Received 9 March 2015
Revised 9 March 2015
Accepted 5 April 2015

# A framework for programmatically designing user interfaces in JavaScript

Henry Larkin
*School of IT, Deakin University, Melbourne, Australia*

## Abstract

**Purpose** – The purpose of this paper is to investigate the feasibility of creating a declarative user interface language suitable for rapid prototyping of mobile and Web apps. Moreover, this paper presents a new framework for creating responsive user interfaces using JavaScript.

**Design/methodology/approach** – Very little existing research has been done in JavaScript-specific declarative user interface (UI) languages for mobile Web apps. This paper introduces a new framework, along with several case studies that create modern responsive designs programmatically.

**Findings** – The fully implemented prototype verifies the feasibility of a JavaScript-based declarative user interface library. This paper demonstrates that existing solutions are unwieldy and cumbersome to dynamically create and adjust nodes within a visual syntax of program code.

**Originality/value** – This paper presents the Guix.js platform, a declarative UI library for rapid development of Web-based mobile interfaces in JavaScript.

**Keywords** JavaScript, Declarative UI, Mobile Web applications, Programming languages, User interface

**Paper type** Research paper

## 1. Introduction

Declarative user interfaces have unique advantages for the development of applications. Traditional programming syntax and API development make it difficult to visualize layouts, often leading to cumbersome and "shortest solution" approaches to interfaces. Although, ideally, artists may design interfaces, there are many situations where the interface is so dynamic that the majority, if not all, of the application's interface must be created within program code.

Figure 1 shows an example of a dynamically generated user interface for a JavaScript-based mobile application for learning Korean vocabulary. In it, individual game tiles are created dynamically, based on how many flip tiles to include in the game based on screen size and game complexity. Furthermore, the colors of tiles, the text and the flipping animations are all dynamic. Such dynamic design is not possible in purely graphical images or even HTML. And yet the program code required to generate dynamic components in JavaScript is overly complex.

The following four examples demonstrate the shortest code approaches, in four different frameworks, to programmatically create a HTML header 1 Document Object Model (DOM) element (H1) and append it to a div element. In Algorithm 1, this is done in pure JavaScript (with the included text variable declaration). As can be seen, this example is overly lengthy and difficult to read. In Algorithm 2, this is done using jQuery[1], a popular library that simplifies some JavaScript actions. In Algorithm 3, a

International Journal of Pervasive Computing and Communications
Vol. 11 No. 3, 2015
pp. 254-269

1742-7371
DOI 10.1108/IJPCC-03-2015-0014

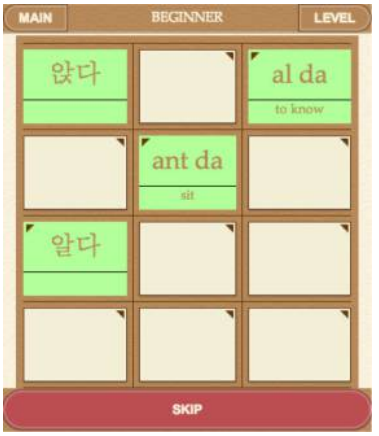


**Figure 1.**
Example of a dynamically generated interface

declarative user interface (UI) framework Nano.js[2] is used. This framework does simplify the code required when compared with JavaScript, but it is still overly verbose. In Algorithm 4, the code solution is shown using the JavaScript framework Guix.js, the framework presented with this paper. As can be seen, its readability is far more ideal than the previous examples, foregoing the need for constructors, object parameters and named parameters within those objects:

Algorithm 1. Using native JavaScript example

```
var text = "Title Text";
var div = document.createElement("div");
var h1 = document.createElement(h1);
h1.innerHTML = text;
div.appendChild(h1);
```

Algorithm 2. Using jQuery example

```
$("<div/>").append("<h1>"+text"</h1>");
```

Algorithm 3. Using Nano.js example

```
new nano({parent: nano.body(), tag: "div"})
.add({tag: 'h1', text: text});
```

Algorithm 4: Using this paper's framework example

```
div(h1(text));
```

Against this background, in this paper, we:

- Review existing JavaScript frameworks for declaring user interfaces (Section 2).
- Present an overview of the approach of this paper, addressing the primary methods that overcome the limitations present within JavaScript (Section 3).
- Introduce an approach to solving multi-lingual issues in user interface development (Section 4).

- Introduce an approach for resolving run state on devices where the application run life cycle may be short (Section 5).
- Present selected case studies (Section 6).
- Eventually, in Section 7, we conclude.

## 2. Existing frameworks

In mobile Web applications (and native applications that use a Web component for their interface), JavaScript is the predominant language for development. There are several basic declarative UI languages for HTML and JavaScript-based applications currently in existence. Most of these frameworks, such as Knockout.js[3] (Algorithm 6) and Quilt.js[4] (Algorithm 6), provide a way of enhancing HTML code with access to data and JavaScript programming. However, they work only within HTML markup, and not in JavaScript itself. One of the frameworks that can be used in JavaScript is Nano.js[2], demonstrated in Algorithm 3 previously. It does provide a declarative framework for JavaScript, though the approach is only a basic framework, and vastly lacks the simplicity and clarity possible. There is room for a new framework to be developed:

Algorithm 5: Knockout.js example

```
There are <span data–bind="text:myItems().count"></span> items
```

Algorithm 6: Quilt.js example

```
<div>
  <%= locales.translate('welcome') %>
</div>
<% if (model.get('visible')) { %>
<div>
  <img src="<%= model.photo_url %>"/>
  <a href="<%= model.get('url') %>">
    <span><%= escape(model.get('first_name')) %></span>
    <span><%= escape(model.get('last_name')) %></span>
  </a>
  <p><%= truncate('bio') %></p>
</div>
<% }; %>
```

There are several issues that arise in developing declarative UI frameworks for JavaScript. First, the type information available in JavaScript is limited when compared with other languages, where even arrays and objects have the same JavaScript type (typeof(array) and typeof(object) both return "object"). Being able to determine the type of the information passed within a node's constructor parameters is useful in determining what should be done with differing types of objects in a parameter list. In the simplest instance, for example, when calling an add() function on a node, different data types would be appended in different ways. Strings would be added as plain text to a component, objects that are a HTML DOM type would be appended as HTML elements and arrays would have all elements individually added, either as text or as HTML DOM elements recursively.

Another issue that arises is that HTML DOM elements have issues with saving event handlers when a DOM object is not attached to the DOM window. Due to the way events

are currently managed in most HTML rendering engines, elements need to be attached to a live window to function properly. A third issue is how to add multi-lingual support, which in this day and age is almost a necessity in planning, even if the initial language of an application would be in the native language of the developers. This does become an issue, as when rapidly designing an interface declaratively in program code, it is common for programmers to hardwire text strings. A final issue is how to handle the short runtime life cycle of mobile and Web apps, where apps are shutdown when not in use, even if not in use temporarily, to save random-access memory. It is a common user expectation that when an application is restarted, it remembers the user state and continue as if the application had never been shut down.

## 3. Approach

This paper describes the issues of and solutions to designing a declarative UI framework for JavaScript, Guix.js, a system for rapidly prototyping JavaScript-based mobile Web applications using a short amount of readable code. The system is designed to be as forgiving as possible and accepting as much as possible as valid input to create the interface. This is due to JavaScript's weakly typed system and the general lack of type checking that is a feature of many other compilers.

The essential perspective, from which short readable declarative code can be designed, is to center each DOM object creation around a function call of that DOM type's name. For example, to create a heading 1 (H1 tag), one would use h1(). To create a heading 2, one would use h2(); to create a div, one would use div(). In each instance, the function call instantiates the DOM element and any supplementary information needed by the rapid prototyping language. Furthermore, all arguments provided during the component's construction are treated as either subcomponents of that container or, in the case of a function, as event handlers. Consider the following example of creating a table with a single row and two columns (Algorithm 7). The table is instantiated after having a subcomponent row instantiated first. This row consists of two text elements added to each cell:

Algorithm 7: Creating a table with one row and two columns

```
table(row("Hello", "there"))
```

The need for the system to be accepting and forgiving of varying data types is important in JavaScript-based applications, as there is a general lack of IDE support for checking type data. This is due to the fact that JavaScript is a weakly typed language. Consider another example given below in Algorithm 8 for creating a ruby element (where transliterations are placed above words, e.g. 你好). Two versions of creating a ruby element are presented, each creating the same output. The ruby function in this framework supports both forms of adding subcomponents. In the first example, specific DOM elements, RB (ruby bottom) and RT (ruby top) can be added in any order. However, as shorthand, one could imagine many cases where a programmer simply types text directly into the ruby function's arguments, intending these to be the bottom and top elements of a ruby component. In this case, the ruby function caters for both situations where if text is provided, then the text is added to ruby bottom and ruby top instantiated components (in that order):

```
Algorithm 8: Two approaches to adding ruby
  ruby(rb("你好"),rt("nǐ hǎo"))
  ruby("你好", "nǐ hǎo")
```

The other essential design feature required for declarative UI languages is that it should be possible to customize a component on-the-fly, without needing to put the component into a variable. In Algorithm 9 below, two radio buttons are instantiated, with the first also becoming selected (toggled). Both are added to a paragraph tag without ever being assigned into variables. The paragraph tag itself also has a function called upon it, adding the group "gender" to each of its child elements (e.g. the two radio buttons). As components are instantiated and used anonymously, there is a need to chain functions, where each function returns a reference to the HTML DOM element being operated on. Without chaining, Algorithm 9 would be far more complex, with each radio button separately having to be stored in a variable, and then included in the paragraph container, as is done in other JavaScript helper libraries (see Algorithm 1 on the first page). Without function chaining, declarative user interfaces would not be possible:

```
Algorithm 9: Function chaining
  p(
    radio("Male").select(),
    radio("Female")
  ).groupChildren("gender")
```

*3.1 Handling variable function arguments*

To be able to deal with any input being provided into a DOM-instantiating series argument, there is a need to track the types of components, and how each component will handle both other components and primitive data types being added. To support this, several mechanisms are used.

First, each component is created with a function of the same name. All functions then call a helper function *htmlcode*, which instantiates the component with a range of additional functions. The "type" of a component, whether it's a native HTML component or a newly designed custom component, is then stored as a field within the component wrapper created by the htmlcode function.

One of the parameters of the htmlcode function is a map of actions to perform on child data types. This provides the ability for every component to have its own optional code to perform on inputs being added. This mapping system is the centerpiece of all components, which takes types (JavaScript primitive types, HTML DOM types or custom object types), and determines what to do with them by using an attached function. The function is given two parameters, the parent component and the child component. The function then executes whatever functionality is necessary to apply the child information to the parent.

Through the same logic, it is also possible for any developer to design their own components, which in turn consist of other components. Algorithm 10 below shows a code snippet of a radio component. In HTML, a radio input type has no feature of providing text alongside the radio button, which is a common feature requested in user interfaces. A more advanced radio component could then consist of the radio input and an optional label. The example below shows how the "text" provided to a radio button (in the form of a *string* data type) is used both as the value of the input component and the

text for the label. Although several details have been omitted (such as creating the label if it does not exist, and storing references to both the radio input and label to speed up access), it shows how a component can be created and can make use of its own sub-components. This allows programmers to design their own components that are not part of standard HTML or other JavaScript libraries, such as panels, menus, game tiles, etc:

Algorithm 10: Radio component snippet

```
function radio() {
    return htmlcode('div', {
        string:function(parent, txt) {
        // Add the value
        parent.find("input").attr("value",txt);
        // adjust the label
        parent.find("label").addChild(txt);
        }
    }).add(input().type("radio"));
}
```

For primitive types such as strings and functions, the type of the JavaScript object can be determined immediately using the JavaScript *typeof* operator. For DOM components such as in the case of RUBY, RB and RT components, the type of the component is stored as a data type value within the component's wrapper. Regardless, when adding elements, the type of an object is extracted and then an appropriate mapping is searched for. When no mapping exists, default actions are used to simply add the component, text or function to the DOM component. In the case of a function, the function is attached to an onclick handler for the component, and to a special object attached to the window for this user interface framework. It will be discussed next.

### *3.2 Dealing with events*

In many HTML rendering engines, events are often lost on DOM objects when they are detached from the DOM window. To solve this problem, each instantiated DOM element in Guix.js has attached to its data attribute, a unique reference in a user interface *events* object map. Inside this map, each DOM element's unique reference has a map, mapping event types to functions. Anytime an existing element is added to the DOM, its unique id is checked in the event table to see if it has events that need attaching, and then all of its subcomponents are recursively called to see if they too have events that need to have events attached.

Consequently, a special *destroy* action also exists, where removing a DOM element safely removes any event handlers attached before proceeding. This destroy function is available to all components. In this case, a programmer does need to be aware of events being mapped. Especially if a program is iteratively creating components and destroying them, there will be an issue with the event space mapping possibly becoming quite large. As JavaScript does not have a native deconstructor implementation, programmers must manually remove a component by calling *destroy* to remove all additional data that have been saved.

## 4. Multi-lingual support

An important consideration in designing mobile applications is to plan in anticipation of multi-lingual support. From a rapid prototyping prospective, multiple languages are rarely considered in the initial creation of an application. It is simply by way of convenience that programmers often hardwire strings into the application without giving thought to how they will later support translations in a simplified fashion. Guix.js overcomes this issue by treating every string as having the ability to reference language variables. The application makes use of a *language* field within the user interface address space. Here, any language constants can be added at anytime by any JavaScript files or loaded as text files through helper functions to assign conversions between constants as used in the application. Whenever a string is added to a component using the default add() function for text, the current language of the interface is first searched to see if a corresponding user interface variable is available, and if so, the mapping is applied. An example of this is shown in Algorithms 11 and 12. Algorithm 11 demonstrates creating a paragraph with the text "help", which is looked up in the language database and translated if an entry exists for the word in the user's language. This lookup matches whole words only, and is case-insensitive:

Algorithm 11: Using multi-lingual constants

```
P("help")
```

Algorithm 12: Multi-lingual specification

```
UI["languages"] = {
    "en":{
        "settings":"Settings",
        "help":"Help",
        "support":"Support",
    },
    "zh-cn":{
        "settings":"设置",
        "help":"帮助",
        "support":"支持"
    },
    "zh-tw":{
        "settings":"設置",
        "help":"幫助",
        "support":"支持"
    },
    "ko":{
        "settings":"설정",
        "help":"도움말",
        "support":"지원"
    }
}
```

The interface language can be changed at any time through the framework's API. When changed, all top-level components attached to the current window have their refreshText() function called, which in turn refreshes the text on all attached child components. This refresh will repeat a search, given the current (new) language, and draw text on components based on this lookup.

Figures 2 and 3 demonstrate how all text within the current window changes immediately between English and Korean when the interface language is changed in a "Settings" page, without reloading the page. This approach aids in the rapid prototyping of mobile applications, where strings can be hardwired and used in fully functional end-result applications, while still allowing the application to be open for future multi-lingual changes.

## 5. Loading and saving data

Another expectation from users of mobile applications is that the application state resumes where the user left the application. In mobile device scenarios, it is common for an application's usage to be interrupted, due to phone events (such as incoming texts), the user rapidly switching between applications or the user temporarily closing their phone to interact with the real world around them. The user expectation is that, when the application is returned to, the application state will resume precisely where the user left the application. In declarative UI programming, there is little room for breaking the declarative flow to search for any existing application state variables and include them in the interface. Rather, it would be ideal to have a mechanism where form-based data are automatically saved and loaded, given the form component's position and optionally a specific id.

To support this, all components within the user interface have the possibility of having their data and all subcomponents' data automatically saved in HTML5 data stores. This is accomplished using simple *save* and *load* functions available on any component, so long as the top-level component has a unique id to reference the saved information in the data store. When the *save* function is called, a name space is created in the HTML5 data store as *save.unique identifier.component name/index* (with

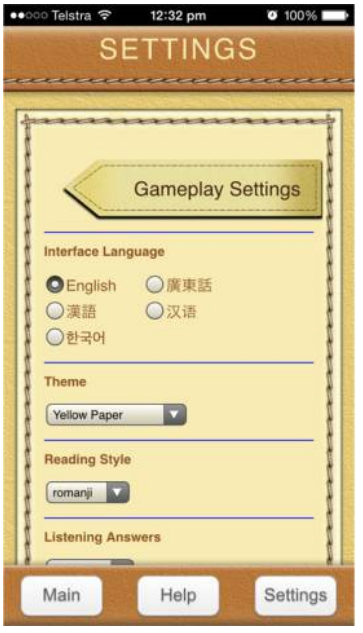


**Figure 2.**
English selected

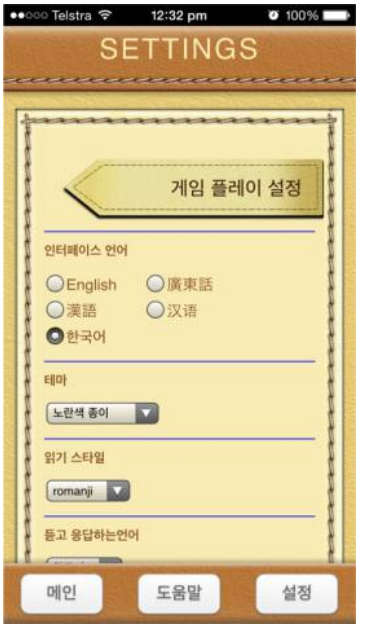


**Figure 3.**
Korean selected

recursive levels of unique identifiers/component names/indexes being used, as appropriate). At each level, a component is referenced by a sequential order, a name or an id. Ideally an id (which is unique) or a name (which does not have to be unique, only unique within the current container) would be provided to allow components to be found. This allows the interface to be more accurate in finding a matching component, particularly when an application is upgraded and user interface components may be added, removed or shifted. However, as previously stated, one of the core premises of this work is that the framework be as forgiving as possible. So even where the developer does not provide unique names, the fields' index positions will be used as a replacement. An example would be a login window, consisting of username and password fields, but without names on the elements (example shown in Figure 4). As this interface is unlikely to change, the framework can trust the programmer's lack of identification of components and reference each component by its sequential index.

When loading data, the reverse process is undertaken. Any component with an id can have data loaded and refreshed over the components in real time, through the load() function. All sub-components within the DOM tree are checked to see if data have been saved, although here, where a save data type does not match up with the form component, it is simply ignored. In the case of an interface being upgraded between


**Figure 4.**
Login user interface

versions, it is possible that some data might not be restored. However, the application will not stop working and will display as much data as it is able to, based on the indexing and/or identification of components and subcomponents.

Data are stored in HTML5 data stores, using the top-level component's id as the key. Each individual field is saved as its own entry within the database, with the id being constructed as each level's id, name or sequence number, separated by an underscore. For example, a "login" component with "username" and "password" fields would have two saved entries:

- login_username = …
- login_password = …

In the case where no ids are present on components, the components are indexed, as follows:

- login_0 = …
- login_1 = …

Alternatively, a UI global variable JSON_TABLES contains a list of specific IDs that are to be saved as JSON[5] representations. If a top-level id has been added to this list, then a JSON render of the data of all sub-components is stored as a text string within the database. Each JSON key is then an id, a name or an index position, depending on which one is available.

A further feature added in the Guix.js framework is that of auto-saving. In mobile applications, where an application may be closed due to being in a background state too long because a device may be low on memory, a common user expectation is for the application to seamlessly resume where the user left off, including all form field edits made. This framework provides an auto-save feature, which can be toggled on any component and will automatically save data of all form components contained at any depth within it. This is an extremely simple approach, from a programmer's perspective, of tracking the state of user input within an application.

In a practical sense, the auto-save function toggles a flag on the given top-level component, and recursively to all child components. Any newly added component will also inherit the status of the auto-save flag. During specific events, such as option toggle or text field blur, if the auto-save flag is set, then the data are stored into the HTML5 data store.

Algorithms 13 and 5 demonstrate the use of toggling auto-save on a settings form. It also demonstrates the use of several components. A form is created, consisting of four primary components and, optionally, any additional settings defined in a configuration variable elsewhere. A H3 tag is created, with multi-lingual support that may or may not be used. As it contains a text string pre-capitalized and ready for immediate output, if it is missing from the multi-lingual language specification then it will still render how the programmer envisions. Following this, three toggle buttons are introduced, which create a custom switch-like component for toggling between two options. A toggle component takes any text string to be added to the label of the toggle, and any array to be used as the values of the toggle buttons (which again make use of multi-lingual support). In the case where data types are used (e.g. true and false), these are looked up in the user interface language specification as "Progress Tracking_true" and "Progress

Tracking_false" reference keys. If values are found, then these are custom-written as the labels. If not found, default "true" and "false" keys are looked up (e.g. to provide generic "Yes" and "No" values). Otherwise, "True" and "False" are used as the text in the last instance.

Also note that a function can be added at any time in arguments, where the function provided is called any time the setting is changed. This provides the programmer with the ability to make quick calls when settings are changed or user interface buttons are toggled, like selected a game's skill level and having the current game stats reset. Also of note is that the toggleSetting() component is different from a toggle() component. The toggleSetting component automatically saves and loads the values of each entry in a unique global *settings* namespace within the application's local data store and within the JavaScript *window.settings* namespace. Guix.js provides an additional set of xxxSetting components that can be automatically used for global settings, without any further variable declarations required. They call the same interface construction logic as their original forms (e.g. toggleSetting calls toggle() component constructor, textSetting calls text() component constructor). However, in addition, they automatically save and load data whenever the data are changed, to the global *window.settings* namespace, as well as in a HTML5 data store *settings*. This is to facilitate the use of data that are used throughout the application for user app configuration. For example, toggleSetting creates a toggle component for a boolean setting, intSetting creates a textbox restricted to numbers and textSetting creates a textbox for working with string settings. As can be seen from the code in the example, the programming required by the programmer is extremely minimal to create a settings interface, complete with automatic loading and saving of settings:

Algorithm 13: Using auto-save on a form

```
form(
  (config.additionalSettings?
    config.additionalSettings : null),
  h3("Learning Settings"),
  toggleSetting("Skill Level",
    ["Beginner", "Advanced"],
    function() { GameSetup.reset(); }),
  toggleSetting("Progress Tracking",
    [true, false}),
  toggleSetting("Wordcap",
    [true, false],
    function() { GameSetup.reset(); })
).id("learning_settings").autosave(true);
```

A demonstration of the interface generated using Guix.js is shown in Figure 5. The first item included is the use of additional settings, where one additional toggle setting is included. Following this, a HTML header 3 is added, and then three toggle setting components. Each toggle setting component has a label (which in this particular CSS style is included before the toggle buttons). Following this are the two toggle options for each case. Note how that a selection in all cases is already made, as there will be either settings retrieved from the application's data store, or default setting values, for each of these keys.

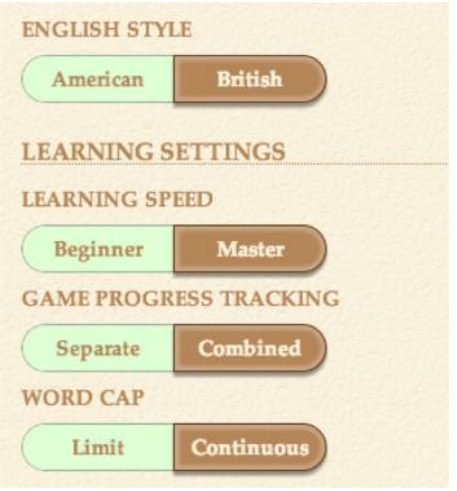


**Figure 5.**
Example settings interface

## 6. Case studies

Several interface elements will now be shown from one of the example applications built with this framework. To extend on from Algorithms 13 and 5 previously, consider the situation of including the previously detailed form inside a tabbed pane. Algorithm 14 demonstrates use of a tabs component in program code, where text strings become tab titles, and components become the tab content for each tab. This demonstrates how an interface can be "built up" in a completely dynamic approach, without the need for variable declarations or specific adder functions. The entire user interface can be constructed in a single declaration. Figure 6 shows the tabbed interface with the "Game" tab selected, and the previous "Settings" panel shown:

```
Algorithm 14: Using tabs
  tabs(
    "Game",
    form(
      // code from Example 13
    ).id("learning_settings").autosave(true),
    "Expert",
    form(),
    "Sync",
    form(),
  )
```

In the next scenario, a footer component for a spaced repetition system (SRS) learning game is created. The program code (Algorithm 15) describes three separate buttons being created, each placed in their own div for positioning purposes and finally placed in a single parent div. Each button has hardwired text ("incorrect", "unsure" and "correct"), which is looked up in the multi-lingual text mappings, where "incorrect" is replaced with "Wrong". Each button also has unique classes added, based on the button styling requirements. Finally, each button also has its own event action to call when tapped. Figure 7 illustrates the rendered footer interface:

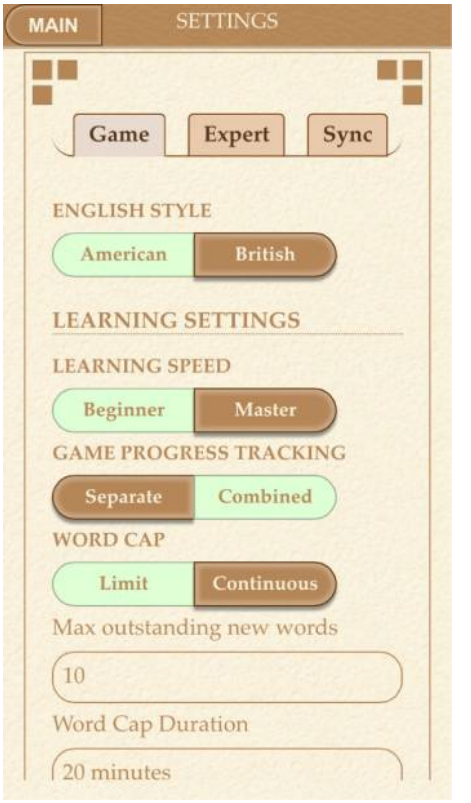


**Figure 6.**
Tabs example


**Figure 7.**
Footer UI rendered

```
Algorithm 15: Footer UI example
  this.footerResponse = div(
    div(button("incorrect").addClass("button–left red").onTap(function() {
      GameSetup.currentGame.setIncorrect();
    })).addClass("ui–block–a"),
    div(button("unsure").addClass("button–center yellow").onTap(function() {
      GameSetup.currentGame.setUnsure();
    })).addClass("ui–block–b"),
    div(button("correct").addClass("button–right green").onTap(function() {
      GameSetup.currentGame.setCorrect();
    })).addClass("ui–block–c")
  ).addClass("ui–grid–b");
```

In this final scenario, the basics of a four-button multiple-choice interface are demonstrated (Algorithm 16). Note that not all of the interface is included for the purposes of brevity. Four specific buttons are created as the user-selectable

multiple-choice options. These are custom user interface components created specifically for this application, as the buttons are one-way toggleable (they "lock down" when pressed), revealing additional information. This button type is labeled "fastCharButton" and is written into the Theme object for this application. The main interface is then constructed as a 3 × 3 table, with the top and bottom rows having the multiple-choice buttons on the left and right sides, and the center row and cell having the question and optional audio button (if the question is an audio event). Figure 8 presents the completed interface, which includes the four multiple-choice buttons detailed in this scenario:

Algorithm 16: Multiple-choice UI example

```
this.buttonNodes = [
    Theme.fastCharButton().onTap(function(){GameSetup.currentGame.clicked(0);}),
    Theme.fastCharButton().onTap(function(){GameSetup.currentGame.clicked(1);}),
    Theme.fastCharButton().onTap(function(){GameSetup.currentGame.clicked(2);}),
    Theme.fastCharButton().onTap(function(){GameSetup.currentGame.clicked(3);})
];
this.mainInfo = table(
    tr(this.buttonNodes[0]," ",this.buttonNodes[1]),
    tr(div(this.audioButton, this.questionNode).addClass("question–mark")).colspan(3),
    tr(this.buttonNodes[2],Theme.inlineButton("?").addClass("skipbutton").onTap
(function(){GameSetup.currentGame.skip();}), this.buttonNodes[3])
);
```

## 7. Conclusion

Developer-friendly programming libraries and architectures appear to be the direction for the fast-paced and high-user-requirement world of mobile application development. Such frameworks require a modern look at the requirements and expectations of user applications within internationally pervasive and short runtime life cycle applications.

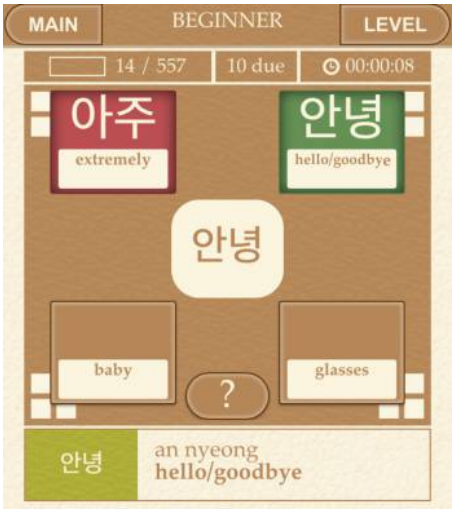


**Figure 8.**
Multiple-choice render

This paper presents a framework for rapidly developing mobile Web applications in JavaScript. It builds on existing principles from declarative UI languages. However, it differs from other JavaScript frameworks in that the user interface can be designed entirely from JavaScript. Furthermore, the system features modern user interface features such as multilingual support and being able to save and load the state of any set of HTML components.

## Notes

1. jQuery Foundation, jQuery, available at: http://jquery.com (accessed 6 March 2015).
2. Nano.js, available at: http://nanojs.org (accessed 6 March 2015).
3. Knockout.js, available at: http://knockoutjs.com (accessed 6 March 2015).
4. Quilt.js, available at: http://pathable.github.io/quilt (accessed 6 March 2015).
5. JSON, available at: www.json.org/ (accessed 6 March 2015).

## Further reading

Abams, M., Phanouriou, C., Batongbacal, A., Williams, S. and Shuster, J. (1999), "UIML: an appliance-independent XML user interface language", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 31 Nos 11/16, pp. 1695-1708.

Ballagas, R., Memon, F., Reiners, R. and Borchers, J. (2007), "iStuff Mobile: rapidly prototyping new mobile phone interfaces for ubiquitous computing", *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems 2007, ACM, New York, NY*, pp. 1107-1116.

Cassino, R. and Tucci, M. (2011), "Developing usable web interfaces with the aid of automatic verification of their formal specification", *Journal of Visual Languages & Computing*, Vol. 22 No. 2, pp. 140-149.

Dix, A. (1999), "Design of user interfaces for the web", paper presented at UIDIS'99 – User Interfaces to Data Intensive Systems, *Edinburgh*, 5-6 September, available at: www.hiraeth.com/alan/topics/web (accessed 6 March 2015).

Dubé, D., Beard, J. and Vangheluwe, H. (2009), "Rapid development of scoped user interfaces", *Proceedings of the 13th International Conference Human-Computer Interaction Part I: New Trends, Springer-Verlag Berlin, Heidelberg*, pp. 816-825.

Frank, C., Naugler, D. and Traina, M. (2005), "Teaching user interface prototyping", *Journal of Computing Sciences in Colleges*, Vol. 20 No. 6, pp. 66-73.

Giereth, M. and Ertl, T. (2008), "Design patterns for rapid visualization prototyping", *Proceedings of the 12th International Conference of Information Visualisation, IEEE, London*, pp. 569-574.

Helms, J. and Abrams, M. (2008), "Retrospective on UI description languages, based on eight years' experience with the user interface markup language (UIML)", *International Journal of Web Engineering and Technology*, Vol. 4 No. 2, pp. 138-162.

Meszaros, T. and Dobrowiecki, T. (2010), "Rapid prototyping of application-oriented natural language interfaces", *Proceedings of Information Technology Interfaces (ITI) 32nd International Conference, IEEE, Cavtat/Dubrovnik*, pp. 97-102.

Nebeling, M. and Norrie, M. (2012), "jQMultiTouch: lightweight toolkit and development framework for multi-touch/multi-device web interface", *Proceedings of the 4th ACM*

*SIGCHI Symposium on Engineering Interactive Computing Systems, ACM, New York, NY*, pp. 61-70.

Ni, L., Xu, Z., Wu, T. and He, W. (2013), "Visualizing linked data with javascript", *Proceedings of Web Information System and Application Conference (WISA), in Yangzhou 10-15 November, IEEE, Los Alamitos*, pp. 211-216.

Phanouriou, C. (2000), "UIML: a device-independent user interface markup", Virginia Tech Electronic Theses and Dissertations, VA Polytechnic Institute and State University, VA.

Rexilius, J., Jomier, J., Spindler, W., Link, F., König, M. and Peitgen, H. (2005), "Combining a visual programming and rapid prototyping platform with ITK", *Bildverarbeitung für die Medizin 2005*, Springer, Heidelberg, pp. 460-464.

Ssekakubo, G., Suleman, H. and Marsden, G. (2014), "A streamlined mobile user-interface for improved access to LMS services", *Proceedings of eLmL 2014 The Sixth International Conference on Mobile, Hybrid, and On-line Learning 23-27 March 2014 in Barcelona, Spain, IARIA, Barcelona*, pp. 92-101.

Trætteberg, H. (2007), "A hybrid tool for user interface modeling and prototyping", *Computer-Aided Design of User Interfaces V*, pp. 215-230.

Yanagisawa, H., Uehara, M. and Mori, H. (2008), "Interface implementation using ajax for web-based instruction set simulator", *Proceedings of Advanced Information Networking and Applications – Workshops, 22nd International Conference, IEEE, Okinawa*, pp. 1511-1516.

## About the author

Henry Larkin is a Lecturer of Mobile Applications at Deakin University. He obtained his PhD in wireless pervasive networks in 2005 at Bond University, followed by a postdoc at Aizu University in Japan. He has published over 30 papers in mobile and Web-related fields. His research interests include mobile applications, Web technologies, ubiquitous and pervasive computing and wireless ad hoc networks. Henry Larkin can be contacted at: henry.larkin@deakin.edu.au