Comparative study of frameworks for the development of mobile HTML5 applications

Tim Ameye
tim.ameye@student.kuleuven.be

Sander Van Loock
sander.vanloock1@student.kuleuven.be

Abstract

1 Introduction

2 Comparison criteria and related work

Many in-depth comparisons of HTML5 mobile frameworks already exists today. However, none of them are scientifically published or use a large proof of concept (POC) to validate their comparison. Blog posts like [**?**; **?**; **?**] all have their own criteria and methodology to assess different mobile frameworks. The overall applications of the criteria changes from use case to use case. [**?**] presents the chosen criteria and discusses each for each framework. [**?**] presents a whole bunch of criteria but all of them are discusses at once per framework. Thereafter, advantages and disadvantages are subtracted and proposed to the reader. [**?**] finally, presents their chosen criteria together with a scorecard and explanation of scores per criteria. Each framework gets evaluated based on the scores for each criteria.

All these blog posts compare more than two frameworks where some of them mobile and some are hybrid (see table 1). Other websites like [**?**; **?**] only focus on two mobile HTML5 frameworks while on the other side of the spectrum [**?**] tries to compare as much as frameworks as possible in a large tabular form.

	Hybrid	Mobile
[**?**]	0	4
[**?**]	2	3
[**?**]*	1**	6

*is still under development
**a combination of Bootstrap of jQuery and Angular JS

Table 1: How many frameworks of which type are compared

As mentioned earlier, we will compare four HTML5 mobile frameworks based upon a large-scale POC. Our strategy tries to combine some elements that can be found in the literature. We derived five important criteria: community, productivity, usage, support and performance. A methodology to measure these criteria will be described below. A score will be assigned to each criteria and will be plotted in a spider graph [**?**]. The POC will drive the methodologies to test the criteria if possible. To be complete, a full detailed table will be provided where the significant difference of the frameworks will be presented.

Community

Social networks provide a good indication to determine the community that goes with the framework. [**?**; **?**] use Twitter followers, GitHub watchers and GitHub forkers as community indication. We would include Facebook likes also. The summation of these four numbers give the score for the framework community.

Productivity

We both implement the full POC in two different frameworks and carefully record the time to implement it. As we both start similar background, time differences can explain different learning curves. To double check this criteria, we also partially implement the POC in the other two frameworks. Just like [**?**], the partially implementation can be the POCs login screen. The summation times to implement the POC and login screen give the score for the frameworks productivity.

Usage

The POC will be subdivided in a number of implementation challenges. To score of this criteria will depend on the completion of the challenges by the framework. The assessment of the challenges can be found in table 2. The summation of all challenges score give the score for the frameworks usage.

Score	Assessment criteria
3	Supported by the framework
2	A plug in is needed
1	Custom implementation
0	Hack or not possible

Table 2: Assessment criteria for implementation challenges

Support

This criteria indicates the correct functionality of the POC implementation on different devices and browsers, similar as the cross platform capabilities of [**?**]. In each context, we will walk trough the scenario and rate the execution. If some parts of the scenario are infeasable, points will be subtracted. The summation of scores for every iteration of the scenario in each

context will give the score for the frameworks support.

Performance

Different aspect of the framework influence the performance. We will test the time to load and render the full POCs and isolated screens with Goolge Page Speed [?], as suggested by [?]. Also, isolated tests will record the loading time of dummy pages with 1000 buttons. The summation of render times will give the score for the frameworks performance.

3 Frameworks

3.1 jQuery Mobile

jQuery Mobile (jQM) is a mobile HTML5 user interface (UI) framework that was announced in 2010 [?]. In November 2011 version 1.0 was released [?] and one year later in October, version 1.2 was released [?]. As at the time of writing, jQM released version 1.3 [?]. The framework is controlled by the jQuery Project that also manages jQuery Core. The latter is a JavaScript library where jQM is dependent on [?]. jQM is among other things sponsored by Adobe, Nokia, BlackBerry and Mozilla [?].

Licence As of September 2012 it is only possible to use jQM under the Massachusetts Institute of Technology (MIT) licence [?]. This means that the code is released as open source and can also be used in proprietary projects [?].

Documentation One can find the documentation of jQM 1.2 on www.jquerymobile.com/demos/1.2.0. On the one hand it contains an overview of all possible UI components. By checking the source code, you can find out what code to write to get the same result. On the other hand it explains the API on how to configure defaults, use events, methods, utilities, data attributes and theme the framework [?].

Code and development jQM is a UI framework and thus provides mainly UI components. jQM provides six categories of components: pages and dialogs, toolbars, buttons, content formatting, form elements and litsviews [?]. One can obtain these components by writing HTML5 with jQM specific `data-*` attributes. When running the application, jQM will add the extra necessary code to correctly show these components by doing progressive enhancement.

There are three ways of writing a web application in jQM [?]. The first one is to write the full application that is composed of many screens, on one single web page. The advantage is that there are initially less requests to the server. The second option is to write a web page for each screen. The advantage here is that the first viewed screen is downloaded more quickly. However, with each transition, the next screen has to be fetched which can delay navigation. This is done with AJAX by default in jQM. Lastly, you can mix the two above to find an optimum by putting the most likely viewed screens on one web page and the less likely viewed on separated pages.

Browser support jQM divides browsers into three grades: A, B and C. An A graded browsers supports everything of the jQM framework, where a C graded browsers only provides basic HTML experience (so for example no CSS3 transitions) [?].

3.2 Sencha Touch

Sencha Touch (ST) is a framework developed by Sencha, a company founded in 2010 as a composition of Ext JS, jQuery Touch and Raphaël. Ext JS is a JavaScript framework for the development of web applications. jQuery Touch is a jQuery plug in for mobile development that adds touch events to jQuery and depends on the WebKit engine. Finally, Raphaël, is a JavaScript library for vector drawings. Pieces of the first two technologies can be found in the implementation of ST framework.

As at the time of writing, ST is at version 2.1.1 [?].

Documentation All documentation for ST 2.1.1 can be found at docs.sencha.com/touch/2-0. The most important features, are provided with code examples and an example of the code after rendered by the browser. The key concepts of ST are explained in extensive tutorials: some texts, some videos.

Another handy tool to discover the ST features is the 'Kitchen Sink' [?]. This is a web application, written in ST, that lines up all possibilities of the framework combined with the corresponding code.

Licenses ST is free within a commercial context in which the developer does not share the code with its users. There is also the option to use an open source version. This comes with a GNU GPL v3 license which implies a free code redistribution as most important property. More detailed licenses can be found at [?].

Code and development ST is written on top of Ext JS, and can also be considered as JavaScript framework. All code needs to be written in JavaScript and loaded by one HTML container. An other important aspect of ST is that is supports the Model-View-Controller (MVC) pattern. Models group fields to data-objects, views define how the content is presented to the user and controllers connect these based on events.

ST contains all UI-elements as JavaScript objects. Just like object-oriented programming, those objects are part of a class system. Classes can both be defined (`Ext.define`) or created (`Ext.create`). Single-inheritance and overriding is also possible.

To enhance performance, it is the programmers task to create components before they are used. In this manner, programmers can mimic asynchronous loading of pages by creating them in advance.

Browser support Just like jQuery Touch, ST is based upon the WebKit browser engine. This forms the major requirement for browser support. Although most mobile browsers contain this engine, some like FireFox Mobile and Opera Mobile lack behind [?]. Following [?], the next release of the Opera browser will contain this engine, a trend that most browser vendors will (have to) follow.

ST offers the programmer methods to ask for the current context where the end-user is working in. Properties like `Ext.env.Browser` and `Ext.env.OS` or methods like `Ext.Viewport.getOrientation` and `Ext.feature.has` can determine this context [?]. The latter has functionalities, just like Modernizer [?].

4 Evaluation

4.1 Community

4.2 Usage

Forms

*jQM (**X**):* Only placeholders were needed in the form elements, but no labels. The former can be achieved with the HTML5 `placeholder` attribute. The latter is mandatory in jQM for assistive technologies, but can be hidden via the `ui-hide-label` class [**?**]. The types of form elements used were `text`, `number` and `email`. The `date` type could not be used, because of the lack of support in mobile browsers [**?**]. Not only the lack of support, but also the need to specify a range the user can choose from, justifies the use of the Date & Time Picker of Mobiscroll [**?**]. The need for a custom date picker with only a month and year needed to be hardcoded.
*ST (**X**):* Most required form elements are provided by the framework and can be easily created. However, two problems became apparent. First, creating custom datepickers as needed in the POC is not supported. It is impossible to ignore the days field and only years can be delimited. Secondly, programmatically setting the value of a radiobutton is not obvious. A new fieldset has to be created on which the `setGroupValue` method needs to be applied.

Form filling

*jQM (**X**):* Loading data into the form for viewing purposes needed to be programmed manually because there is no automatic mapping between the data and the form. All the form elements on this form had to be set as read-only. This can be done easily for the types `input` and `textarea` with the `readonly` attribute. With `fieldset`, the `disabled` attribute was the solution. Lastly with `select`, the other options were deleted from the list.
*ST (**X**):* To show the list of clickable expenses, ST provides a navigation view. This mechanism allows to create the list of expenses together with a handler that gets triggers after an item selection. The handler fills an empty form with the expense record belonging to the list item and makes this the visible view. A back button is automatically created to return to the overview list.

Form validation

*jQM (**X**):* Because of the lack of support on mobile browsers for the `required` attribute [**?**], the plugin of Zaefferer [**?**] was used to check for required fields. This plugin also provides built-in validation methods that were needed: `number`, `email` and `date`. The dependency expression that was needed is also available in the plugin. When fields were not validated, a dialog is needed to show the errors. Normally the plugin shows these errors below the corresponding field, so customisation of the plugin was needed. To give the invalid fields a red border, CSS was used directly for `input` and `textarea`. Because of the progressive enhancement for `select` and `fieldset`, a bit of DOM traversal was needed.
*ST (**X**):*

AJAX, JSON & XML

*jQM (**X**):* AJAX requests are made via the jQuery library where jQM is dependent on. Handling plain data and JSON as response is straightforward and the data can be used directly. When using XML, the data has to be traversed like with HTML, this means by using selectors. Sending JSON can be achieved by converting the instance into a string object using the vanilla JavaScript function `JSON.stringify` and add it as data to the jQuery AJAX request.
*ST (**X**):* AJAX request can be done either explicitly via a direct `Ext.Ajax.request`-call or implicitly via stores. Stores really show the power of the framework by doing as much work as possible under the hood. They can be configured with a model to define the structure of the recorded objects. A proxy configures readers and writers that define where the data can be read or written. This can be locally at the client side or via a remote server. Readers and writers contain the format of the data - JSON or XML - and automatically parse this data to fields of the corresponding model.

Attaching evidence

*jQM (**X**):* A form element of type `file` is used to let the user attach the evidence. Depending on the mobile device, the options are to attach a local file or to take one with the camera. After selecting the correct evidence, it must be converted to base64 which is done by using the FileReaderAPI and a canvas. The evidence is read by the FileReaderAPI, converted to an image and imported on the canvas. The base64 encoding can be retrieved by calling the `.toDataURL()` on the latter and is stored in session storage. However the FileReaderAPI is not supported on Android 2.3 and lower or iOS less than 6.0 [**?**]. Another limitation is the file size that can be imported on a canvas on iOS devices which is depend on the RAM size [**?**]. The final limitation is the size of the encoded evidence in base64, which can exceeded the limit for session storage, that is depend on the mobile browser [**?**].
*ST (**X**):*

Signature

*jQM (**X**):* No functionality is given from jQM to draw a signature, so a plugin was searched. At first, Signature Pad [**?**] was used. Because of the time spend on changing the default layout, jSignature [**?**] was used instead. An advantage over the former was the automatic scaling to 100% width and the layout without the bells and whistles. However, the plugin does not work on Android 2.3 and lower [**?**].
*ST (**X**):* Drawing a signature with a pen or a finger is handled by a plugin [**?**]. Plugins can easily be added to the framework by placing the plugin file in the `ux` folder and loading it in the main JavaScript file. In most cases, these plugins are written by commited ST users and hosted publicly on GitHub.

Opening a PDF

*jQM (**X**):* The download the PDF, a hidden form is used because AJAX is not the preferred way for fetching raw data. When the users taps an expense form in the list, this hidden form is submitted with the correct parameters to download the PDF. The mobile device opens the corresponding native PDF viewer to show the PDF.

*ST (**X**):* Also, a plugin for a PDF-viewer can be found [**?**]. Many of these plugins can not be used as-is. I needed to adapt the PDF-viewer, for example, so I could fetch the PDF from the backend with a parameterized POST request.

Autocomplete

*jQM (**X**):* As of version 1.3, jQM has built-in autocomplete support [**?**]. Because the POC used 1.2, the plugin of Matthews [**?**] was used. To show only five suggestion, a little customisation by the `slice` function was needed because the plugin did not provided this.

*ST (**X**):*

Splitscreen & device-specific lay-out

*jQM (**X**):* No functionality is provided by the framework, so a search for plugins was started with [**?**] which led to: Splitview [**?**], SimpleSplitView [**?**] and Multiview [**?**]. All these had shortcomings. The first changed the jQM library code, the second uses jQM 1.0.1 and the last had problems with adapting to different browser sizes. [**?**] showed the use of CSS3 media queries to accomplish the split screen without a plugin. The documentation of jQM 1.2 also used a similar approach [**?**]. This approach is also encouraged in the documentation of jQM 1.3 [**?**]. The smartphone menu can be accessed on both tablet and smartphone devices when clicking on the sub header.

*ST (**X**):* The main screen of the POC requires a splitview in tablet mode. A `vbox` layout splits the viewport with a vertical axis. The `flex` property defines the ratio of the sizes of both resulting components. The right component is the page that changes by clicking the menu items in the left component. This can be realized via a `card` layout where changing the page implies setting the active component of the `card` layout

Offline

*jQM (**X**):*

*ST (**X**):*

5 Future work

6 Conclusion

A Comparison table

Criteria	jQuery Mobile	Sencha Touch
Community		
Twitter followers	5	1200
GitHub watchers	7	23
GitHub Forkers	9	225
Facebook likes	3	355
TOTAL	24	1803
Productivity		
Toggle 1	XYZ	XYZ
Toggle 2	XYZ	XYZ
Toggle 3	XYZ	XYZ
TOTAL		
Usage		
Challange1	XYZ	XYZ
Challange1	XYZ	XYZ
Challange1	XYZ	XYZ
TOTAL		
Support		
Scenario 1	XYZ	XYZ
Scenario 2	XYZ	XYZ
Scenario 3	XYZ	XYZ
TOTAL		
Performance		
T1	XYZ	XYZ
T2	XYZ	XYZ
T3	XYZ	XYZ
TOTAL		