# Creating Data Verity's Mobile Front End

Allison Crouch
Carter Mann
Nick Wunder

Data Verity

June 17, 2014

## 1.0 Introduction

Data Verity provides business intelligence solutions, primarily to banking institutions, and is looking to expand into other markets. Data Verity develops and supports a data management tool that collects client generated information and performs sophisticated statistical analysis to equip their client's policy makers with the best possible information about their business. Currently, Data Verity has a proven desktop application capable of providing dynamic tables, charting, and report generation. In order to grow within the banking industry, and expand into other markets, it is critical that Data Verity develops a mobile application to attract regional and national clients.

Our main goal is to take the existing desktop application and put it on a mobile device. The mobile front end will have a different look and feel from the desktop site but will incorporate all existing functionality. Once this mobile application is commissioned, clients will have the option to take their business away from their desks and out to the field.

The target for this new mobile application is doctors and their patients. Specifically, the application should be a tool to guide pregnant patients through their pregnancy and facilitate communication with their doctors, i.e. informing the doctor about the health of the patient through surveys, medication schedules, and reminders about upcoming appointments. The application should notify mothers about the size and development of their fetus every day, along with providing helpful information about exercises and diets. It should provide the ability for patients to document their experiences, such as side effects of medication, and produce reports for both the doctor and the patient based on the information provided. Finally, the application should enable the user to create a time lapse of photos taken throughout the pregnancy into a movie of the baby's development.

## 2.0 Requirements

Functional Requirements:

- The mobile application needs to be able to create multimedia content to deliver to users.
- Communication: Doctors need to be able to communicate with individual patients.
- Notifications: Patients will need to receive notifications based on custom timelines.
- Document: Patients will need a way to document experiences.
- Reporting: The mobile application needs to produce reports for patients and doctors, informing them about frequency of behaviors, appointments, and previous illnesses.
- Time-lapse Photos: A photo interface that will put together a time-lapse of photos into a movie, where the application has a semi-transparent version of the last picture over the picture viewer so that you can match the view as closely as possible.
- Authorization: Patients will need to be able to authorize doctors to contact them, and doctors will need to be able to approve the patient.

Non-functional Requirements:

- Coding must be done in javascript.
- Packaging of the application must be done using Sencha.
- Apache Cordova must be used as the application programming interface (API) for the camera.
- The application must be published for iOS and Android.

Project Risks:

- Team members have limited experience with application development, Sencha, and Apache Cordova.
- One team member has limited experience with Linux.
- Potential difficulty translating functionality across platforms.

**3.0 System Architecture**

Data Verity uses a typical web application architecture in a unique way (Figure 3.0.1). Client and server side communication is handled through asynchronous JavaScript and XML (AJAX) requests. Typically an AJAX request asks for a specific set of data. Enterprise Service Platform (ESP) extends this typical response by requesting additional user view configuration information. For example, each ESP table has a unique set of actions. In a typical implementation each action would be coded along with each button on client side JavaScript. Instead, ESP serves the button configuration to the client, decoupling the behavior of the button from the user interface implementation.

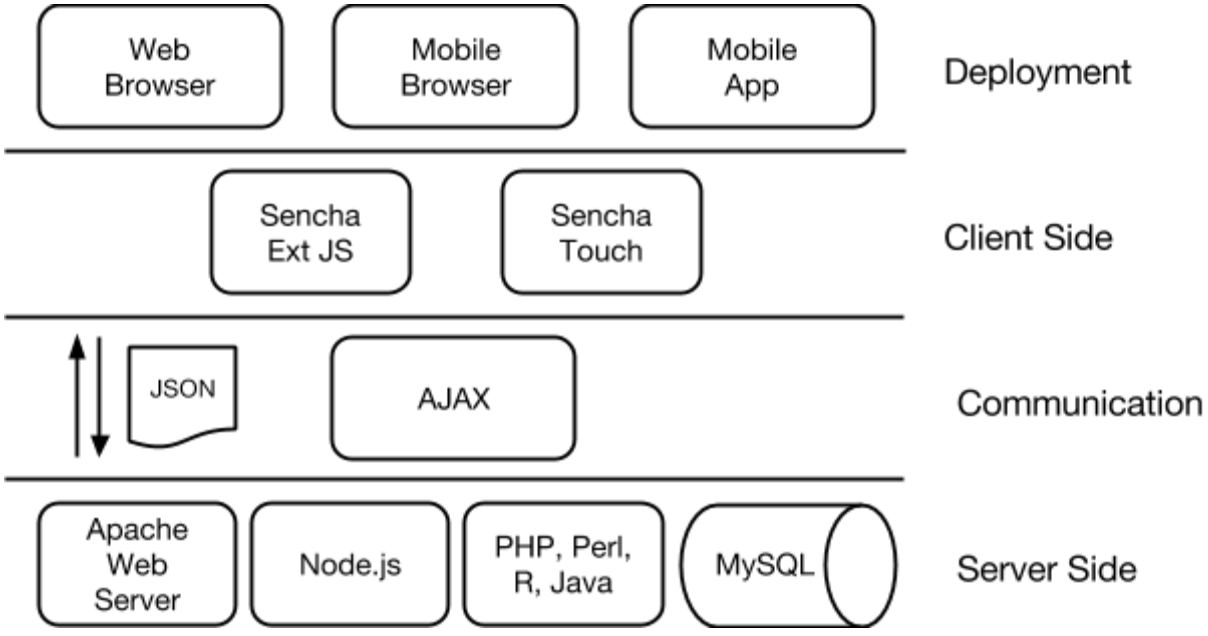


**Figure 3.0.1** ESP System Architecture.

ESP leverages Sencha Touch, a HTML5 framework. Since the Sencha Touch core is written in JavaScript a single code base supports deploying native mobile applications to multiple platforms, e.g. iOS, Android, Windows. Sencha Touch is build on Sencha Ext JS 4—the main engine behind both mobile and desktop applications. The client side layer is responsible for rendering user views, and acts as an interface to the web server. User actions generate a JavaScript Object Notation (JSON) file which ESP sends to the web server. The JSON file is parsed in a PHP controller, data is collected, and a JSON response is served back to ESP. Finally the response is parsed and the user's view is updated with the requested data.

## 4.0 Technical Design

### 4.1 Mobile Design Challenges

The main challenge with mobile design is the limited screen real estate. Desktop applications have more options to display information. ESP uses three views to convey information; desktop view, tab view, and home view. The desktop view acts similarly to a standard operating system desktop, displaying the various tools and widgets available. The home view is a customizable home screen, openable from the desktop view, which contains specific user information. The tab view is used within the home view to separate charts and tables into defined groups. The charts and tables are all rendered immediately in widgets, usually contained inside a tab, so the user can easily compare different data sets and edit each of them separately.

This approach is not something we could accomplish with the limited area available on a mobile device. Instead, we chose to create a new portal view for the mobile client. When the user logs in to the application, we load the data from the server for all of the tables and charts similar to the desktop client. Unlike the desktop application, we do not render any of them on startup. Instead, we created a dashboard which contains objects called portals. The portals contain objects called portlets that are associated with them. These portlets are displayed as a menu when the portal has been selected from the top bar of the dashboard. The portlets within each portal serve as buttons that, when pressed, will access one of the charts or tables that have been retrieved from the server. When the user selects a portlet from the menu for viewing, the chart or table of data is rendered on screen, stacked on top of the dashboard. This prevents excessive clutter on the dashboard, which can serves as a main menu for the application.

On the desktop version, there is room to put features in different places. Using a similar design on a mobile site would clutter the limited screen space. Instead, we created a single menu button that, when selected, folds out into a list of all the actions that are available for the chart or table the user is currently viewing. Using this menu also allowed us to leave some of the table visible, which allows the user to change their selections with the actions still available as a sidebar.

Another challenge we faced with the workflow was stacking the open windows. When the user has a table or chart open, they usually have the option to open several other windows from that popup. If they choose to do so, the new window will appear in place of the previous one. After finishing their

business in the new window, the user would expect to return to what they were originally looking at upon closing the current window. By default, the application did not operate this way, instead automatically rendering the dashboard whenever any window was closed. To have the application perform in line with what a user would expect, we had the application generate new windows with unique ids. This allows the program to stack newly rendered windows on top of previous ones without losing the previous object. Later, when the new object is closed, the application automatically looks for and opens the most recently created window in place of the dashboard. As a secondary benefit, this allowed us to prevent duplicate windows of the same item from being created by checking the new window's id against all existing ones and terminating any duplicates when the new window is rendered. Using this system, the user cannot create multiple instances of the same window, preventing any potential exploits or memory issues.

### 4.2 Asynchronous JavaScript

Since the majority of the client side code base is JavaScript, it is important to understand the lifecycle of a typical JavaScript file. All JavaScript files are compiled client side with a runtime engine, e.g. Google's V8 engine, Mozilla's SpiderMonkey and Rhino, and Apple's Nitro. Most engines run on a single thread. Consider a script running on a single thread. In a request response model, after a request is sent the thread blocks until it is signaled when it received a response (Figure 4.2.1). Unlike other languages JavaScript engines evaluate code in the event driven model (Figure 4.2.2). Furthermore, JavaScript functions are not preempted, instead the script continues execution after a request is sent. Sometime later when the blocking operation signals, the response is evaluated.

It is easy to create race conditions when using non-blocking operations. We can control the execution of asynchronous functions using the JavaScript callback message queue and event loop (Figure 4.2.3). When execution starts functions are placed on the stack. As asynchronous functions are encountered, a callback and message is registered in the current session. Sometime later, the event associated with that message is triggered and the message is placed in a message queue.

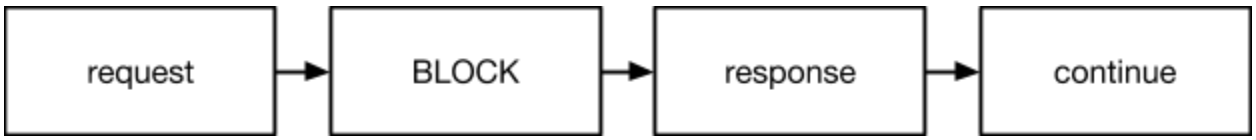


**Figure 4.2.1** The request response model.

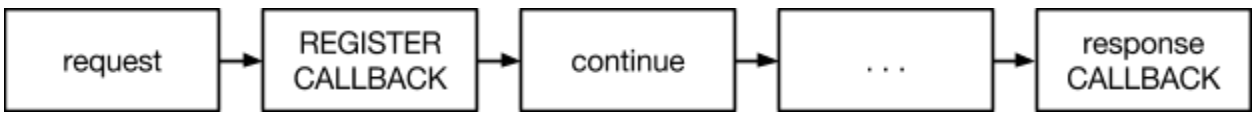


**Figure 4.2.2** The event driven model.

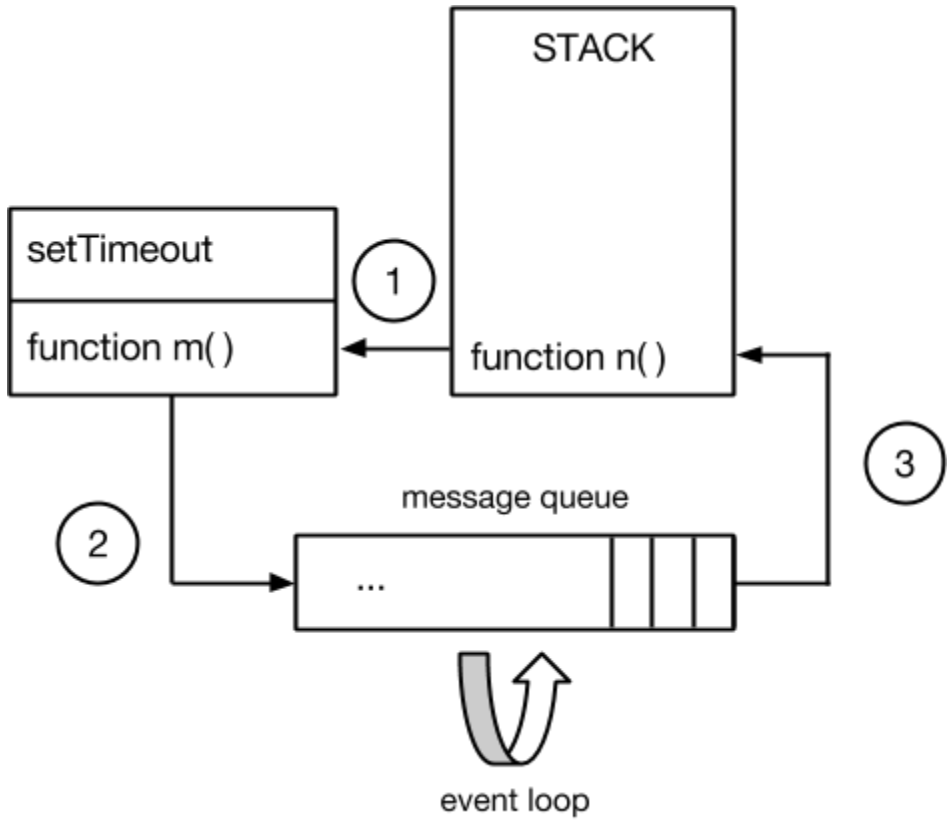


**Figure 4.2.3** JavaScript callback lifecycle.

**5.0 Design and Implementation Decisions**

A big part of our implementation lies in languages, frameworks, and tools. JavaScript, HTML5, Sencha Ext JS, and Sencha Touch are the best solutions to our design challenges. We rely on Ripple(beta) for in browser user interface testing and native devices for hardware API integration.

JavaScript is our primary programming language for the project. Our client is currently using JavaScript for their existing client side code base; staying with the same language was the most natural choice for porting the functionality from their existing desktop system to mobile platforms. An important feature JavaScript provides is support for today's common mobile devices operating systems—iOS, Android, and Windows.

HTML5 is necessary for communication with the client's data servers, to request the data that the frameworks would be rendering. It would be used occasionally alongside the javascript, usually to give the code a definite path to use and for the innerHTML of some applications.

Sencha Ext JS is a JavaScript framework for building web applications with an extensive AJAX API for client-server communication. This framework is used by our client to build a desktop version of the application.

It makes sense to build a mobile application using the same framework. Sencha Touch extends Ext JS with mobile functionality. The Touch framework is the foundation for this project, allowing us to maintain as many similarities as possible between the two versions of the site.

Ripple is an extension for Google Chrome that acts as an emulator for mobile devices within the web browser. We used the Ripple Beta tool to emulate the appearance of our application on mobile devices. Ripple was useful as it allowed us to retain access to the Chrome Developer tools while giving the appearance of a mobile environment. This allowed us to view the relative size and readability of our application and iterate accordingly without the hassle of running multiple devices.

Apache Cordova is a set of device APIs that allow mobile application developers to access device functionality such as the camera or accelerometer from JavaScript. This allows us to use the camera in our application without having to use any native code for the device it is running on. Combined with a Sencha Touch UI framework, Cordova allows the application to be developed with just HTML5, CSS, and JavaScript, which means that the application can run on any device without any necessary modifications to the native code.

There were, of course, implementation issues throughout the project. One of the most obvious ones were the difficulties in porting Sencha functionality from the desktop version to the mobile client. For starters, there is a difference in how Sencha for the desktop works and how Sencha for the mobile works, so porting the code in requires that the code be read over and rewritten for parts where the Sencha differs. There was also a problem in that we cannot directly port functionality from the desktop to the mobile, because the mobile does not have the room on its screen to show everything the desktop does comfortably.

We encountered some issues with the emulators used for developing the mobile client. One persistent issue was that the android emulator was very slow, which meant that there were issues with the server timing out our requests despite nothing having gone wrong.

Outside of the mentioned difficulties, there are systemic issues with Sencha that we became aware of as we worked our way more deeply into the code. An immediate problem we came across was that, if one platform wanted both sides to extend a feature, both bases had to design it themselves. The code was just different enough that the mobile could not extend the desktop version and vice versa.

Similarly, in order to implement new features on an already existing class, changes needed to be added to both the mobile and desktop source. The differences in the code, beside the implementation of buttons and moving them to a menu, lied in the fact that function names and function variables have different meanings from Sencha Touch to Sencha Ext or have the same names but different meanings.

## 6.0 Results

### 6.1 The Mobile Front-End

We successfully implemented the client's existing system on the mobile front-end. Furthermore, we refactored the code that is used on the mobile platform, and implemented all buttons provided by the server.

Sencha Touch is a valuable tool for the project, as it allows us to write the classes and functions necessary for the project in a similar format as their desktop counterparts written in Sencha Ext JS. However, the Sencha Touch framework is an expansive toolset that takes time and dedication to learn, which slowed the project's progress significantly early on. Exacerbating the problem, the documentation for the framework was oftentimes vague on what a function, event, or method actually did, sometimes to the point of causing additional confusion. Additionally, many of the documentation examples given are the simplest solutions possible, making the documentation unnecessary as we tried to write more involved code that used those same functions in a way that few of the examples touched on.

### 6.2 Lessons Learned

It is difficult to know exactly how long stories will take. We had three major keystones with, what seemed like, more than enough hours to accomplish them. The fact is, time estimates are hard. We expected to create a mobile-front end with in two weeks. Then move on to develop a mobile medical application using our newly created mobile platform, wrapping up by shipping the product on the Apple Store. We eventually discovered that each keystone required more time, and more effort than our estimates.

In reality, the first keystone involved getting though the JavaScript learning curve then the steeper learning curve of the Sencha framework. Getting familiar with both JavaScript and Sencha were prerequisites to building a working mobile-front end. Knowledge of JavaScript needed to extend beyond just learning the language; we needed to understand how it works as well. This proved to be an ongoing learning experience. As we encountered a new problem with our design, the solution was, more often than not, solved by learning about a feature in JavaScript.

Understanding JavaScript closure scope are vital in sound mobile application design. JavaScript closures are difficult to understand which adds to the challenge of debugging. Fortunately Google Chrome Developer Tools call stack feature is capable of changing scope during debugging; we were able to quickly determine what *this* was with little effort after learning more about the Developer Tools.

Finally, testing mobile applications without a native device is burdensome. The Android emulators are slow and unresponsive. We decided after several days of testing to focus development for an iOS device. We did not anticipate the leed time in setting up Apple products. Beyond establishing a the development environment, the Apple Developer portal required approval directly from the source—Apple.

### 6.3 Future Work

All of our future work that can be done or planned involve outside code or machines; for example, one of the the items was creating an iframe support that would make viewing images much easier. This would help in improving the user interface, and would support the photos for the image overlay. Another item was for making an image overlay, so that the patient could see changes over the progression of time, which was one of the requirements we were unable to begin working on. And the last one was implementing both local and push notifications, so that the patient could receive reminders and information from their doctor. This was also a part of our requirements that we were unable to get to. Adding these features would create a full-fledged mobile application that can be tailored to any industry.