server and processes the response. An application-level event is raised that is processed by the Job List and Job Detail forms (if they are open), which refreshes the data on the screens and lets the user know that changes have happened.

An auto-update can also be forced by the user from the Job List main menu.

This timer should be running when in Job List and Job Details only.

1.2.3.2 GPS Tracking

Fully documented in class Ti.App.GeoUpdateTimer, in namespace Webservices.

If configured, this timed process sends a message on the queue informing the server of the user, vehicle and last known GPS co-ordinates. These are stored on the Vehicle and in tracking on the server.

This timer should always be running.

1.2.3.3 Pending Messages

Fully documented in class Ti.App.PendingDataTimer, in namespace Webservices.

Whenever a message is sent to the pending queue, this timer picks up the message and attempts to send it in the background. Any errors are discarded and the message is left if unsuccessful. A successful send automatically attempts to send the next message.

This timer should always be running.

1.2.4 Version Numbering

Note: Android client version numbering is not intended to stay in line with .NET application numbering, either client or server.

Android Client version numbering comes in several parts:

- Android Version Code - this is the numeric integer version number used by the Google Play application (the market) to check whether a version of an app stored on a device is greater than or less than the version on the market, and whether to offer to update it. Example: 24.
- Android Version Name - the version number we want to call this build of the application. For a built and released version, this is the same as the titanium version. Example. "1.2.21"
- Titanium Version - the version number we want to call this build of the application. Examples: "1.2.21"

The Android Version Name and the Titanium Version should always match for a released version of the app.

When testing internally, is is normal for the Android Version Code and Name to be incremented to the next full version, while the Titanium version will be marked as an alpha build (indicated with an alphabetic character after the revision e.g. "1.2.21a"). It is the Titanium version that is displayed prominently on the device at log-on. No testing build should ever be distributed to a client or the Market.

Note: The current Android Version Code should be reflected DBConnection.js as well - see the following section for details.

Examples: The last built and released version of the code has a checked-in Tiapp.xml populated as follows:

- Android Version Code - 24
- Android Version Name - "1.2.21"
- Titanium Version - "1.2.21"

When making the next changes, developers should change their local file as follows:

- Android Version Code - increment by 1 - 25

- Android Version Name - increment revision number - "1.2.22"
- Titanium Version - increment revision number and add alpha indicator - "1.2.22a"

Note: Minor versions should be increased per new client release. Major versions only on major rewriting of code or additional functionality set.

Note: Local changes to Tiapp.xml need not be checked in - this should be completed during the build and release process. This should only take place if other fundamental changes are made to the file.

The final build process (whomever completes this) will check in a version of this file, with values as follows:

- Android Version Code - 25
- Android Version Name - "1.2.22"
- Titanium Version - "1.2.22"

It will be marked with the full change history of this version, from file CHANGELOG.txt - see the Build and Release process for details.

1.2.5 Database Creation and Updating

The database is created (or updated) at the start of the application, through functions in DBConnection.js, in namespace Database.

The database is in SQLite3 format.

The database decides whether it needs to be updated based on

- the version in the database
- whether this is a first log-on
- whether the version requires a database update.

In order to achieve this:

- The Android Version Code must be updated in the Tiapp.xml file.
- The latest version must be updated in function InstallDB.
- The last version that required a database update must be updated in function InstallDB.

The function always stores the version that created or updated the database in table SYSTEM.

The function checks whether:

- The stored version on the database is less than the last version that required a database update.
- If it does, the database is recreated with no data.
- It updates the latest version of the app in table SYSTEM.
- It updates a system parameter to show that standing data needs to be completely refreshed.

The Login process checks the parameter and re-downloads all data required (as if it's a first login)

1.2.6 Variable Notation

Follow standard variable notation as follows:

- global - preceded with 'g' or 'g_'
- parameter to a function - preceded with 'p' or 'p_'
- variables preceded with 3-character type, for example:
    - str - String
    - obj - General or custom object
    - lng - Long
    - int - Integer
    - bln - Boolean/Flag
- Titanium objects should be named as follows:
    - lbl - Label
    - pck - Picker
    - btn - Button
    - tbl - Table

- tbr - Table Row
- vw - View
- chk - Checkbox
- scr - ScrollView
- img - ImageView
- tab - Custom Tab or Tab
- sli - Slider
- swt - Switch
- tgp - TabGroup
- tvw - Table View
- tvr - Table View Row
- txt - Text Area or Text Field
- udf - User-defined field object (also obj)
- win - Window

Note: These lists are not comprehensive or infallible. They are intended to guide towards a maintainable, cohesive documentation.

DAL Objects:

DAL class PDA_JOB would be used to create object PDAJOB.

Examples:

- global boolean to control Blah - g_blnBlahFlag
- text field Blah - txtBlah
- label for the text field above - lblBlah

1.2.7 Garbage Collection

Warning: Although there are issues with garbage collection in Titanium, this information is still being verified. As standard, we should follow these procedures as they will make for easier garbage disposal, whatever the reason. See http://docs.appcelerator.com/titanium/2.1/#!/guide/Managing_Memory_and_Finding_Leaks for more details.

- Create an object vars. **Note:** Although this doesn't obey the naming conventions above, it does allow for the easy conversion of existing code, and should be followed until all is complete. After, these conventions will change the name to objVars.
- All variables should be created under this object in that form. For example, a Blah text field would be declared as

```
vars.txtBlah=Ti.App.style.creatTextField('winMine_txtBlah', ' ', {});
```

- All events should be declared as a function variable, to easily identify them for event removal later. For example:

```
vars.AutoupdateEvent = function () {
  vars.winMine.close();
};
```

- Upon closing the window:
  - remove any events attached to the window or the app referencing the window, referencing the function object used to create it.
  - Set the window object to null;
  - Set the vars object to null;

Example:

```
vars.winMine.addEventListener("close", function () {
    Ti.App.removeEventListener('OrientationChangeOBS', evtOrientationChange);
    // Sets all the objects to null - is this enough to dispose correctly?
    // If not, we'd need to loop through...
    vars=null;
    // Possibly run some callback function or other processing here.
});
```


1.2.8 Styling Objects

Ti.UI objects are used to add buttons, labels, windows etc. Object arrays are used to define the properties of that object, such as positioning and styling information.

Titanium supports styling through specifying a series of identifying properties directly against an individual variable. It does not support reusable generic styling and restyling on demand, only at runtime.

To allow for a greater reuse of styling parameters (classes) and restyling (predominantly on orientation change, which must be handled manually in Titanium, the style object was created, placed into Ti.App.style. This is defined in Style.js.

This represents a series of reusable, individual ID and bespoke customer classes that are applied to items automatically at creation. It also handles identifying these objects more clearly so that methods of this object can be used to force a restyle at certain events.

Style types are:

- Reusable Class - defined in Ti.App.style.styles object.
- Styling for specific ids - defined in Ti.App.style.ids.
- Customer-specific classes and ids - defined in Ti.App.style.configItems, that overloads classes and ids if they exist. **Note:** This is largely replaced by Server-side styling, detailed below.

1.2.8.1 Adding and Amending Classes

New styles are added by adding a new named object to the styles, ids or configItems objects, as follows:

```
Ti.App.styles={
    myExistingStyle : {
        left : "1%",
        top : "50%"
    },
    myNewStyle : {
        left : "2%",
        top : "51%"
    }
}
```

Landscape styles are used in preference when styling or restyling an object, depending on the orientation of the device. This is calculated in Style.js and recalculated whenever an orientation change event fires. When this fires, this in turn generates an application event that can drive a listener in a Ti.UI.window or Ti.UI.tabGroup object, if created, which typically restyles all objects under the window, using Ti.App.style.restyleItems.

Landscape classes and styles that have landscape forms (in styles, ids and configItems) are defined as follows:

```
Ti.App.styles={
    myExistingStyle : {
        left : "1%",
        top : "50%",
        obsRestyle : true
    },
    myExistingStyle_land : {
        left : "1%",
        top : "30%",
        obsRestyle : true
    }
}
```

Note: The restyle function overloads properties from the new style onto the existing object. In the example above, if the object using this class myExistingStyle is restyled on an orientation change to landscape, the only change will be that the top property will change from 50 to 30 percent. Consider the following example:

```
Ti.App.styles={
    myStyle : {
        left : "1%",
        top : "50%",
        width : "98%",
        obsRestyle : true
    },
```


```
    myStyle_land : {
        right : "1%",
        top : "30%",
        width : "49%",
        obsRestyle : true
    }
}
```

The properties of the object are not removed upon restyle, so the result of restyling from portrait to landscape would result in an object with the properties as follows:

```
    {
        left : "1%",
        right : "1%",
        top : "50%",
        width : "98%",
        obsRestyle : true
    }
```

This may not be what was intended. It should be defined like this:

```
Ti.App.styles={
    myStyle : {
        left : "1%",
        right : null,
        top : "50%",
        width : "98%",
        obsRestyle : true
    },
    myStyle_land : {
        left : null,
        right : "1%",
        top : "30%",
        width : "49%",
        obsRestyle : true
    }
}
```

If the device starts in landscape orientation, the landscape class will be used which will look correct. Only when it is reoriented from one to the other will the problem become evident. This is why the properties that are not required are nulled if they are to be overwritten.

Note: If no restyle is specified, the object will not restyle at all (except when the preference Style property is changed, when everything is forced to be restyled) it won't restyle at all, regardless of the presence of an "_land" class.

The class also supports resizing pixel sizes if required, based on the resolution of the device. This is largely most effective on font and row height properties, where percentage heights are difficult to gauge.

This is achieved using Ti.App.resMod, as follows:

```
Ti.App.styles={
    myStyle : {
        left : null,
        height: (100*Ti.App.resMod),
        top : "30%",
        width : "49%",
        obsRestyle : true
    }
}
```

Another resolution modifier, Ti.App.resModX, exists for width modifications. Note that this is normally only required for specific height and width views - the signature imageView for example.

These styles are automatically applied to UI objects that are created through Ti.App.style.create<Object> methods, such as Ti.App.style.createLabel.

1.2.8.2 Classes Available

Standard Classes available are:

- small/smaller/medium/larger/large - font size helper classes
- left/right - text/object horizontal alignment helper classes
- generalButton - sets the display characteristics of a general button. Used in conjunction with positioning classes, as follows:
    - btnSplitBottom
    - btnBottom
    - btnSplitMiddle
    - btnMiddle
    - btnTop
    - btnScan
    - btnLeft
    - btnCenter
    - btnRight
- generalText/wideText - full-width positioning
- halfText - half-screen width text box positioning (right-hand side)
- generalLabel - general label styling
- tableHead - header for a table or a title for a screen.
- tableRow - general styling for a row, taking into account resolution modifiers for size.
- evenRow - slight background styling for even rows.
- tableView_Status/_A/E/I/X/C - used to format the row based on the status of the order, or change status after receipt. Used in Job List.
- settingsRow - unused as yet.
- portraitWindow/landscapeWindow/anyWindow - basic settings for any type of window. Always use anyWindow
- instrLabel - class for instructions label on Job Details - should be an ID!
- tabContainer/tabDetails/tabProducts/tabNotes/tabInfo/tabPreworkActivities/tabMCRefs - generic tab items
- checkedBox/unCheckedBox - classes used by custom check-box object.
- tabOn/tabOff - classes used by custom Tab object
- productTable - A class specifically for tables
- scanText/scanTextBtn - used within the Barcode object
- popupWinView - used for pop-up window views
- scrollViewTable - used for mainly full-screen views that scroll vertically.
- scrollViewRow - used for the above.
- claused/cancelled/normal - used for Signature - container clausing.

1.2.8.3 Server-side Styling

Device styling changes:

1. The definitions for a specific style (e.g. PLINK, LANE etc.) can now be found:
    1. In the device code in the same place (within the Ti.App.style.configItems object declared in style.js). This has been kept for backwards compatibility with older versions of the server which cannot send the style definitions (will eventually be removed).
    2. On the server database on the EPOD_PDA_STYLE table. I have set the data up on the V3 dev database with the style definitions for all currently existing styles.
    3. In the server code as part of the EPOD_SETUP_LISTS stored procedure. This is for the initial creation of the style data for all of the styles for new sites (the ones not required for the new site can then be deleted).
2. There have been some minor changes to the way styles are defined to allow the style data to be stored and passed as a JSON string:
    1. The style definitions (both data and code, including Ti.App.style.configItems, Ti.App.style.ids and Ti.App.style.styles code) has had to stop using getString to set text. These were being evaluated when the app loaded (before any style definitions have been read from the database) and would never be re-evaluated. This also caused the existing bug where changing style would not change some titles until you had exited/re-started the app (as the strings were not being re-evaluated and so were not picking up changes to the styles label IDs). The setting of these titles have been moved to the code that creates the object so that the translations are evaluated when the individual screens are built (after any style definitions have been read from the database).
    2. The styles stored in the data on the database need to have double quotes around the property names and any string values. For example where the style definition in the code is borderColor:?#220000? the style definition in the data is "borderColor":"#220000". Note that this does not affect the code in Ti.App.style, just the definitions in the database.
    3. The use of Titanium constants (e.g. Ti.UI.SIZE, Titanium.UI.KEYBOARD_NUMBER_PAD or Ti.App.resMod) has changed. These need to be stored in the database as strings, the JSON parser on

the device then uses a reviver function to evaluate these strings and use the evaluated value in the strings place. The reviver function (and the regex pattern for recognising the strings that need to be evaluated can be found within the vars.funUpdateStyle function in Login.js). For example where the code has height : Ti.UI.SIZE the data must have ?height?:?Ti.UI.SIZE?. Where a value is set to a multiple of Ti.App.resmode (e.g. fontSize : 22 * Ti.App.resMod) the entire property value must be stored as a string (e.g. ?fontSize? : ?22 * Ti.App.resMod?). Again this does not affect the code in Ti.App.style, just the definitions in the database.

From an Android development point of view. it is expected that the following to be the process for developing new styles or large changes to styles:

1. Create a style on the server with a specific style ID and no style definition. This is so the server will pass the new style ID to the device without any style definition, the device code (designed for backwards compatibility) will then use this style definitions from the configItems property that matches the style ID.
2. Develop the style as usual with code in configItems but bearing in mind 2.a, 2.b & 2.c above (i.e. do not use getString and use double quotes around any property value that needs evaluating because it is/contains Titanium constants).
3. When the style is as required convert the style from configItems to a valid JSON string (by adding double quotes around property names and any values using of Titanium constants). There is a web tools page (the ?Style Data Converter? page in the ?Device Tools? section of the web tools page: \\spekefs2012\Projects\Product\EPOD\Support\Utils\Web Tools\index.html)
4. Use the valid JSON string to update the style definition on the development database and test that this is passed to the device and parsed correctly.
5. Finally remove the style from configItems (unless this is a new style for an existing client using an old version of the server) and use the JSON string to update the EPOD_SETUP_LISTS stored procedure.

1.2.8.4 Creating Objects

Standard Ti.UI objects with Ti.App.style.create<Object> functions available that 'overload' the standard UI styles are:

- Label
- Picker
- Button
- Table
- TableRow
- View
- ScrollView
- ImageView
- Tab
- Slider
- Switch
- TabGroup
- TableView
- TableViewRow
- TextArea
- TextField
- Window

Each object creation function allows you to specify:

- The id of the object
- The class to be applied. Several can be specified, space-delimited.
- An object array containing in-line styles to apply.

The id of the object is typically defined as the window to which this element belongs, plus '_', plus the variable name.

- <window>_<varName>

Example:

- winMine_txtBlah

Iterating fields (for example in a table) would be ID'd with the iteration number:

- <window>_<varName><iterationVariable>

Example:

- winMine_lblBlah1
- winMine_lblBlah2

For example:

```
var vars={};
vars.winSelf=Ti.App.createWindow('winSelf','anyWindow',{});
vars.vwSelf=Ti.App.createView('winSelf_vwSelf',,{});
vars.winSelf.add(vars.vwSelf);
vars.vwSelf.add(Ti.App.createLabel('winSelf_lblText','generalLabel',{
    text : "Enter Text",
    top : "50%",
    left : "1%",
    width : "48%"
}));
vars.txtText=Ti.App.createLabel('winSelf_txtText','generalText',{
    top : "50%",
    left : "51%",
    width : "48%"
}));
vars.vwSelf.add(vars.txtText);
vars.winSelf.open();
```

- This example shows a variety of creation and opening mechanisms.
- It also shows a variety of styles applied, from Class, ID and in-line.
- Typically, a variable does not need to be created separately if it is never referenced again - see the creation of the label above.
- The example also shows the use of the vars variable, typically used to make disposing of objects quicker on closing of a window (see Garbage Collection)

1.2.8.5 Hierarchy of Classes

- Class (last takes precedence over first)
- Device-Specific Class (Hi-resolution, Orientation - last takes precedence over first)
- Style-specific Class (last takes precedence over first)
- Style-specific Device-Specific Class (Hi-resolution, Orientation - last takes precedence over first)
- ID
- Device-Specific ID (Hi-resolution, Orientation)
- Style-specific ID
- Style-specific Device-Specific ID (Hi-resolution, Orientation)
- In-line properties (highest precedence)

1.2.8.6 Custom Controls

Custom controls have been created as part of the style object, in order to simplify the creation of a complex series of options, or to create an object that Titanium doesn't provide.

- Custom Tab - used when a tab object is required within a window, rather that outside, like TabGroup.
- Checkbox - a standard check-box, that changes from a cross to a tick, and vice versa.
- UDF - User-defined field object. This is a object of many fields, defined by an XML configuration, e.g. Vehicle Checks.

- **Warning**: customTab and how to use it.

- **Warning**: Vehicle checks - UDF configurable fields and how to use them.

1.2.9 Source Control

When checking in items, use the Team Synchronisation view of Titanium so you can see all changes you have made from the baseline.

Click *Synchronise* when using for the first time - always synchronise changes before a build.

The legend on the file-list should indicate whether:

- The local file is changed and needs committing.
- The SVN file has changes in that you need to incorporate into your local files.
- Conflicting changes (both of the above).

In all cases, this view can be used to see the changes and incorporate changes into your local file until the changes have been merged - simply double-click on the file to be shown a list of all changes. Use the buttons provided to search for, incorporate or ignore changes.

The following standard rules apply to Android work:

- Assign the development to yourself (either through the standard Supimix procedures or from the Log List)
- Check out the source code through the Source Control software or direct from the Studio software.
- Make any changes and test locally.
- When complete, check in your changes with an appropriate comment, initially referencing the log. For example:
    - 291925 - Fixed X when Y happens
- Update the log to Development Complete status in Supimix (or Fixed status on a log list), ensuring that you enter resolution notes.
- Assign the log to a tester, who will then test the functionality and close the log (or reassign to be released in a patch).

1.2.10 Commenting Code

Code is automatically documented through JSDoc, so valid commenting of code (functions, classes, properties, methods and files themselves) is important.

- Comment with /** ... */ BEFORE the declaration of the item you wish to comment.
- Multi-line comments should always be preceded with an asterisk on each line, as in the examples below.

Files should have a comment on the top as follows:

```
/**
 * @file Description of the file
 * @namespace ANameForTheFile
 * @version 1.0
 */
```

Functions and definitions that are not part of a class should be commented as follows:

```
/**
 * This function does blah.
 * Links to external sites (like the documentation) can be done like this
 * {@link http://172.198.45.54/calidus-assist/EPOD/index.php/PDA_Job_Details}.
 * html can be inserted to make the comments look better in the generated documentation,
 * as follows:
 * <ul>
 * <li>Address & Contact</li>
 * <li>Instructions</li>
 * </ul>
 * @param {function} funJDCallback Each parameter will be documented like this.
 * Put square brackets around them to indicate optional.
 * @param {PDA_JOB} PDAJOB Links to other parts of the documentation can be made
 * with the link command, as follows: {@link Database.PDA_JOB}
 * @returns {Ti.UI.window} The window object to be displayed.
 * @memberof ANameForThisFile
 */
```

-

Class definitions/prototypes should have the main function commented as follows:

```
/**
```


```
 * This object is declared and instantiated in this file.
 *
 * @memberof ANameForThisFile
 * @constructor
 **/
```

Class methods and properties should be commented as follows:

```
/**
 * This method does as follows
 **/
```

Notes:

- @memberof must reference an actual namespace that has already been declared, but not necessarily within this file. So, a namespace declared in file1 as UI can be referenced in file 2. Indeed, the file itself can be made a member of another namespace, to group lots of other namespaces together. So, for example, app.js is named UI, Login.js is named Login, but is a member of UI.
- Properties and methods of a class do not need to be declared as a memberof something, as they are automatically part of the constructor class.
- All other commented functions in Namespaced files must be a member of a namespace, or they will be listed as Global functions in the documentation.
- Properties or methods of a class can be linked to with {@link namespace.class#property}
- Functions or variables in a namespace must be linked as follows: {@link namespace.function}
- @version should only be used on files. It should be version 0.1 while being written, then 1.0 when finished. The version only needs to be changed if there are major changes to the document.
- For EPOD application specifics:
    - DAL objects should be a member of the Database namespace
    - New screens should be a member of the UI namespace
    - New connection-based files (i.e. webservices) should be a member of the Webservices namespace.

1.2.11 Building

The application can be tested on the PC through an emulator. To set this up:

- Click *Run/Run Configurations*.
- Set up a configuration as follows:
    - Android SDK: 2.3.3
    - Distribution Location: Either the Projects folder for a full distribution (P:\Releases\EPOD\Android\Distro), or any local folder (e.g. C:\Users\you\Titanium\distro).
    - Keystore Location: P:\Product\EPOD_Development\Development Tools\AndroidKeyStore\KeyStore
    - Keystore Password: Contact the development team for details.
    - Key Alias: Calidus-EPOD

1.2.12 Developer Testing and Debugging

The application can be tested on the PC through an emulator. To set this up:

- Click *Run/Run Configurations*.
- Click on Titanium Android Emulator, and follow the instructions and defaults.
- Once set up, this can be used from the Titanium tool-bar, once the project is selected.

Note: the emulator does not have to be restarted each time you make a change. Either:

- Stop the app on the device and click the app on the emulator desktop.
- Click the run button again, which will stop the app on the emulator and restart it.

Notes:

- When debugging on the emulator and you want to pick up latest changes, you could click the run button again in Titanium. Faster can be clicking on the app within the emulator - it will reload almost every change (that has been saved). Sometimes it fails though.
- When attempting to run the app again from titanium, you get lots of errors regarding can't build because file is in use. You can try stopping previous background tasks, or you can persevere - it usually picks up again after a few goes. Stopping and starting Titanium resolves lots of things...

- Preference changes and i89n (multilingual) changes do NOT get picked up when restarting the app from the emulator. In fact, sometimes you may need to force this by cleaning the project and rebuilding, then installing from Titanium again.
- adb can time-out if left for a while. If you start the app on the emulator and it hangs, that's probably what happened. Start the app from Titanium. If that doesn't work, reset adb from DDMS or Monitor or through the command line (using adb kill-server; adb start-server). If that doesn't work, exit the emulator, the start the app from titanium - should load another emulator. Combinations of the above (resetting ADB and exiting Titanium and the emulator should always work. If it doesn't, restart your PC.

Note: See the Testing section to see how to use the emulator.

Titanium also allows debugging through the emulator. This can be set up and run very similarly to the standard testing, but through *Run/Debug Configurations* and the **Debug** button from the main tool-bar.

- Breakpoints will be obeyed.
- Implicit breakpoints will occur whenever a runtime error is encountered, regardless of whether the error is caught and handled.
- Watches can be added.
- Immediate checks can be keyed in.

Please see the Titanium documentation for more details.

Running on the emulator (or a rooted device) allows the user to examine the SQLite database installed on the device. From a cmd prompt:

```
cd {Android SDK Location}
adb -e shell
sqlite3 /data/data/com.obslogistics.epodandroid/databases/CALIDUS_EPOD
```

Once in the command-line, you can use:

- .help
- .tables
- Basic SQL

Consult the SQLite documentation (http://www.sqlite.org/docs.html) for more details.

Whether running through an emulator or on a device, the system log contains the most information, showing all errors and warnings from Titanium Runtime, but also your own debug comments (written into the code with the logDebug function.

However, these can be better accessed through Android's debug monitor, DDMS. To use:

```
{Android SDK Location}\tools\ddms.bat
```

You can see the log and filter it easily through here. You can also monitor memory usage and garbage collection - see the Android (http://developer.android.com/tools/debugging/ddms.html) and Titanium (http://docs.appcelerator.com/titanium/2.1/#!/guide/Managing_Memory_and_Finding_Leaks) documentation for details.

1.2.13 Development QC Process

Code reviews are in place for the EPOD projects and must be followed - see section Development Code Reviews for details.

1.2.14 Other

This section details observations, suggestions and potential pitfalls that may aid the Titanium developer.

Warning: Additional notes will be added to the bottom of this section as discovered.

- **Include files once**

Our application is structured so that all code calls all other code, hierarchically, so globally accessible code included in a parent will be accessible in the child. If you include it again, you simply overwrite the definition and slow down the window

you want to display. The rules are:

- Include code only at the level that you need it.
- Include it only once.

- **Describe synchronous and asynchronous parts of the code, and why return and callbacks are your friends.**

When writing Titanium apps, there are two sections:

- ♦ When you are building a screen, with all the Ti.App.style.createXXX() code
  ♦ When you are executing a screen (in Event code)

Build code is executed serially, so declare your functions before you use them.

Event code is asynchronous, so calls to functions will execute at the same time as the calling code. Serialise these if necessary by including call-back functions as parameters, or make the function return a value and check for the return when calling. So, "if (myFunc()==true)" will execute synchronously, whereas "myFunc();" will execute asynchronously.

- **Close and null DB and RS objects after use, or you get runtime errors.**

Straight error - just follow the rules above and you won't get errors with recordsets and DBs.

- **Scrolling views and why they're better and worse than a table.**

There's a bug in tables implementation - memory loss is possible when loading or unloading table rows. It's also less efficient in terms of number of objects (1 less in the hierarchy) if you put these in a scrolling view and control the rows yourself. See examples in Signature.js (the containers tab).

Basically, a table is made up of:

- tableView
- tableViewRow
- View
- And finally contents.

This has been deprecated from both Android and Titanium in new versions, for the following reasons:

- Laggy
- Overly memory hungry
- Memory leaks

The other issue we have with our tableViews that we use is that we created these right at the start, when we really didn't have much of a clue as to how to use Titanium. Now that we have more understanding of the language, it's clear we've implemented things poorly.

The good news is, it can be replicated far easier (lighter, faster and with less objects) using scrollViews.

The hierarchy is as follows:

- tableView - a scrollView object with vertical scrolling and a fixed height and width
- tableRow - a view object with a fixed height, with tableView as a parent
- Contents

The advantages are too numerous to mention - as well as resolving the issues listed above, these can also have variable height rows, automatic height rows, horizontal and vertical scrolling? the list goes on.

The major differences between using tableViews are:

- With less hierarchy of objects come some changes to events, when parents and children are being checked.
- The assigning of 'rows' isn't through a custom 'data' property, adding through an array of objects, but rather through simple (and standard) add methods used in standard views.
- Removing 'rows' from the 'table' is now through the standard remove method, although this can get a little difficult to handle.

Other examples:

- Services.js (the products tab)
- style.js' UDFields or UDFields2 objects.

- **"Error generating R.java from manifest" when building**

Check your TiApp.xml file - there may be problems with the version of Titanium being run, or the versions of the plugins. Bear in mind that each dev environment may be slightly different, so a checked-in TiApp.xml may have slightly different versions in them. If you find differences, make sure your environment is totally up to date.

- **"--keystore value "undefined" not valid" when building or running**

The Package configuration of the Keystore location is invalid. This happens irregularly. To fix, check the Package configuration's keystore parameter, click Browse and select the keystore again. Build and runs will now work.

- **Problems compiling/building Android projects**

Note that the version of Android installed and the version of Titanium installed are extremely important to keep in line. For example:

- Titanium 3.3.0 is valid only up to version 20 of the Android SDK.
- Titanium 3.5.1 is valid only up to version 22 of the Android SDK.

If later versions of the SDK are installed, Titanium will always use the latest versions, regardless of whether they are later than the maximum values allowed, which means you will receive errors when compiling, usually around the command 'aapt'. Check the compatibility matrix (http://docs.appcelerator.com/platform/latest/#!/guide/Titanium_Compatibility_Matrix), and ensure you only have the correct version installed of the SDK.

Warning: Additional notes will be added here

1.3 Titanium Module Development

Note: This process has been validated with Appcelerator Studio and Titanium SDK 5.0.2, although upgrades to some of the required programs were completed.

1.3.1 Development Environment

Note: Always check the latest documentation resources here:

- http://docs.appcelerator.com/platform/latest/#!/guide/Installing_Titanium_Advanced_Tools

Requirements:

- Oracle Java 6 (latest subversion) JDK 32-bit

- Android NDK
- Cygwin - NDK dependency
- GMake - NDK dependency
- Gperf - NDK/Windows 7 dependency
- ActivePython
- Scons
- Ant

Note: The majority of these installers will be available in P:\EPOD_Development\Development Tools\Android Module Building\

Titanium must then be configured for creating and building modules

1.3.1.1 Oracle Java 6

Oracle Java 6 (latest subversion) JDK 32-bit (NOT Java 7) - see http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html#jdk-6u45-oth-
The latest version at the time of writing is in P:\EPOD_Development\Development Tools\Android Module Building\jdk-6u45-windows-i586.exe.

Once installed, set JAVA_HOME to the installation directory (e.g. C:\Program Files (x86)\Java\jdk1.6.0_45\bin).

1.3.1.2 Android NDK

Installing the NDK on your development computer is straightforward and involves extracting the NDK from its download package.

Before you get started make sure that you have downloaded the latest Android SDK and upgraded your applications and environment as needed. The NDK is compatible with older platform versions but not older versions of the SDK tools. Also, take a moment to review the System and Software Requirements for the NDK, if you haven't already.

To install the NDK, follow these steps:

From the table at the top of this page, select the NDK package that is appropriate for your development computer and download the package. Uncompress the NDK download package using tools available on your computer. When uncompressed, the NDK files are contained in a directory called android-ndk-<version>. You can rename the NDK directory if necessary and you can move it to any location on your computer. This documentation refers to the NDK directory as <ndk>. You are now ready to start working with the NDK.

1.3.1.3 ActivePython

- Install Python (>= v2.5)
- Add in your PATH or configured with PYTHONHOME.

1.3.1.4 gperf

- Download or copy the zip,
- Unzip to a preferred directory
- Add the path to the bin directory to your PATH e.g. C:\gperf-3.0.1-bin\bin.

1.3.1.5 ant

- Version 1.7.1 or above
- Download or copy the zip
- Unzip to a preferred directory
- Add the path to the bin directory to your PATH e.g. C:\apache-ant-1.7.1.

1.3.1.6 Install the Eclipse Java Development Tools plugin

From Titanium Studio:

- First we enable the default (Helios or Indigo) software repository, and then find and install the JDK toolkit.
- Enable the Proper Software Repository
    - ♦ select the Help / Install New Software... menu

- under the top Work with drop-down menu, click the link in the phrase, Find more software by working with the Available Software Sites preferences
- enable the Eclipse Helios or Indigo Update Site
- click the OK button

- Find and Install the Java Tooling
  - using the Work with drop-down menu, select Eclipse Helios Update Site
  - wait for the package list to be populated
  - select the Programming Languages / Eclipse Java Development Tools package from the list
  - click the Next button
  - click the Next button on the Install Details screen that follows
  - accept the license agreement
  - click the Finish button
  - once the installation is complete, click the Restart Now button

1.3.1.7 Add Titanium scripting

- From Titanium studio, go to Window/Preferences, choose Studio/Platforms, copy the Titanium SDK home value.
- Add this to the Windows PATH.
- From the command line, may need to run 'titanium setup' to configure the CLI.

1.3.1.8 Switch Titanium workspace

As an additional note, you need to make sure the path to your Titanium Studio workspace does not contain spaces. This is a limitation of the Android NDK.

- From the File menu, select Switch Workspace > Other... to open the Workspace Launcher.
- In the Workspace Launcher, browse to the workspace that you want to switch to.
- Click OK.

You can then import your projects from the old to the new workspace.

1.3.1.9 Check the PATH

Once installed all need to be available in your environments path as well as the following environmental variables. Examples:

- ANDROID_HOME - C:\Program Files (x86)\Android\android-sdk
- ANDROID_NDK - C:\Program Files (x86)\Android\android-ndk-r8b
- ANT_HOME - C:\apache-ant-1.7.1
- GTK_BASEPATH - C:\Program Files (x86)\GtkSharp\2.12\
- JAVA_HOME - C:\Program Files (x86)\Java\jdk1.6.0_27
- Path - C:\Python27\;C:\Python27\Scripts;%ANDROID_NDK%\build;%ANDROID_NDK%;%JAVA_HOME%\bin;%ANDROID_HO Files (x86)\GnuWin32\bin;C:\Program Files (x86)\gperf-3.0.1\bin;%ANT_HOME%/bin;

1.3.2 Setting up Titanium Preferences

- Click *Window/Preference*
- Click *Studio*
- Click *Platforms*
- Click *Android*
- Enter the Android NDK home location

1.3.3 Creating a Module

Note: Always check the latest documentation resources here:

- http://docs.appcelerator.com/platform/latest/#!/guide/Creating_a_New_Titanium_Module

- Click *File/New/Mobile Module Project*
- Enter details, moving between with the **Next** button
    - Proj Loc:
    - Name: Lowercase, no symbols or spaces
    - Module: com.obslogistics.<name>
    - Target: 5.0.2 (or latest Titanium module)
    - Deploy: Android
    - Manifest: As required
    - Install Platform: 4.4.2 or latest supported
- Click **FINISH**

Wait for the module to be created.

1.3.4 Adding libraries to a module

Libraries are manually added, so it's likely that the projects will not load the requisite JAR files, as they will be in a different place on your machine.

- Paste JAR files into *<app>\android\lib* folder.
- Right-click on the project and select *Project Properties/Java Build Path/Libraries/Add JARs*
- Select the required JARs and click OK.

1.3.5 Building Modules

Click on the project, then click on the Package icon. If the installation process is followed above, there should be no issues.

1.4 SVN

1.4.1 Moving a repository

You need SVN tools installed on the new server. See here:

- P:\Development\EPOD_Development\Development Tools\Apache-Subversion-1.12.2.zip
- https://www.visualsvn.com/downloads/ - use Apache Subversion command line tools.

1.4.1.1 Step 1: Backup your old Repository

The first thing you need when moving from one server to another is a dump of your subversion repository.

```
svnadmin dump /path/to/repository > repo_name.svn_dump
```

The dump file contains all the revisions you have ever made to your svn repository, so it will probably be quite large (it even includes files you may have deleted in a previous revision).

1.4.1.2 Step 2: Create the new Repository

Now, simply transfer the dump file on to your new subversion server, and create an empty repository:

```
svnadmin create /path/to/repository
```

1.4.1.3 Alternative to Step 1/2: Copy the repository

Copy the repository directory to a new server, as long as you want an identical copy and history, and the machines are the same operating system.

1.4.1.4 Step 3: Import your old repository into the new one

Next import your dump file:

```
svnadmin load /path/to/repository -F repo_name.svn_dump
```

You may want to force subversion to use the same UUID for the new repository as the old repository. To do this add --force-uuid to your svnadmin load command. In my case I wanted to do this. If you have already loaded your repository, there is a way to set the UUID at a later date, check the docs.

1.4.2 FAQ's

What if someone committed a new revision to the old server during installation?

You can easily import the new revision, by creating an incremental dump on the old server:

```
svnadmin dump --incremental -r 1234 /path/to/repository > rev1234.svn_dump
```

Copy the file to the new server.

Now to import that revision on your new server:

```
svnadmin load /path/to/repository -F rev1234.svn_dump
```