Fox on the Run II

Eric Selje
1308 Spring St.
Madison WI 53715
Voice: 608-213-9567
Twitter: EricSelje
Eric.Selje@gmail.com

In our last episode, we made data from a Visual FoxPro database accessible via a native mobile application. It was quite a journey, but in the end we had a native Android application that pulled our FoxPro data down via a web service and displayed it to the users. A truly useful mobile line of business application will need to do more though. We need to make the data editable, of course, but more importantly we need to be able to work offline and synchronize data from all of our devices back to our master database and with each other. We'll explore how to do that in this whitepaper.

Part 1: Quick Review of Fox on the Run

Last year we started by making the case that mobile applications are surging in demand, and I believe we've seen that prove itself in the last year. We then dared to try the improbable: Take a Visual FoxPro database (in fact, the very database that the Southwest Fox organizers use for their conference) and create a mobile application that displays that data. In order to bring that in within the constraints of the session, we picked one particular set of technologies to accomplish that and our final application left us with only read-only data, but we did succeed in doing this.

Let's start by reviewing how we accomplished this. This is merely a quick overview, however, so please refer to last year's whitepaper for more detail.

We started with a subset of the Southwest Fox database whose schema looked like **Figure 1**:

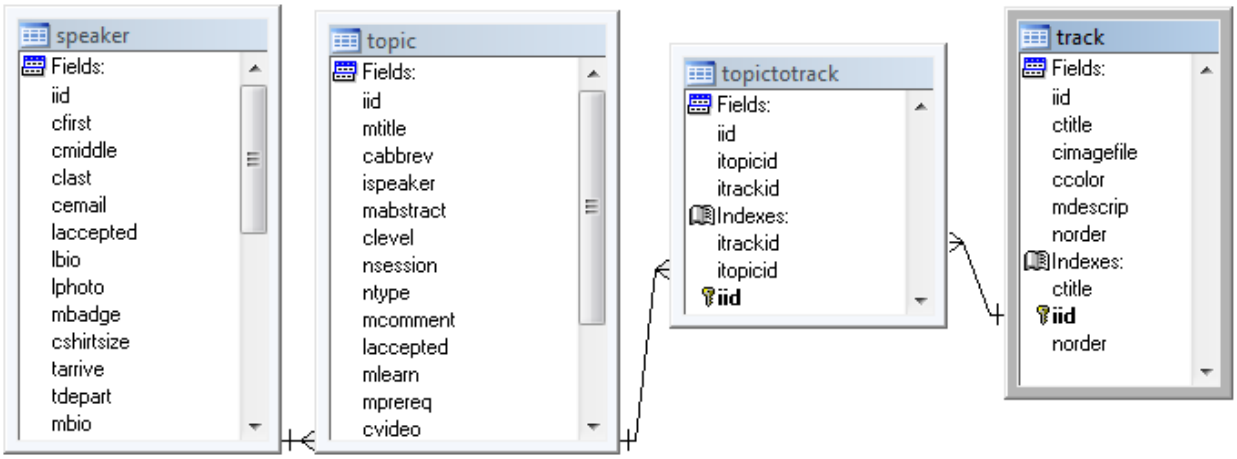

Figure 1: Our sample database schema

We then created some quick business object wrappers around each of our main tables, Speaker, Topic, and Track. We used West Wind's business object class, wwBusiness, to do this because we really like coding as much as possible in FoxPro and we really don't like learning entirely new paradigms, such as Microsoft's Web Services or their latest flavor of the month, WebAPI.

We added a new *getList()* method to our base business object class to return all the rows of the associated table in JSON (JavaScript Object Notation). This was easily accomplished with West Wind's wwJSONSerializer class.

After configuring IIS and firing up our West Wind application, we tested it and got our data back (see **Figure 2**)!

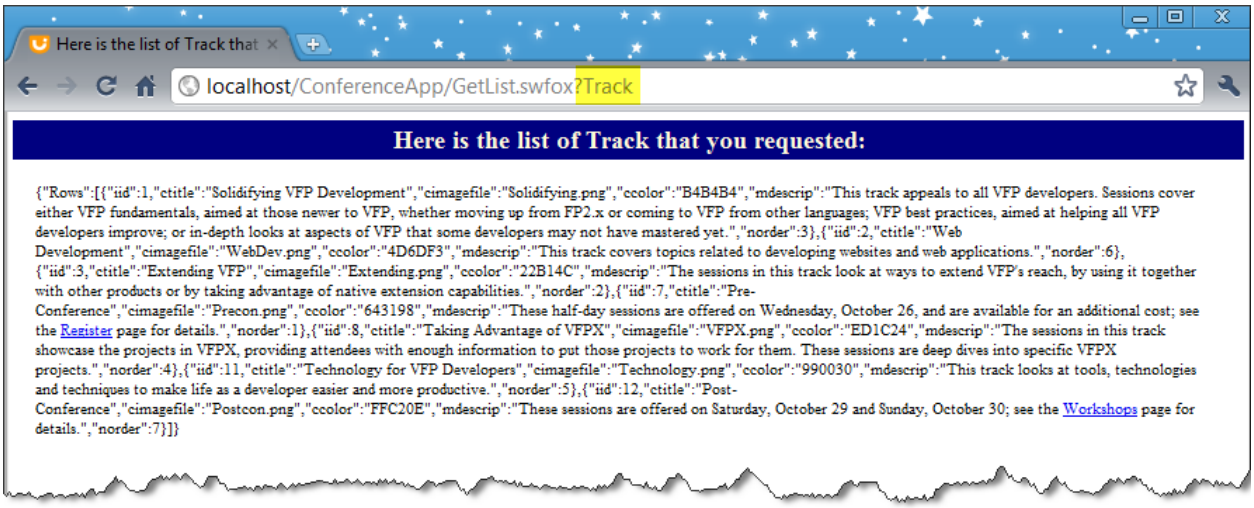

Figure 2: Our VFP Data returned as JSON to the web browser

Our next step was to create an actual HTML5/CSS/JavaScript web application that turns that raw data into something usable for our users (**figure 3**). We got a little boost from jQuery Mobile to give it some nice styling.

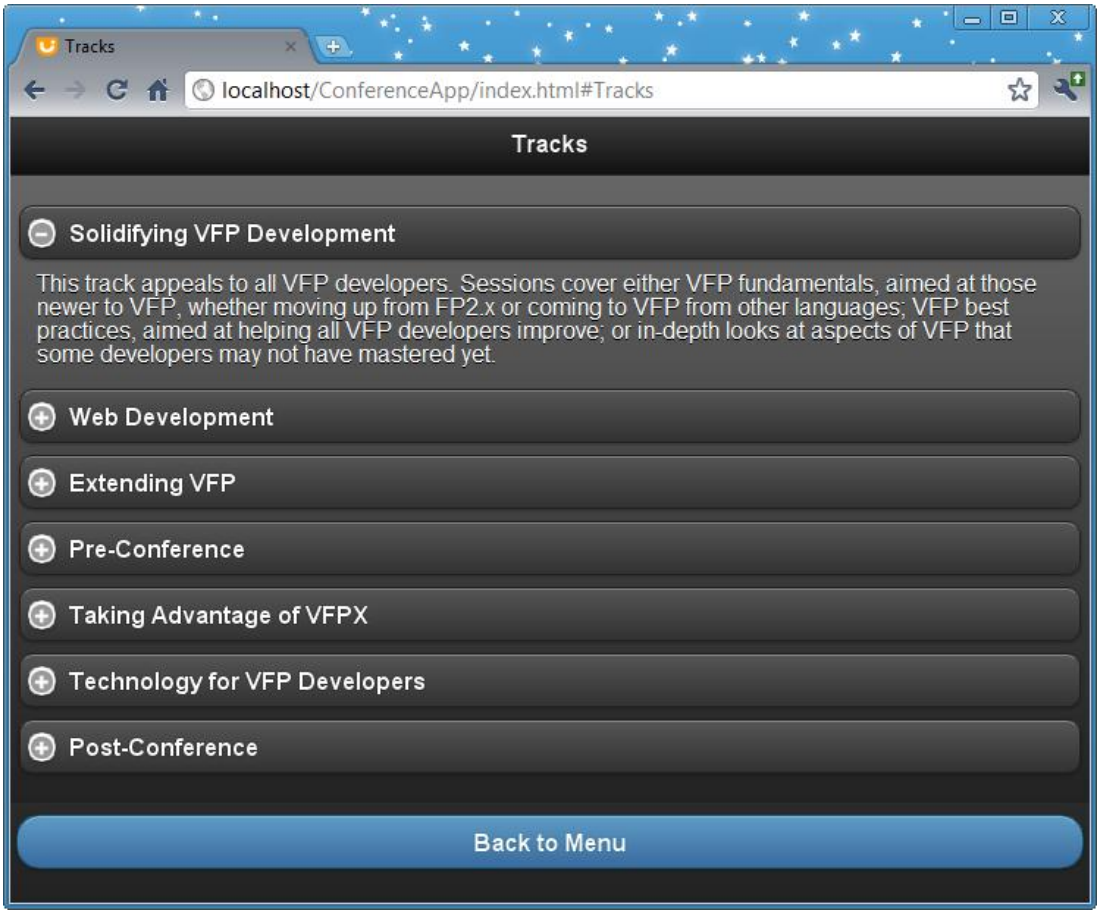

Figure 3: The same data as Figure 2, but spiffed up with jQuery Mobile

At that point we had a perfectly usable web application that actually looked good on mobile devices as well. But we also wanted to take advantage of features of the mobile device that HTML5 alone couldn't do, so we imported our HTML5 site into PhoneGap, added a few native features like muting the ringer and vibrating the phone, and compiled our application into a native Android .APK file for distribution.

We could have also chosen to compile it to a native iPhone application if we had been using a Mac. **Figure 4** shows our app running in the iPhone emulator.

Whew, that was a wild ride!

Figure 4: VFP Data on an iPhone!

Part 2: Running On

As good as that was, what we were left with was an application that merely allowed us to see our Visual FoxPro data. Real applications need to be able to also collect data from its users while they're mobile. That's what we'll focus on in this whitepaper.

Our main concern when writing business applications for mobile devices (smartphones, tablets, laptops, etc.) is that our device may not always be "online". Believe it or not there are still many places that aren't covered by a cell phone signal, let alone the latest and greatest 4G LTE. This means three things:

1. We need to ensure our application loads on the device every time, regardless of whether we're actually able to reach a particular URL. We'll look at two ways to make sure this can happen.

2. We need to ensure we have data cached locally rather than reaching out to the web service and loading the data when it's needed.

3. We need to ensure any changes to the data that are made locally make their way up to the master database, and vice versa. This needs to happen reliably with proper conflict resolution and without using too much of our precious bandwidth.

These considerations add a fair amount of complexity to our application, but we've come this far so let's keep running on…

Is the Application available?

Our first concern is whether we can load our mobile application at all, as nothing else matters if we cannot even get into the application.

Native Apps

The easiest way to ensure this cannot fail is to completely install it on the mobile device itself. With a PC, this would mean running an installation routine that puts everything the

applications needs on the hard drive. With a mobile device we can also create and distribute a “native application”.

We did this last year by starting with an HTML5/CSS/JavaScript application and importing it into a PhoneGap project, which allowed us to compile the web application into a native application that looked pretty good even though we didn’t write one line of Java or Objective-C. This solution even gave us the ability to take advantage of native features of the device such as alerting and vibrating as well as native sensors such as GPS and the camera.

The landscape has changed a bit in the subsequent year, with PhoneGap now acquired by Adobe and the open-source version is now called Cordova, but it’s still a viable solution. Titanium Appcelerator is also quite compelling, and there are a few other products on the market that also look good, with more and more coming all the time.

Creating a native application does have its disadvantages. Distributing custom applications over the sanctioned app stores is not trivial, and you cannot easily enforce that your users download the most recent version of your application.

So for our let’s assume we’re not going to compile a native application, but rather we’re going to create a straight HTML/CSS/JavaScript application. This way all we have to do to keep our users up-to-date is deploy a new update onto our web server and the users will get it when they request our URL.

But how will they request our URL if they’re not online?

Manifest Density – Caching Your Application

Browsers have had the ability to cache pages locally for a while, but this feature wasn’t standard across browsers and was outside the control of developers until HTML5 introduced *Application Caching*.

With Application Caching you get three advantages[i]

1. Offline browsing - users can navigate your full site when they're offline
2. Speed - cached resources are local, and therefore load faster.
3. Reduced server load - the browser will only download resources from the server that have changed.

To implement application caching, all we have to do is decorate each page’s <html> tag with an extra attribute that points to the name of the manifest file. E.g.

```
<html manifest=”swfoxapp.manifest”>…</html>
```

Note: If you use a “master page” that contains the only <html> tag in your entire application, so you only have to add this in one place.

Much like a shipping manifest, an application manifest file simply enumerates every file that goes along with the application stores them locally the first time we connect to our web page. If we try to hit the same web page after that and our application is offline, the browser will use the stored files rather than throw a 404 error.

Our manifest file for our Southwest Fox application would look like **Figure 5**.

```
CACHE MANIFEST
index.html
web.config
css/html5reset.css
css/jquery.mobile-1.0rc1.css
css/style.css
js/jquery.mobile-
1.0rc1.min.js
js/jquery-1.5.1.min.js
js/modernizr-1.7.min.js
```

Figure 5: swfoxapp.manifest

If you're thinking it might be difficult to remember to maintain your application manifest after each change you make, I agree. This reveals an advantage of the new trend towards "Single-Page Applications" (SPAs): There are fewer files to maintain. Fortunately modern development environments (including Visual Studio 2012) can automatically generate the manifest file for you during the build process.

You can do a couple of pretty neat things your manifest file, such as specify which resources are inaccessible unless you're online, or provide alternative fallback options for when a resource is offline. This is handy when you're application requires you to be online in order to authenticate the user, for example. By adding the following line to your application manifest, you're telling the browser it *must* be online in order to access the login page:

```
NETWORK:
login.html
```

Any time the manifest file changes, the new files will be redownloaded from the server, otherwise it remains cached on the local device as is. With all of the files in the application cached, you can ensure your app is available regardless of whether you're online. If you want to force a download of a new version of your application, simply make any small change to the .manifest file, like a version #, comment, or timestamp and that will force the browser to pull down anything new.

Note: If your app is destined to be compiled into a native app with PhoneGap, Appcelerator, or a similar tool, application becomes a moot issue. At compile time, all assets for the application are gathered and linked into the application.

Also: Even if your web application is not meant to run offline ever, you should take advantage of Application Caching to improve its performance. If the manifest hasn't changed, the browser will be relieved of the responsibility of checking each file you request and simply use the cached version, greatly improving your app's response time.

With application caching in place we are now ensured our application will load even if we're offline, but now we need to deal with getting our *data* into our application when we don't have access to our web services.

Is the Data available?

Network Status Detection

It would be ideal if there was a quick and reliable way to ask the browser if we're online or not. While there is a ***navigator.online*** property which would appear to be just the panacea we need, the current state of browser support for this property is a mess. Some of this comes from the different interpretations of what it means to be "offline." Does it mean we've switch our browser into offline manually via the "Work Offline" option? Does it mean our cable was disconnected or we lost our wireless connection? What if the cable is intact but our router went down? For now, *navigator.online* is an unreliable indicator of whether we're offline.

Some browsers implement the ***document.ononline ()*** and **document.onoffline()** events, but that isn't consistent either. If you can enforce a specific browser for your application, trapping these events might be a suitable solution except they don't actually tell you that your web service is reachable. What happens if connectivity is actually fine right up to your server, but your server is down?

The most reliable method is to write a simple Ping web service for your application, and if that web service fails then you know you're offline. You might code this like **Figure 6**[ii]: In JavaScript, the setInterval() method is like a FoxPro timer(), so this is essentially saying "Hit the default web service test page for our app every 30 seconds, see if we're online, and set a flag I can check. "

```
 <script type="text/javascript">
        var offline = function () {
            status = false;
            display = function () {
                alert("Offline Status is " + this.status)
            };
        };

        // Implement global ajax error function.
        $(document).ajaxError(
function (e, xhr, settings, exception) {
            offline.status = true;
            alert("Offline? " + offline.status + "\nError in: " +
 settings.url + "\nError: " + exception);
        });

        updatePendingInterval = setInterval("testOffline()", 30000);
        testOffline();

        function testOffline() {
            $.post(<Our web server's test page>,
function () { offline.status = false; offline.display })
                .error(function () { offline.status = true; offline.display });
            clearInterval(testOffline);
        }
    </script>
```

Figure 6: Detecting Online Status

Now that we have a fairly reliable way of detecting our online status, what should our strategy be for getting at our web service? Perhaps something like this, where we get the data from the web service like before as long as we're online, but if we're offline we'll use a cached version?

```
If (offline.status = false)
  useDataFromWebService();
else
  useDataFromLocalDataCache();
```

The problem with this approach is that at some point we'll have to synchronize any changes we make in our local cache back to the main database. How would that happen? The answer once again is "It depends." It depends on the following factors:

- What device are we using for our application? Is it so generic that we don't have any say in the browser our users will be using, or can we specify one?

- What method are we using to store the data locally? HTML5 includes various local storage standards to choose from, we could go with something like SQLite, or go completely off the menu with something like CouchDb.

- How are we handling things on the server side? If we're sticking with our own hand-written web services, we'll have to code our synchronization methods manually. What does that entail? What other choices are there?

Let's walk through a few of our options:

Device and Local Storage Options

If our mobile application were intended to only run on a full-fledged Windows laptop, life would be fairly easy because we VFP developers could easily write our VFP apps and have the data stored locally in either VFP or perhaps SQL Server Compact or Express. There would be some considerations on how to synchronize the data of course, which we'll get to in a bit.

However these days tablets and smartphones are the thing and HTML5 is the way to write applications that will work across all of these devices. HTML5 has myriad ways of storing data locally:

Cookies

Yes, good old reliable "cookies" are still an option for storing data locally. However cookies aren't a great choice for caching much beyond the most basic authentication information because web sites are limited to 20 cookies per domain and 4k per cookie, so you'd be striving to come up with a scheme to store your site's data within those limits. Cookies can

also expire, be blocked, or even wiped out by the user thus making it even more unsuitable for this task.

Web Storage

Web storage takes cookies to the next level by lifting the storage limit to at least 5 MB per domain. Much like cookies, the information is stored in Name=>Value Pairs although we can specify that we want the data kept forever or just as long as our session is running:

```
localStorage.setItem('myKey', 'myValue');
var myVar = localStorage.getItem('myKey');

sessionStorage.setItem('myKey', 'myValue');
var myVar = sessionStorage.getItem('myKey');
```

Web storage can only store strings, which may seem like a limitation at first but really isn't too bad because entire objects can be serialized into JSON (JavaScript Object Notation) strings, and images can be encoded in Base-64. The biggest drawback to web storage is that the data isn't inherently secure while it sits on the disk, so if someone found your device they could theoretically see your application's private information. I'm sure you've read about that sort of thing happening in the media.

WebSQL

The next step up from Web Storage is WebSQL, which is a variation of SQLite. WebSQL brings the syntax and terminology that is very familiar to us database developers: tables, indexes, SQL, transactions, etc. Here's a snippet of how you might store the rating of a session you saw at Southwest Fox into a WebSQL table that may or may not even exist yet on your local device:

```
var db = openDatabase('swfox', '1.0', 'Southwest Fox', 2 * 1024 * 1024);
db.transaction(function (tx) {
  tx.executeSql('CREATE TABLE IF NOT EXISTS evals (id unique, sessionId int,
rating I)');
tx.executeSql('INSERT INTO evals (sessionId, rating) VALUES (1, 5)');
});
```

With its natural feel and expanded storage capacity, WebSQL would seem a natural choice for us to use and it would be except for one major drawback: it's not supported by Firefox and, less tragically, Internet Explorer. It's also been deprecated by the W3C standards committee, but that shouldn't deter Visual FoxPro programmers.

If you have the ability to direct which browser your users will access your web application with, then WebSQL may be a perfectly suitable solution. And there are a few open-source client –side synchronization libraries that work with SQLite, such as WebSQLSync, OpenMobster, and Persistence.js so you don't have to write the code completely yourself. We'll discuss synching more in a bit.

IndexedDb

Mozilla's alternative to WebSQL is called IndexedDb. IndexedDb is more like Web Storage with its key/value pairs, but it's built for high performance. It's not relational though, so aggregating the data into an object for local storage will take a bit of coding, as will synchronizing the data back to our relational master database. It's also unsupported by most mobile browsers including iOS Safari and Android's browser, so this probably isn't your best choice.

Storage Wrappers

Wouldn't it be nice if there was "one HTML local storage technology to rule them all?" In a way there is, if you go with a *storage wrapper*. These JavaScript libraries encapsulate the best local storage option available to your device so you don't have to worry about the details. Examples of these types of libraries are:

- Amplify.js
- Lawnchair.js
- Store.js
- JayData.js

I've only recently come across JayData while I was researching the paper and haven't had an opportunity to implement it yet, but it looks very promising. It's especially appealing because it has built-in synchronization facilities.

CouchDb

I want to give a special mention to CouchDb. It's not an HTML web storage technology, nor is it a relational database (it's NoSQL). It also requires a service to be running on the local device unless you use the PouchDb (Portable Couch) JavaScript implementation, but this actually makes it perfect for native device implementations. It's remarkable because this database was built from the ground up to solve the exact issue that this paper addresses: offline applications. CouchDb's premier feature is built-in synchronization to a CouchDb server.

What's interesting to me from a technical point of view is how similar CouchDb's synchronization technique is to a familiar problem that we developers constantly contend with: source code version control. What CouchDb is doing behind the scenes when it synchronizes is getting any new data from the server, comparing it with the version it already has, automatically merging any changes it can (based on rules you set up), and then sending your local changes back to the server. Does that sound familiar? Because CouchDb uses a "NoSQL" data scheme rather than a relational one, entire maps of data are represented in one JSON-like document rather than collections of tables, rows, and columns. This makes the comparison and merging of documents (aka records) much easier.

If I were beginning a brand new web application that needed offline capability I would give CouchDb serious consideration.

Server Side Options

The other side of the equation when solving your data synchronization equation is what is running on the server side.

SQL Server

SQL Server comes built-in with technology called "Merge/Replication" that allows you to partition parts of your database into subsets and synchronize that data to other instances of SQL Server (including Express editions). This is ideal if you're running variations of SQL Server on both the client and server sides.

If you're using Visual FoxPro on the server side you might consider writing a SQL Server Integration Services package to synchronize the Merge/Replicated SQL Server data with Visual FoxPro data, or use Sybase's SQL Anywhere wrappers around your DBFs to handle the synching.

A brand new alternative is Azure Mobile Data Services, part of the Mobile Services SDK. As of this writing though this option is only available for Windows 8 RT apps, although Microsoft has promised to expand that in the future and may have already done so by the time you're reading this.

CouchDb

CouchDb should be mentioned once again because it's also a server side database and has synchronization built-in. If your data is already in SQL Server or Visual FoxPro, you would have to once again come up with some way to synchronize that data between the CouchDb server and your master data, but at least now you've moved that synchronization issue up to the servers instead of between the client and the server, so it may be a little easier to implement.

Web Services

The term "web services" in the sense that I'm using it here means a server set up to receive requests from the client, process them, and send back the results. This is what we did last year, using IIS and West Wind Web Connection wrapped around our Visual FoxPro tables and this works great if you don't mind writing a little code.

We asked earlier about this algorithm for working offline:

```
If (offline.status = false)
  useDataFromWebService();
else
  useDataFromLocalDataCache();
```

But this approach is more preferable from a maintenance standpoint:

Node.Js

I want to mention one intriguing solution that's recently come become quite popular: Node.Js. "Node" is a server-side JavaScript interpreter with built-in server capabilities, allowing you to quickly implement a web services solution in pure JavaScript. It's very lightweight and doesn't require the user to run IIS, Apache, or anything else on the server.

JavaScript doesn't natively talk to Visual FoxPro data of course but I think someone clever could contrive a file-based messaging pipeline to get the data back and forth between the client and Visual FoxPro.

```
useDataFromLocalDataCache();
If (online or manually_requested)
    SyncLocalDataWithWebService();
```

We should never ask for our data directly from our web service, now that we're aware that it won't always be available. Instead we should always get and save data to the local storage cache using one of the technologies in the previous session. This will ensure we only have one copy of "dirty" data that needs to be synched for a given timeframe.

In addition we need our local cache to periodically synchronize its data with our web service, either automatically when we detect we're online or manually via some user interaction. We should also write some functionality into our application so that our device can get a completely fresh set of data and discard everything that's currently stored on it (useful when the database schema changes), or even wipe out the data altogether in case the device goes missing.

Attaining Synchronicity

Because I'm not assuming any particular technology is getting used in this paper, I won't have specific code on how to perform synchronicity in its entirety, but here are considerations we need to keep in mind as we plan our solution:

Conflict Resolution

We may have multiple devices out in the field adding, deleting, and changing our data simultaneously. The easiest way to ensure two records don't get added with the same primary key is to use implement GUIDs for the primary key.

We also need to decide how atomic we want our change tracking to be. Row-level conflict detection may be suitable for most applications, but in implementations where multiple devices might be simultaneously editing parts of the same record, column-level tracking might be preferable. For column-level tracking, a separate "audit" table containing the name of the table with the change, the primary key of the record that was changed, the name of the column that was changed, the timestamp of the change, and the new value would be all you need. You may also want to record the node id (device name/id), the old value, the data type of the field, or whatever else as well.

It's necessary to lay down some business rules to determine what happens in the event of a conflict. It won't be desirable to bring up an interface that allows the user to determine this manually, as most users will implore the application to accept their changes at the expense of all others. Instead we need to decide whether the "winner" is based on a timestamp (earliest or latest entered wins, or last to commit wins, for example), or maybe instead based on the seniority of the user who committed the change (aka "Management wins").

Another thing to keep in mind is that you cannot simply delete records because those deletions would never be synchronized. Instead use a deleted flag or better yet a datetime field that records when the record was deleted. Of course you'll have to add logic in your queries to exclude "deleted" records.

Bandwidth Conservation

The other consideration when synchronizing is to use as little bandwidth has possible in order to make the process as efficient on the device's battery, the user's metered usage, and the server's processing cycles as possible.

This is achieved by recording an "anchor" timestamp on the device when synchronization last occurred. When asking the master database for what's new, you should only pull down changes that occurred since the last anchor, and conversely only send up data that's changed since the last anchor. Keep in mind that connectivity may be lost while sending/receiving data, so don't update that anchor timestamp until full synchronization is successful.

Summary

While the previous whitepaper, Fox on the Run, gave specific instructions on how to implement a solution that deploys Visual FoxPro data onto a mobile web application, this whitepaper takes a broader survey of the myriad considerations and possibilities available to make that application work when it's not always online.

We reviewed the different interpretations of "offline," how to reliably detect that status, and strategies for working around being offline including ways to make our application available as well as our data. Lastly we looked at the considerations for synchronizing our locally cached data once our connectivity is restored.

As I begin to implement various combinations of these ideas, I'll be sure to update my blog at www.SaltyDogLLC.com, so check there periodically for more information.

---

[i] From http://www.html5rocks.com/en/tutorials/appcache/beginner/

[ii] Based on http://stackoverflow.com/questions/3181080/how-to-detect-online-offline-event-cross-browser