Cross Platform Mobile Application Development using Titanium

Abu Sayed Chowdhury and Brett Duncan

Department of Computer Science, Georgia State University, USA

Abstract. There has been a large growth in mobile applications in recent years, and multiple mobile platforms have emerged. When developing mobile applications supported by multiple platforms, such as iOS or Android, it usually requires developing for each platform separately using the development tools provided by the platform vendor. Cross-platform mobile development approaches have emerged as a result, to minimize the development costs when developing for multiple platforms. For this project, we explore Appcelerator Titanium, an Eclipse based IDE for developing mobile application for multiple platforms. The developer writes code in JavaScript and the app is processed with a JavaScript interpreter. The application we develop is Conway's Game of Life for Android, BlackBerry, and mobile web. We will provide an assessment of Titanium based on the development experience.

1 Introduction

Mobile applications have experienced much growth in recent years, and several mobile platforms have emerged, such as Android, iOS, Windows Phone, and more [1–6]. If a developer wants to develop an application for multiple platforms, they must download the software development tools from the platform vendor of the platform that they want to develop for. Each platform has a different language and tools: Android requires Eclipse and uses the Java programming language, iOS requires Xcode and uses the Objective-C programming language, and Windows Phone requires Visual Studio and uses the C# programming language. Developing mobile applications for multiple platforms in this fashion requires learning of multiple languages and tools, leading to high development costs. The type of apps created using the vendor supplied tools are purely native. The UI elements have a look specific to the platform, and the performance of the apps is the best compared to non-vendor supplied tools. Native apps also have the most comprehensive access to the hardware device functions such as the accelerometer, the microphone, the GPS, etc.

Cross-platform development tools have emerged for the purpose of assisting developers develop for multiple platforms with minimal effort. Developers want to create apps that still have the native look and feel and similar performance to purely native apps. The type of mobile apps generated by cross-platform tools can be categorized as web apps, hybrid apps, interpreted apps, and generated apps [7].

1. **Web apps** are based on HTML and JavaScript. Web apps have the advantage that they don't have to be installed and are easiest to deploy to multiple platforms. The disadvantages of web apps are that they have limited access to native device features and user interface (UI) components. It also and a longer time to load the web page compared for a web app compared to native apps [7]. The performance of web apps is also not as good compared to native apps.
2. **Hybrid apps** are a combination of web and native apps. They are also built using HTML and JavaScript. They embed HTML apps in a native container [7] and are made to simulate the look and feel of a native application. An API is provided for accessing native functionality [4].

3. **Interpreted apps** provide APIs for native device features, and the app is packaged with an interpreter [4]. The app being packaged with the interpreter can make the application size relatively large, as discussed later in this paper. Titanium, the tool used in this paper, falls into this category. Titanium uses JavaScript, and a JavaScript interpreter is packaged with the app. A different interpreter is used for each supported platform.
   Interpreted apps have a slight performance hit because of the extra time needed to interpret the app. Interpreted apps also have limited access to hardware and APIs for accessing native features.
4. **Generated apps** are compiled in a similar way that native apps are, and a platform-specific version of the app is generated for each desired platform [7]. Generated apps have performance comparable to native apps because native code is generated. However, generated apps are usually limited to simple data-driven apps.

2 Tool Used

The cross-platform mobile application development tool we chose to use was Titanium Studio version 3.4.1 [8]. Titanium is an Eclipse based IDE in which developers write code in JavaScript and XML. The apps generated by Titanium are native. System functions can be accessed by APIs that Titanium provides. During compilation, Titanium packages the source code with a JavaScript interpreter. When the app is run, the source code is processed by the JavaScript interpreter [1, 4].

Titanium uses the Alloy framework for application development, which uses the model-view-controller architecture [8]. Figure 1 shows the project structure of a Titanium mobile app.

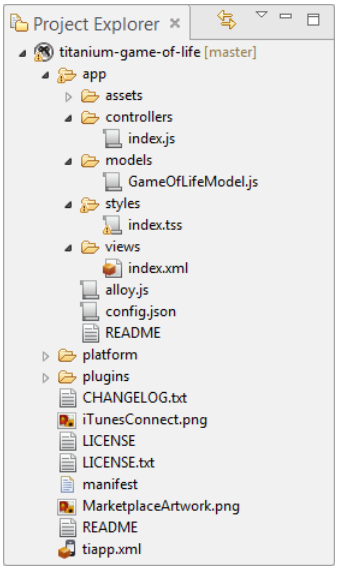

Fig. 1. Project Structure of Titanium Mobile Apps.

2.1 Alloy Models

Alloy models contain the business logic, rules, data, and state of the application, and are located in the app/models/ directory. It inherits from Backbone.js, which is a JavaScript library providing a JSON interface for creating models. Figure 2 shows an example of an alloy model. The JavaScript file book.js describes a book object. It contains two String fields called title and author. If either field

is left blank, it will default to a dash. The “sql” type configures Backbone to use the SQL adapter to sync with a SQLite database on Android/iOS to access a table in the database called ‘books’.

app/models/book.js

```
exports.definition = {
       config: {
       "columns": {
       "title": "String",
       "author": "String"
       },
       "defaults": {
       "title": "-",
       "author": "-"
       },
       "adapter": {
       "type": "sql",
       "collection_name": "books"
       }
     }
}
```

Fig. 2. Example Alloy Model. [8]

2.2 Alloy Views

Allow views provide native GUI components for the application, and are located in the apps/views/ directory. Views are comprised of XML markup files and Titanium style sheets. Titanium style sheets (TSS) files use JSON-like syntax to define attributes of elements in XML files, in a similar way in which Cascading Style Sheets (CSS) define attributes for elements in HTML files. Examples of an Alloy view and a TSS are shown in Figures 3 and 4.

app/views/index.xml

```
<Alloy>
    <Window class="container">
        <Label id="label"
onClick="doClick">Hello, World</Label>
    </Window>
</Alloy>
```

Fig. 3. Example Alloy View in an XML File. [8]

UI elements can also be created from inside JavaScript files using the Titanium.UI module [8]. The Titanium.UI module can also create native GUI components. To create GUI components like a Button or a TextArea, you would call Titanium.UI.createButton or Titanium.UI.createTextArea respectively.

```
app/styles/index.tss

".container": {
        backgroundColor:"white"
},
"#label": {
    color: "#999" /* gray */
}
```

Fig. 4. Example Alloy TSS. [8]

2.3 Alloy Controllers

Alloy controllers contain the application logic for controlling the UI and communicating with the model, and are located in the apps/controller/ directory. Figure 5 shows an example of an Alloy controller. In this controller, a function is defined for handling the event when an element is clicked. As shown in Figure 3, the Label element has an onClick attribute assigned to "doClick", so if this element is clicked or tapped, the doClick function would be called. The Label element has an id of "label", and in Alloy controllers, elements from XML files can be accessed by using $.id notation. For example, in the controller in Figure 5, the Label element is accessed using the notation $.label.

```
app/controllers/index.js

function doClick(e) {
    alert($.label.text);
}
$.index.open();
```

Fig. 5. Example Alloy Controller. [8]

3 Application Developed

3.1 Conway's Game of Life Rules

The application we developed using Titanium was Conway's Game of Life [9]. Conway's Game of Life is a game played on an $n \times n$ square board which simulate births, deaths, and survivals of cells. Cells can occupy any point on the board, and a cell contains eight neighbors to the top left, top, top right, left, right, bottom left, bottom, and bottom right. The simulation advances in single units at a time. The following are the rules for determining survival or birth of a cell:

1. A live cell survives in the next generation if it has two or three live neighboring cells.
2. A live cell dies if it has one or fewer live neighbors
3. A live cell dies if it has four or more live neighbors.
4. A cell is born if an empty cell has three live neighbors.

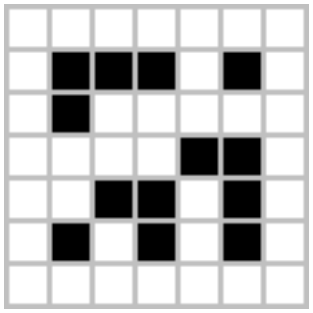

Fig. 6. Example Game of Life board.

Figure 6 shows an example of a game of life board. The black shaded squares represent live cells, and the white cells represent dead cells.

The game of life continues until no more cells are active, or the cells get stuck in a still life or oscillator state. Figure 7 shows examples of still life structures, and Figure 8 shows one instance of an oscillator state.

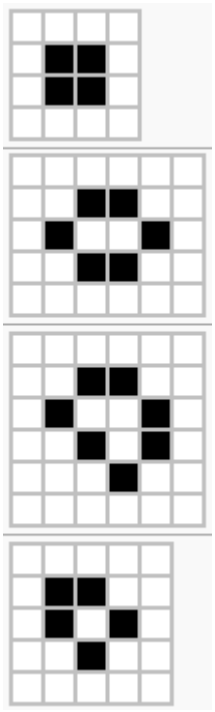

Fig. 7. Examples of still life structures.

3.2 Development Process

The development of the Game of Life app only occured in the app/controllers/index.js and app/views/index.xml. A model file was not created because models are required to have an exports.definition as shown in Figure 2. A model file containing an two-dimensional array representing a model of the board was desired, but the “columns” field did not appear to support an array data type, and creating an array outside of the exports.definition generates an error and is not allowed.

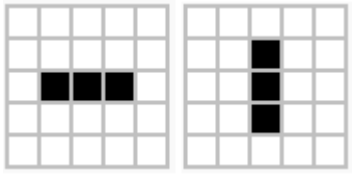

Fig. 8. One instance of an oscillator state.

The board is set up as a 20 $\times$ 20 grid. A start button is below the grid. When the app starts, cells are randomly generated to be alive or dead. Live cells are colored blue, and non live cells are white. The user can touch/click cells to toggle them to be alive or dead. When the start button is pressed, the game starts and continues until all cells are dead or are in a still life or oscillator state.

To generate the grid, individual `View` elements are used to build it from scratch, since the Titanium API did not appear to have a GUI element for a table with multiple columns. The `View` elements were generated using the Titanium.UI.createView() function call. The function takes a JSON object for describing properties of the `View`. We defined each of our `View` elements to have a height and width of 10 (platform specific) units, a background color of white (some are randomly changed to blue at a later point), a black border, and a function that toggles the background. Each `View` gets an event listener added that if it is clicked, the function that toggles the background color is called.

The start button is created from within the app/views/index.xml file. It is given an onClick attribute of doClick in index.xml, as shown in Figure 9.

```
<Alloy>
    <Window class="container">
        <Button id="startButton" onClick="doClick" title="Start"
            top="80%" width="100" height="50" />
    </Window>
</Alloy>
```

Fig. 9. The contents of app/views/index.xml

In the main controller file of index.js, the doClick() function, as shown in Figure 10, is implemented to start or stop the game depending on if it is active or not.

```
function doClick(e) {
    if (this.id == 'startButton') {
        if (!gameActive) {
            startGame();
        }
        else {
            stopGame();
        }
    }
}
```

Fig. 10. Some contents of app/controllers/index.js, implementing the doClick function for the start button.

Figure 11 shows the startGame() and stopGame() function implementations.

```
//Starts the game, game advances in half second intervals
function startGame(e) {
        $.startButton.title = "Stop";
        gameActive = true;
        gameInterval = setInterval(tick, 500);
}

//Stops the game
function stopGame(e) {
    $.startButton.title = "Start";
    gameActive = false;
    clearInterval(gameInterval);
}
```

Fig. 11. The functions to start and stop the game.

As shown in Figure 11, the startGame function calls the setInterval function, which is a pure JavaScript function that calls a function every specified number of milliseconds. In our code, setInterval(tick, 500) calls the tick() function every 500 milliseconds. The tick() function is a function we defined that updates the game grid based on the rules of the Game of Life. Figure 12 shows the implementation of our tick() function.

Figure 13 shows the implementation of the cellAliveNextTick() function, which the tick() function uses. This function checks all eight neighbors and returns true or false on whether a cell is alive at location i, j at the next tick.

Figure 14 show the variable initializations of the app. The statement $.index.open() is used to open the app window when the app is launched.

4 Result

We developed Android, BlackBerry, and mobile web versions of the Game of Life app. To generate the Android and BlackBerry versions of the apps, the Android software development kit (SDK) tools and the BlackBerry native development kit (NDK) were downloaded, as required. We did not develop an iOS version, because the iOS SDK requires an Apple computer, which we did not have access to. The web version of the app did not require any additional downloads. Figure 15 show the resulting apps for Android, BlackBerry, and the mobile web respectively. The Android version was ran on a physical device. The BlackBerry version was ran on a BlackBerry simulator, which runs in VMWare. The mobile web version was run on Firefox on a PC.

The Android version took the longest to compile. The `View` elements had slow loading times. As the app loaded up, you could see each cell of the grid load up one by one. The application size was 18.08 MB. By comparison, two other Game of Life applications available from the Google Play store are only 250KB [10] and 1.3 MB [11]. The performance of the Android app slightly lagged when first starting the game, but after a few seconds, the game advances in 500 millisecond intervals as expected.

The BlackBerry version also had a long compile time, and the BlackBerry simulator takes several minutes to start up. There were no loading issues for the `View` elements, however. The `View` elements did not have the black borders around them like the Android and mobile web version counterparts. The performance of the BlackBerry version was smooth with no lagging.

The mobile web version took the least amount of time to compile. The performance of the app was smooth with no lagging.

```
function tick() {
    if (numLiveCells == 0) {
        stopGame();
    }
    //the survival table is used to determine if a cell is occupied or not
    //for the next time unit
    var survivalTable = [];

    for (i = 0; i < tableRows; i++) {
        for (j = 0; j < tableCols; j++) {
            if (cellAliveNextTickAt(i, j)) {
                survivalTable[i + tableRows*j] = 1;
            }
            else {
                survivalTable[i + tableRows*j] = 0;
            }
        }
    }

    //Use survival table to adjust game table for next time unit
    for (i = 0; i < tableRows; i++) {
        for (j = 0; j < tableCols; j++) {
            if (survivalTable[i + tableRows*j] == 1) {
                if (!cellExistsAt(i, j)) {
                    tableView[i + tableRows*j].backgroundColor = 'blue';
                    numLiveCells++;
                }
            }
            else {
                if (cellExistsAt(i, j)) {
                    tableView[i + tableRows*j].backgroundColor = 'white';
                    numLiveCells--;
                }
            }
        }
    }
}
```

Fig. 12. The tick() function.

5 Reflection

Overall, Titanium was straightforward to use, and had a good documentation. We learned that Titanium is more fit for making business oriented database driven applications because of the Alloy framework allowing you to make models with fields and allowing easy connection to a SQL database. Titanium also has the advantage of being able to generate purely native apps.

Some limitations of Titanium are that it has a large application size. The application size on the physical Android device we tested on was 18.08 MB, while two other Game of Life apps from the Google Play store were only 250KB and 1.3 MB. Titanium has a limited API; there is no Titanium `View` element that allows a JTable equivalent, i.e. a table with multiple rows and columns. The table/game grid had to be generated with individual Titanium `View` elements from scratch using nested for loops. The loading time of the `View` elements on the Android version was slow; you could see each `View` element loading slowly, one by one, line by line. Titanium also has a slow compile time for the Android and BlackBerry versions.

```
function cellAliveNextTickAt(i, j) {
    var numNeighbors = 0;
    var isAlive = cellExistsAt(i, j);

    if (i > 0 && cellExistsAt(i - 1, j)) {
        numNeighbors++;
    }
    if (i > 0 && j > 0 && cellExistsAt(i - 1, j - 1)) {
        numNeighbors++;//top_left neighbor
    }
    if (i > 0 && j < tableCols - 1 && cellExistsAt(i - 1, j + 1)) {
        numNeighbors++;//top_right neighbor
    }
    if (j > 0 && cellExistsAt(i, j - 1)) {
        numNeighbors++;//left neighbor
    }
    if (j < tableCols - 1 && cellExistsAt(i, j + 1)) {
        numNeighbors++;//right neighbor
    }
    if (i < tableRows - 1 && cellExistsAt(i + 1, j)) {
        numNeighbors++;//bottom neighbor
    }
    if (i < tableRows - 1 && j > 0 && cellExistsAt(i + 1, j - 1)) {
        numNeighbors++;//bottom_left neighbor
    }
    if (i < tableRows - 1 && j < tableCols - 1 && cellExistsAt(i + 1, j + 1)) {
        numNeighbors++;//bottom_right neighbor
    }
    return (
        (isAlive && (numNeighbors == 2 || numNeighbors == 3))
        ||
        (!isAlive && numNeighbors == 3));
}
```

Fig. 13. The cellAliveNextTick() function.

```
$.index.open(); //Opens this window.

var tableRows = 20;
var tableCols = 20;
var tableHeight = 10;
var tableWidth = 10;
var tableTop = 10;
var tableLeft = 10;
var numLiveCells = 0;

var tableView = []; //Array holding the View elements that make up the table.

var gameActive = false;

var gameInterval; //Variable that setInterval() assigns to.
                  //Needed to cancel the interval with clearInterval(gameInterval)
```

Fig. 14. Variable initializations.

Another limitation was that Titanium's Alloy model restricts you to only being able to define Alloy models, as in Figure 2. One cannot omit the exports.definition initialization and create their

Fig. 15. The Game of Life mobile applications for Android, BlackBerry, and mobile web, respectively.

own model file without an error being generated. This limited our development process as we were forced to define our model in the same file as the controller, which was in apps/controllers/index.js.

6 Conclusion

In conclusion, we believe Titanium is a good cross-platform mobile application development tool for creating data-driven mobile applications. It generates native apps using the interpreter approach. It uses the JavaScript language, which is widely known already, so it is likely a developer already knows the language. If not, it is one of the easier languages to pick up. Although the application size and the performance is not up to par, we recommend Titanium for developing cross-platform mobile applications.

References

1. J. Ohrt and V. Turau, "Cross-Platform Development Tools for Smartphone Applications," *Computer*, vol. 45, no. 9, pp. 72–79, Sept 2012.
2. A. Ribeiro and A. da Silva, "Survey on Cross-Platforms and Languages for Mobile Apps," in *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, Sept 2012, pp. 255–260.
3. E. Angulo and X. Ferre, "A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX," in *Proceedings of the XV International Conference on Human Computer Interaction*, ser. Interacción '14. New York, NY, USA: ACM, 2014, pp. 27:1–27:8.
4. A. Sommer and S. Krusche, "Evaluation of Cross-Platform Frameworks for Mobile Applications," in *SE 2013*, 2013, pp. 363–376.
5. M. Palmieri, I. Singh, and A. Cicchetti, "Comparison of cross-platform mobile development tools," in *Intelligence in Next Generation Networks (ICIN), 2012 16th International Conference on*, Oct 2012, pp. 179–186.
6. I. Dalmasso, S. Datta, C. Bonnet, and N. Nikaein, "Survey, Comparison and Evaluation of Cross Platform Mobile Application Development Tools," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, July 2013, pp. 323–328.
7. S. Xanthopoulos and S. Xinogalos, "A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications," in *Proceedings of the 6th Balkan Conference in Informatics*, ser. BCI '13. New York, NY, USA: ACM, 2013, pp. 213–220.
8. "Titanium 3.X - Appcelerator Docs," March 2015. [Online]. Available: http://docs.appcelerator.com/titanium/latest/
9. M. Gardner, "Mathematical Games–The Fantastic Combinations of John Conway's New Solitaire Game "life"," *Scientific American*, vol. 223, pp. 120–123, October 1970.

10. G. Hirlekar. (2013) Conway's Game of Life. [Online]. Available: play.google.com/store/apps/details?id=com.gaurav.gameoflife
11. Firefly. (2014) The Game of Life. [Online]. Available: play.google.com/store/apps/details?id=simon.jeu.LeJeuDeLaVie