The Mobile Helix Link HTML5 SDK

by Mobile Helix

http://www.mobilehelix.com

Table of Contents

Introduction

This document provides an overview of the Mobile Helix Link HTML5 SDK. The Link SDK consists of four major components:

1. A library of JavaScript, CSS, and image resources that can be incorporated into any web-based application to build high performance, native-like mobile experiences using HTML5.
2. A JSF integration built as an extension library to PrimeFaces. Building applications using the JSF integration requires JSF2.0, a Java EE container, and PrimeFaces.
3. A series of plugins to Apache Cordova, built for Android and iOS with support planned for Windows 8 in the near future. These extensions come in two flavors: community and proprietary. As a general rule, community plugins are implemented to function without the security features of the Mobile Helix Container and the associated key management and encryption. Proprietary plugins require integration with the Link system and deliver all of the security benefits provided by Link. The security features of Mobile Helix Link are described elsewhere in the Link documentation.
4. The Link web services. These services are available to application developers who are building enterprise applications to run within the Link system and they are available to developers using Link's cloud-based service offering.

Licensing

All open source components of the Link HTML5 SDK are licensed under a BSD-style license that can be found in the source file entitled “LICENSE” in the Link HTML5 SDK download.

The open source projects provided by Mobile Helix include:

- The Link HTML5 SDK, which we describe below
- The Link Container Community Edition, which is a containerized browser on the device that includes Apache Cordova to allow HTML5 apps to access native device features. The Link Container is composed of:
    - A platform specific implementation of a containerized browser. Currently released for iOS, with Android soon to follow.
    - An integrated version of Apache Cordova 2.7.0, with appropriate modifications to integrate Cordova into the containerized browser.
    - A handful of Apache Cordova plugins that are used to enhance the capabilities available to HTML5 apps running inside of the containerized browser.

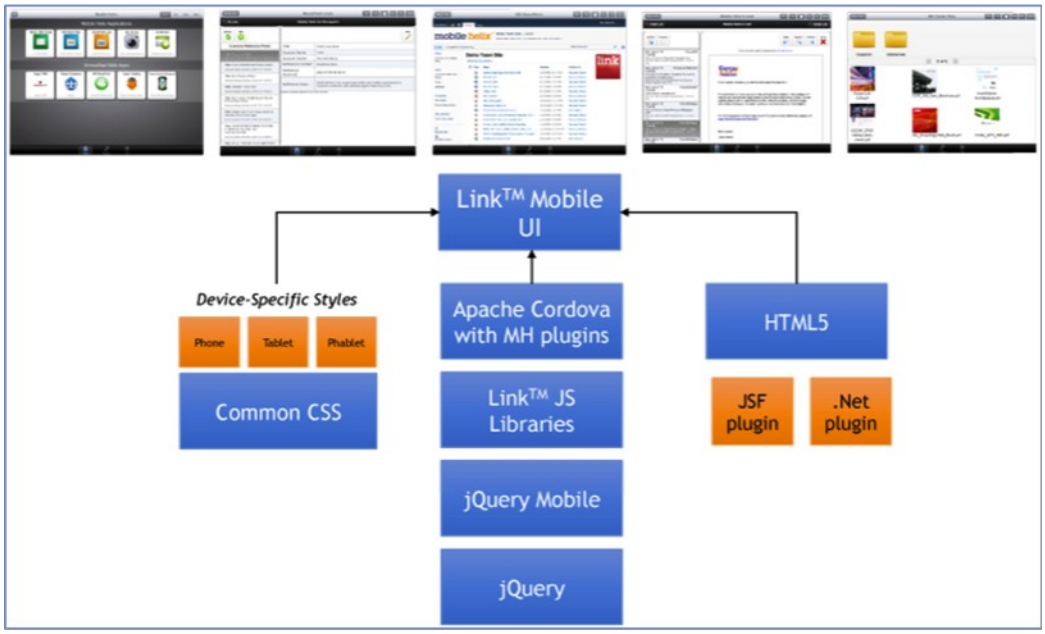
The Community Edition of the Link Container differs from the commercial version available to Link customers in its integration with the Mobile Helix Link system. In particular, the Mobile Helix Data Security Platform (documented on http://www.mobilehelix.com and outside the scope of this document) is an integrated system for user authentication, authorization, encryption, and key management that requires all of the components of the Link system. As we make components of Link available as cloud services, we do intend to release the corresponding components of the Container as open source under the same licensing terms.

How to get the SDK?

The Link HTML5 SDK and all Community plugins are available as open source software. These are available on GitHub at https://github.com/shallem.

Customers of Mobile Helix Link can also download the Link HTML5 SDK and the Link Container including all proprietary plugins from the Link support center.

The Architecture of the Link HTML5 SDK



The diagram above provides an overview of the architecture of the Link HTML5 SDK. There are three components of the SDK:

- A set of style sheets designed to render mobile-optimized HTML5 apps.
- A JavaScript library, built on top of jQuery and jQuery Mobile with extensions to support Apache Cordova and a number of enhanced features built by Mobile Helix that are described in this documentation.
- Standard HTML5, with a Java Server Faces (JSF) plugin available to make it easy to generate the HTML5 markup using JSF. We intend to release a .NET plugin in Q3 of this year. The JSF plugin is an extension to the popular PrimeFaces JSF component library.

Key Benefits of the Link HTML5 SDK

Understanding why the Link HTML5 SDK is valuable provides insight into its design, structure, and usage model. Each section below describes a benefit of the Link SDK and how that impacts application developers.

Develop Mobile Apps using Familiar Tools and Technologies

To ease the transition from the enterprise web to mobile, the Link HTML5 SDK is designed to leverage the existing tools, technologies, and processes supporting the enterprise web. Specifically, the Link HTML5 SDK:

- Enables developers to build mobile apps using existing tools for development and deployment and existing infrastructure. This means that apps built with the Link HTML5 SDK are built in

the same IDE that developers use for building any web application.

- Enables developers to debug apps using desktop browsers. Developers are already familiar with the debugging tools available in standard desktop browsers and use them regularly. Apps built with the Link HTML5 SDK are first debugged on the desktop then, once they are working in a browser, can be further debugged using FireBug Lite or any other tool of choice on the mobile device. Currently both Chrome and Safari are fully supported desktop browsers. Firefox and IE10 support are on the roadmap.
- To further the point above, Mobile Helix is increasingly ensuring that all Apache Cordova plugins available via either the standard Cordova library or Link's extensions to it perform reasonably on the desktop so that the core application code can be tested in a standard browser. This does not mean that they do exactly the same thing because there is, for example, no “iOS Photo Roll” on a desktop – it does mean, though, that when the code attempts to access the photo roll it will open the “Photos” folder to select a photo on a desktop machine. The rest of the code that uses the returned image data can then be tested on the desktop browser.
- Delivers applications via standard web services and HTTPS infrastructure. Link apps are ***not*** hybrid-native in the traditional sense. The Link Container (and its Community edition companion) are native apps that run on the device. However, each individual app does not require its own container. There is one native app on the device that is the browser environment used for downloading and running each app that you build. Apps are downloaded from your web delivery infrastructure (i.e. app servers, web servers, etc.) over HTTPS. In this respect they are no different than any other web application, whether it is targeted for mobile or not.
- Aside from the infrastructure benefits of the point above, Link HTML5 apps can dynamically generate HTML in the same fashion as any standard enterprise web application built with .NET, Java EE, PHP, or any other framework. While providing a full offline experience requires that an app function properly without access to the server, not all apps or app features make sense fully offline. In addition, even when offline the Link Container's browser cache will provide a static view of the last version of a page that the user viewed. In many cases, this static page is sufficient to make the app functional offline. The summary is that the same programming model and much of the same code used in enterprise web apps can be re-purposed for mobile use.
- For JSF developers, the Link HTML5 SDK enables a seamless migration from the desktop JSF World to mobile. There are caveats, especially for offline-enabled applications, that require that JSF developers read the section entitled “Migrating from JSF for Desktops to the Link JSF Integration.” However, the concepts and fundamentals are familiar. Beyond the conceptual integration, per the previous point, Link HTML5 SDK apps are able to re-use all of the server-side infrastructure you have already built in EJBs and Persistence layers running in your Java EE containers. The more code you can re-use, the faster and more reliably you can take your enterprise apps mobile.

Develop Native-like User Experiences

Familiar tools and infrastructure is great, but we all know that mobile apps are only as good as the user experience. Users have high expectations for intuitive, touch-first UIs. With the Link HTML5 SDK developers can build apps every bit as wonderful to use as native apps, but without the many headaches of native development, starting with the proprietary tools, architectures, SDKs, and languages that are

required to build native UIs for each mobile platform.

The Link HTML5 SDK is based on standard and open technology – specifically jQuery Mobile – and all UI components in the SDK that are not part of jQuery Mobile are built as plugins for jQuery Mobile. The Link SDK will feel familiar for any jQuery developer to pickup and run with. However, jQuery Mobile does not on its own answer the many challenges of mobile app development. The Link HTML5 SDK solves a number of specific problems that make mobile HTML5 apps challenging:

- Full screen layouts – generally a native mobile app is designed as a full screen layout with individually scrolling components inside of that layout. In an e-mail app you might have a header for navigating between the folder list and your inbox, a footer for navigating between e-mail/contacts/calendar, and in the middle a list of e-mails and the body of the current e-mail you are viewing. That part in the middle needs to scroll on its own without forcing the whole screen to scroll. Apps built in the Link SDK are full-screen apps, and all of the technology to implement independent scrolling is built into the SDK.
- Adaptive UIs – an adaptive UI is an app that looks different on a phone than it does on a tablet without requiring specific coding to each device. In Link, UI components are adaptive, meaning they are styled and act appropriately on different screen sizes without any involvement from the programmer. In addition, the standard CSS3 `@media` tag is an essential partner in making sure that your own UIs adapt cleanly. The Link SDK JSF integration makes it particularly easy to build adaptive UIs by simply placing phone vs. phablet vs. tablet styles in different CSS files.
- Full component suite – through our efforts to build the Link Email and Link Content Share applications, we have developed and integrated a comprehensive component suite for HTML5 app development. Everything is tested and incorporated seamlessly into the Link SDK. See the “Component Library” section of this documentation to learn about all of the components now at your disposal.

Develop Offline-Enabled Mobile Apps

Mobile apps should be designed to run anytime, anywhere. This requirements means that they must work offline. The good news is that HTML5 includes multiple standardized mechanisms for creating offline user experiences with HTML5. The bad news is that the restrictions and nuances make them difficult to use ubiquitously for enterprise apps. The Link HTML5 SDK is designed to make it easy to use offline apps with a familiar and cross-platform API that will feel natural for enterprise web developers. In addition, customers of Mobile Helix Link can be assured that all offline data is encrypted using Link's key management and security features.

Specifically, the challenges with HTML5 offline storage that the SDK aims to solve are:

- Enterprise apps generally download data from a server, manipulate that data, and allow users to add or modify said data in the app. This pattern is ideally suited to object-relational mapping, which is why ORM has become so popular through frameworks like Hibernate and JDO for Java, and NHibernate and the ADO .NET Entity Framework for .NET. The Link HTML5 SDK includes an enhanced version of PersistenceJS, an open source O-R-M framework for JavaScript. The Link enhancements to PersistenceJS make it easy to synchronize any JSON objects against a local database.
- There are multiple competing standards for HTML5 offline storage. To address this challenge, the SDK abstracts the underlying browser standard from the programmer via PersistenceJS.

Today, only WebSQL is supported which limits desktop debugging and mobile apps to webkit-based browsers (e.g. Android, iOS, Chrome, Safari). We are hard at work adding IndexedDB support, which will expand browser support to include recent versions of Firefox and Internet Explorer 10.

- Device-local storage is not ideal in all situations. When storing preferences, for example, it would be better to be able to optionally synchronize data across devices. The Link HTML5 SDK provides access to Link's cloud services which do exactly that. Apps built using the SDK can optionally mark certain preference-style data for synchronization across devices. Preferences are represented as simple key-value pairs, mapping keys to JSON values. To maximize performance, only changes in preferences are downloaded when an app loads, and the most recent preferences download is always available locally for offline execution.

In addition to the usability challenges of developing offline HTML5 applications outlined above, in an enterprise context security is of the utmost importance. With the Link Container's proprietary plugins, full encryption for all offline data is transparently available to application developers without requiring apps to manage keys, explicitly encrypt data, or worry about all of the associated policies that IT might want to set around data lifetime and retention.

No Vendor Lock-in

Because Link HTML5 apps can function both inside of the Link Container and in any standard browser, there is no vendor lock-in with the Link HTML5 SDK. Should you decide not to use the Link infrastructure or to have certain apps run outside of the Link infrastructure, you can do so without any changes to your app. If your app uses Cordova plugins, you can integrate it directly with the open source components available from the Apache Cordova project and with the community extensions to Cordova that Mobile Helix provides on github. In addition, as noted above, we are increasingly ensuring that when possible we provide JavaScript-only alternatives to the functions of the Cordova plugins that work appropriately on modern desktop browsers. There is not always a perfect analogy available, but we do our best to find one and integrate it into our JavaScript libraries.

What is PrimeFaces and how is it related to the Link SDK?

PrimeFaces is an extremely popular JSF component library built by Prime Teknoloji. PrimeFaces is intended for use as both a desktop component framework and a mobile framework. The Mobile Helix Link JSF integration is built as an extension to PrimeFaces, which has several benefits:

1. All components in PrimeFaces, PrimeFaces Mobile, and PrimeFaces Extensions are available for use – nothing in the Mobile Helix Link SDK conflicts with the PrimeFaces project.
2. The style of JSF development and many aspects of the implementation are familiar to PrimeFaces developers, and there is an increasingly large community of PrimeFaces developers out there.

Beyond those points, our code is not at this point in any way integrated with the PrimeFaces project. We apply our best efforts to ensure that JSF developers can use the Link SDK with any recent version of PrimeFaces.

System Requirements for Using the HTML5 SDK

Developing apps with the Link HTML5 SDK requires standard components familiar to any enterprise web developer:

- A modern, webkit-based browser. Google Chrome and Apple Safari are both recommended. Chrome version 26 or later or Safari version 5.0 or later are both sufficient.
- Any standard web server or application server
- In order to use the JSF integration, a Java EE container including JSF 2.0 is required and PrimeFaces 3.4 or later must be included in your project
- The Link SDK JSF integration requires the following additional open source components:
    - commons-lang3-3.1.jar
    - jackson-core-asl-1.9.11.jar (we do not support jackson 2.x at this time)
- The Link JSF SDK has been tested with Mojarra 2.1.11 and later

Guide to this Documentation

This documentation is divided into three sections, which introduce the major concepts in the Link SDK:

1. “Data in the Link SDK,” which introduces how the Link SDK synchronizes data to the client and how offline data works.
2. “Components in the Link SDK,” which provides an overview of the UI components available as plugins to jQuery Mobile and how jQuery Mobile and these additional plugins are used in the JSF integration.
3. “Integration with the Link System,” which particularly focuses on security and how the Link system ensures the secure delivery and execution of HTML5 apps.

Data in the Link SDK

Introduction

Essential to any enterprise mobile app is the synchronization and management of data. The Link SDK's model for data management assumes that the mobile app has the following characteristics:

5. It consumes data from one or more data sources on the enterprise network. This data may be required when a user is offline. A good example is a user's email inbox, which is stored in the mail server on the enterprise network, consumed by the mobile e-mail client, and should be available for offline access.
6. It generates data which needs to be transmitted to systems that operate within the enterprise network. Again, using the email example, composing an email is an example of data that is generated in the mobile client and needs to be transmitted back to the enterprise network.
7. It manipulates data locally, and some of those manipulations must be mirrored to the enterprise data source. Again using email, typical manipulations are to delete messages, mark them as read, or file them in a folder. These manipulations must be mirrored back to the mail server, and they may trigger UI updates in the client.

Beyond this general outline of how an app consumes, creates, and manipulates data, there are a few other considerations that, while not required, will ensure that your mobile app performs well. First and foremost is the idea of synchronizing data rather than downloading data. Again, the email example illustrates the concept well – the first time a user opens the e-mail client it is sensible to expect that the user waits a minute or two while the app downloads the user's inbox. Thereafter, the app should simply synchronize changes – adds/updates/deletions – and hence only a small amount of data should travel between the enterprise network and the mobile app when the app is used frequently. The Link SDK data model supports the concept of transmitting delta objects, which add, update, or delete data on the client, as an alternative to re-transmitting the same baseline data each time the app connects to the enterprise data source.

The second important concept built into the Link SDK is object storage. This documentation is by no means the appropriate place to introduce object oriented concepts and programming. However, the point of most importance is that the developer should, as much as possible, view the underlying data storage mechanism as opaque to the mobile app. The Link SDK allows objects and relationships between those objects to be stored locally and retrieved when offline. The interface to local data storage is via PersistenceJS, which manipulates objects and stores them to local data storage using standard HTML5 APIs. Whether it is WebSQL, IndexedDB, or something else under the covers should remain as opaque to the app and the app developer as possible.

A Brief Introduction to PersistenceJS

PersistenceJS is a very simple object storage mechanism. Persistent objects are defined using PeristenceJS schemas, and objects are generated from these schemas and populated in your app just like any other JavaScript object. When new data is created and is ready to add to local storage, the JavaScript object is passed as an argument to the global persistence object. That object is then tracked

by the global persistence object and flushed to the local storage mechanism when the `flush` method is invoked. It is essential that you review the PersistenceJS documentation before proceeding any further. The documentation is short, but it introduces essential concepts in PersistenceJS, including object schemas, which are referenced heavily in the text below. To learn about PersistenceJS, go to http://www.persistencejs.org.

The Link SDK builds on top of Persistence JS to add three major components of functionality:

- Automatic generation of PersistenceJS schema from JSON template objects. This is described in the next section. The idea is that rather than explicitly creating multiple PersistenceJS schemas and the relationships between them, the app defines a template object that has the appropriate fields, relationships, and data types. This template object's data may be mock-up data. From this template object, the Link SDK generates and tracks all of the PersistenceJS schema objects required to store objects that follow the defined template.
- Automatic synchronization of JSON object data to local storage via the schema objects created as described in #1. Synchronization includes the ability to specify delta objects at any point in the object hierarchy. Delta objects allow an object to specify changes to data that is already stored by the client.
- Automatic schema tracking. This allows developers to change the schema by simply changing the template objects and associated data. These changes will be automatically mirrored to local storage.

Each of these concepts is described in further detail below.

Synchronizing Data: Schemas

While schema objects may be created explicitly using PersistenceJS, the preferred method for creating schemas in the Link HTML5 SDK is to generate them automatically using a template object. A template object is simply a data-less JSON object that can be explored via reflection to infer the schema. For example, the following object would be a reasonable template for the inbox object in an email app:

```
{
    “name” : “”, /* Unique name of this e-mail folder (e.g., 'inbox') */
    “owner” : “”, /* Owner is a string property. */
    “lastSynchronized”: “”, /* Dates are stored as strings. */
    “messageCount” : 0, /* Number of messages in the inbox. */
    “messages”: [{ /* The inbox has a one-to-many relationship with messages. */
              “senderEmail” : “”, /* String representing sender email. */
             “receiveDate”: 1, /* Dates are represented as MS from the epoch. */
              “senderSubject”: “”, /* String representing the subject. */
              “htmlBody” : “”, /* HTML text of the body. */
      }]
}
```

These template objects are then explored via reflection and automatically converted into a

PersistenceJS schema object via the function `Helix.DB.generatePersistenceSchema`. For more detail on this function, please consult the javadoc documentation.

The object above accurately describes the structure of our simplified inbox, but it does not supply quite enough information to generate the schema. This additional information is supplied via special fields that can appear in any object or sub-object within the schema template. These special fields are:

Field Name	Description	Required?
`__hx_schema_name`	Name of the schema. This name is used in the app to get a reference to the PersistenceJS schema object for this object. This schema object can then be used to create new objects that should be stored locally or to query the local storage based on the values of object fields.	Yes
`__hx_key`	Name of the primary key field for this object. This field must be unique for each distinct object.	Yes
`__hx_sorts`	JSON array of fields that are likely to be used on the client for sorting (e.g., email sender, date email was received, etc.).	No
`__hx_filters`	JSON array of field names that are likely to be used on the client for grouping or filtering.	No
`__hx_type`	Specify the special value 1001 to indicate that this object is a delta object.	Only for delta objects

These special fields must be added to the template object above to make it complete and ready for schema generation:

```
{
    “__hx_schema_name” : “Folder”,
    “__hx_key” : “name”,
    “owner” : “”, /* Owner is a string property. */
    “lastSynchronized”: “”, /* Opaque string sent to the server on each load. */
    “messageCount” : 0, /* Number of messages in the inbox. */
    “messages”: [{ /* The inbox has a one-to-many relationship with messages. */
              “__hx_schema_name” : “Message”,
              “__hx_key” : “messageID”,
             “__hx_sorts” : [“receiveDate” , “senderEmail” ],
              “messageID” : “”, /* Unique GUID for this message. */
              “receiveDate”: 1, /* Dates are represented as MS from the epoch. */
              “senderEmail” : “”, /* String representing sender email. */
              “senderSubject”: “”, /* String representing the subject. */
              “htmlBody” : “” /* HTML text of the body. */
      }]
}
```

Once a schema is generated, it is stored in the global map `_hxAllSchemas`. Individual schemas are accessible from that object by name. The schema names correspond the values of the

`__hx_schema_name` field in the template objects.

Synchronizing Data: Data Objects

Once schema objects are created or generated, data can be synchronized to local storage. Data objects follow the schema templates in structure, but they are populated with real data (e.g., the email messages in a user's inbox). In addition, the only special fields required in the data objects themselves are the `__hx_schema_type` field, which identifies the schema object corresponding to any JavaScript object, and `__hx_type` which, as described above, identifies delta objects when they appear in the JSON object hierarchy.

For example, the data object for a user's inbox might appear as follows:

```
{
    “__hx_schema_type” : “Folder”,
    “owner” : “Seth Hallem”,
    “lastSynchronized”: “2013-05-29T11:24:00”,
    “messageCount” : 100,
    “messages”: {
              “__hx_schema_type” : “Message”,
              “__hx_type” : 1001, /* Delta object. */
              “adds”: [{
                  “messageID” : “D46563E61A1949F889DA4846D4748504”,
                  “receiveDate”: 1369099797000,
                  “senderEmail” : “support@mobilehelix.com”,
                  “senderSubject”: “Follow-up to your inquiry”,
                  “htmlBody” : “<div>Dear Mr. Smith,<p>Thank you ...</p></div>”
               }]
    }
}
```

This data object would then be supplied to the JavaScript function `Helix.DB.synchronizeObject`, which iterates through the object using reflection and updates the local storage appropriately. Note that the above example uses a delta object to indicate that a new message was added to the user's inbox. Using a delta object allows the server to send changes to the client each time the client asks the server for new information rather than re-sending the entire inbox each time a synchronization between the client and the server must occur.

For a full documentation of the `Helix.DB` API, please see the generated documentation for the file `persistence.helix.js`.

Synchronizing Data: Schema Changes

As an app evolves, the structure and fields that it must store locally on the client change. The Link SDK automatically tracks the structure of each object that is stored locally, including its fields, their types, and the object's relationships to other local objects. When these aspects of an object change, the schema generation code recognizes the change and migrates the underlying data store appropriately. From the programmer's perspective, changing the local object schema is as simple as updating the template object and the corresponding data objects.

The JSF loadCommand Entity

The load command is a JSF component built into the Link SDK that encapsulates an AJAX load and local data synchronization of a server-side object. For example, the inbox we have been discussing might be represented on the server-side in an EJB as an object consisting of a list of messages (stored as an array) and a list of attributes (stored as object properties). The `hx:loadCommand` entity generates a series of JavaScript calls that encapsulate the generation of schema objects, the generation of schema using the API described above, the serialization of object data stored in a server-side bean, and the client-side storage of that data.

Parameters of a Load Command

The load command entity accepts parameters for three purposes.

The first purpose is to transmit request data to the server so that the server can load the appropriate information. These parameters are supplied as standard URL-encoded parameters in an HTTP post request sent to the JSF servlet. A typical usage for such parameters is to send a synchronization key, which indicates that last time the client synchronized with the server. This key would then be used to generate a delta object representing changes since the last synchronization.

The second category of parameters are used to display a meaningful loader while the load is in progress. This includes a message to display in the standard jQuery Mobile loader component and the theming you prefer for that component.

The third category of parameters determine how to synchronize data retrieved from the server with local storage. These parameters include the schema object for the serialized data (described below), the client-side widget variable used for storing the result of the load, and the completion function to be called after the local synchronization is complete.

loadCommand Parameters

The `hx:loadCommand` entity accepts the following parameters:

Parameter Name	Parameter Type	Required?	Description
name	String	Y	Name of the JavaScript function generated from this markup. This function must be called from the client-side to invoke the AJAX load.
error	String	N	String property containing an error message, if one

			occurs during execution of the load command. If this command is not specified then a null object is returned by the load command upon error.
oncomplete	String	N	JavaScript function to execute when the load is complete. The first parameter to this function is the load status (either “success” or “error”).
widgetVar	String	Y	Name of a JavaScript variable with global scope that will be used to store the data object after it has been downloaded and synchronized. This data object while be a PersistenceJS data object. For more information on PersistenceJS data objects visit https://github.com/zefhemel/persistencejs.
loadingMessage	String	N	Message display in the jQuery Mobile loader while the load command is executing.
loadingTheme	String	N	Theming for the jQuery Mobile loader. For more information on loader themes, visit http://api.jquerymobile.com/page-loading/.
value	JSF EL	Y	Specifies the server-side bean property used to retrieve the data that is synchronized to the client. The value getter will be invoked in two circumstances – first, during initial page rendering to generate the object schema for the returned value, and second whenever the load command is invoked. The JSF bean must handle the first case where the value getter is invoked without the “cmd” listener being invoked. In this case, the bean **must not return null**. Instead it should return an empty object, which can then be walked via reflection to find data annotations (see below).
cmd	JSF EL	Y	Specifies a function to invoke prior to getting the value. A typical pattern is to use the actionListener to load data into the JSF bean (e.g., from a database or a web service), store the loaded object in the bean, then retrieve it with the value getter.

loadCommand Phase Listener

The loadCommand phase listener enables JSF load commands to be invoked without regard to the JSF view state. Mobile applications have two key attributes that inter-operate poorly with JSF's view expiration mechanism. First, mobile apps developed using jQuery Mobile are most often delivered as a single page within a single get request. This means that the JSF view is never refreshed once it is created by the initial get request to the JSF servlet. When the JSF session expires after the specified session duration configured in web.xml, the JSF view also expires. At that point, all queries back to JSF beans, even if they are view scoped, will fail with a `ViewExpiredException`.

The second, related poor interaction occurs because mobile apps are designed to support offline access. Because a user might work offline for a few hours then return to online usage without ever refreshing the JSF page, the same view expiration issue can occur through a very normal usage pattern. While one solution is to set the session duration to be extremely long, this has the effect of forcing the JSF servlet to retain session state for a very long time, which correspondingly loads down the app server unnecessarily and does not scale well to large numbers of users.

As an alternative, the loadCommand uses a phase listener to instantiate the backing bean required to execute the load command on the fly, generate the desired result, and hand back the resultant JSON object to the client. If an application is intended to work offline, then load commands ***should*** be the only mechanism by which the page on the client interacts with JSF backing beans. If that is the case, then the JSF session duration can be set to a single minute to minimize server load, and load commands can happen at any time due to the load command phase listener.

NOTE: the loadCommand component will not function unless the phase listener is installed!

To install the load command phase listener, add the following markup to faces-config.xml in your application:

```
      <lifecycle>
        <phase-listener>org.primefaces.mobile.filters.LoadCommandListener</phase-
listener>
      </lifecycle>
```

Generating Data Schemas in JSF

When using the JSF integration, the `loadCommand` markup automatically generates the object schema according to the returned object from the `value` attribute getter . This schema skeleton is generated via Java's reflection API, using programmer-inserted annotations to guide the serialization process.

NOTE: Due to the "type erasure" property of Java generics, returned objects cannot use Java generics! For more information on type erasure, visit
http://docs.oracle.com/javase/tutorial/reflect/member/fieldTypes.html. Instead of using generics, used typed arrays of objects.

The following annotations must be specified to guide the schema generation and object serialization process. For more information on each annotation, consult the javadoc for the `org.helix.mobile.model` package.

Name	Required?	Parameters	Description
ClientData	Y	None	At least one getter in each object that is intended for synchronization to the client must be marked as ClientData. Functions marked as such must have the form `get<field name>` or `is<field name>`, similar to any standard JSF getter. Returned types from annotated getters can be any primitive type, object type, or an array of objects.

ClientDataKey	Y	None	Exactly one getter in each object should *also* be marked with the ClientDataKey annotation, which is a unique key field (can be a string or integer) identifying the object. An index with a unique constraint is created in the client-side database on this field.
ClientSort	N	displayName : String	Mark 0 or more getters as fields that should be indexed on the client. The displayName parameter is used by components such as the dataList to automatically generate a list of sortable fields that a user can choose from when sorting that list.
ClientFilter	N	displayName: String	Conceptually similar to the ClientSort annotation, but used for specifying a field that is used to filter or group a list, rather than sort it. Filters are used by the dataList component to make it easy for a user to restrict the list to objects matching the current selection, and groupings reformat the list grouped by the unique values of the field with this annotation.

The object below is a simple example of an annotated object that will be synchronized to the client.

```
/**
 * Encapsulates a single SharePoint site downloaded via SharePoint web services.
 *
 * @author Seth Hallem
 */
public class SharepointSite {
    /* A convenience name for the site. */
    private String siteName;

    /* The URL of the site. */
    private String siteURL;

    /* A map from UUID to metadata objects for the lists included in this site.
     */
    private Map<String, ListMetadata> siteLists;

    /* Constructor */
    public SharepointSite(String siteURL,
            String siteName,
            Map<String, ListMetadata> siteLists) {
        if (siteName != null) {
            this.siteName = siteName.trim();
        }
```

```
        this.siteURL = siteURL;
        this.siteLists = siteLists;
    }

    /**
     * Getter for the siteName field. Returns the name of the site, which is a
     * field we want to allow users to use to both sort and filter the list
     * generated on the client-side via the pm:dataList component.
     *
     * @return The name of the SharePoint site.
     */
    @ClientData
    @ClientSort(displayName="Site Name")
    @ClientFilter(displayName="Site Name")
    public String getSiteName() {
        if (this.siteName != null && !this.siteName.isEmpty()) {
            return this.siteName;
        }
        return this.siteURL;
    }

    /**
     * Getter for the siteURL field. This field represents the full URL that
     * uniquely identifies this SharePoint site. This is the one (and only)
     * field in this object marked as the key. This generates a unique index on
     * the client side for this object.
     *
     * @return The URL of the SharePoint site.
     */
    @ClientData
    @ClientDataKey
    public String getSiteURL() {
        return this.siteURL;
    }

    /**
     * Getter for the lists encapsulated within this site. SharePoint organizes
     * data within a site into a variety of different lists. The ListMetadata
     * objects provide an overview of each list contained within this site. On
     * the client side, a separate table is generated for the ListMetadata
     * objects and a one-to-many relationship is automatically generated between
     * the SharepointSite objects and the ListMetadata objects.
     *
     * @return Array of MetaData objects.
```

```
     */
    @ClientData
    public ListMetadata[] getSiteLists() {
        ListMetadata[] listsArr = new
                ListMetadata[this.siteLists.values().size()];
        return this.siteLists.values().toArray(listsArr);
    }

    /**
     * Getter for the map from unique list names (UUIDs) to the associated
     * metadata object. Note that this getter is NOT synchronized to the client
     * and it cannot be synchronized to the client because the return type is a
     * generic. However, within the JSF beans manipulating these objects we
     * continue to use the map data structure rather than the flattened array
     * that we synchronize to the client.
     *
     * @return Map from list UUID to associated metadata.
     */
    public Map<String, ListMetadata> getListsMap() {
        return this.siteLists;
    }
}
```

On the client, each object type that is reachable from a serialized object is turned into a single PersistenceJS schema. Each schema is named using the fully-qualified object name of the serialized object (e.g., `com.mobilehelix.appserver.sharepoint.Lists.SharepointSite` for the object above).

Delta Objects in JSF

Delta objects are an essential component of an app's data synchronization strategy, as they offer an important alternative to re-sending the same baseline data each time an app synchronizes with the enterprise data source. A perfect use case for delta objects is the email inbox. The inbox may have many messages in it, but if the mobile email app is used frequently the set of changes between synchronizations is likely small.

Delta objects in JSF must implement the `org.helix.mobile.model.DeltaObject` interface. This interface is documented in the generated documentation supplied with the Link SDK download package.

JSF Example

```
      <hx:loadCommand name="loadList"
                      widgetVar="currentList"
                      value="#{listView.selectedList}"
                      actionListener="#{listView.loadList()}"
```

```
                oncomplete="listLoadComplete(currentList);"/>
```

The markup above generates a JavaScript function called `loadList` that downloads data from the `listView` JSF bean. The downloaded data is stored according to a schema that is automatically generated from the return object of the `selectedList` getter. Prior to invoking the getter, the function `loadList` is invoked on the backing bean to download the list of data. Once synchronized, the loaded data is stored in the globally scoped variable `currentList` JavaScript variable and the completion method is invoked. The completion method invokes another locally defined JavaScript function (not shown here) and it passes the now defined widget variable as a parameter.

Summary

The Link SDK is designed to make it easy to serialize data that originates from an enterprise server and synchronize that to a mobile client. The client can manipulate data, generate new data, and, ideally, synchronize deltas with the server. This sequence can be accomplished either via the Link SDK's JavaScript APIs or, for those using JSF, via the `hx:loadCommand` JSF entity. For more details on the JavaScript APIs, please consult the generated documentation supplied with the Link SDK download package.

The Link HTML5 SDK Component Library

Introduction

Components are user interface controls and styles that, when combined together in a single page, create the mobile user experience. The Link SDK's component library is built on top of jQuery Mobile, and many of the components are enhanced versions of what is already available with jQuery Mobile.

Components may either be placed in HTML markup directly using stylized attributes to identify the component, or they may be generated from JSF entities when the JSF integration is used. All components follow the jQuery model of enhancement – the markup for a component is specified using simplified HTML tags and data-* attributes to identify components and parameters. When the markup is loaded, the markup is "enhanced" into the full layout and styling required to present a visually rich component in the browser.

This documentation is not an exhaustive documentation of jQuery Mobile. The components described here are only those that either do not exist in jQuery Mobile or have been enhanced beyond what jQuery Mobile provides. All of the functionality in jQuery Mobile is available to your app, however, and if you do not see a component here that would help your app please consult http://api.jquerymobile.com/.

Pages in jQuery Mobile

jQuery Mobile apps are composed of pages with each page corresponding to a single screen within an app. A typical app is delivered as a single HTML document with a number of pages contained within that document. The main advantage of creating apps in this fashion is that once the app is loaded on the mobile client, the client does not need to contact the server again to download additional resources, scripts, or HTML pages. Transitions between screens in the app are extremely fast giving the app the feel of a native app. In addition, because the Link Container and Link Gateway aggressively prefetch, compress, cache, and de-duplicate data, apps load extremely quickly when delivered through the Link infrastructure.

Switching between pages is achieved via hashtag navigation. Hashtag navigation can occur directly via `<a/>` markup as documented extensively in the jQuery Mobile documentation or via JavaScript. The Link HTML5 SDK borrows a simple wrapper for jQuery Mobile's JavaScript-based navigation API the PrimeFaces Mobile project. The function `PrimeFaces.navigate` allows JavaScript code to initiate a page change. The details of this function are documented in the generated API documentation included with the Link SDK.

Each page is actually represented in the HTML markup as a `div` tag with the `data-role="page"` attribute. For apps that are not built using JSF, it is easy to create this markup directly. Again, http://api.jquerymobile.com/ is the definitive source for understanding how to write HTML code for jQuery Mobile.

JSF Integration

jQuery Mobile apps generated via the Link HTML5 SDK's JSF integration follow the simple structure diagrammed below:

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:pm="http://primefaces.org/mobile"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:hx=”http://mobilehelix.com/jsf”
    contentType="text/html">
   <hx:page title="#{bundle.SharepointListBrowserTitle}">
        <f:facet name="postinit">
            <hx:outputStylesheet name="main.css" library="css" version="1"/>
            <hx:outputScript name="sharepoint.js" library="js" version="1"/>
        </f:facet>

        <hx:view id="AllLists">
            <pm:header title="#{bundle.ListsListLabel}"
                       fixed="true">
            </pm:header>
            <ui:include src="allLists.xhtml"/>
        </hx:view>

        <hx:view id="ListView">
            <pm:header title="#{bundle.SharepointListBrowserTitle}"
                       id="FolderHeader"
                       fixed="true">
                <f:facet name="left">
                    <p:button value="#{bundle.SharepointListsLabel}" icon="back"
                              href="#AllLists?reverse=true"/>
                </f:facet>
            </pm:header>
            <ui:include src="selectedList.xhtml"/>
        </hx:view>

     </hx:page>
```

```
</f:view>
```

The markup above generates a two screen app. The first screen, called “AllLists” is a listing of all SharePoint lists available for display in this app. It could equivalently be a list of all folders in an email user's mailbox. The second screen, called “ListView,” is used to display the detailed contents of a single list. Again an email app would be structurally similar, with the “ListView” view used to display a single mail folder.

Note that the markup example shows a mixture of `hx:` tags and `pm:` tags. The former are tags that are developed by Mobile Helix or are modified versions of the tags supplied with PrimeFaces mobile. All `pm:` tags are part of PrimeFaces mobile and are not modified by Mobile Helix.

In addition to the overall structure, the markup above demonstrates a number of important tags:

8. The `hx:page` component can have a “postinit” facet, which allows HTML markup to be generated after all other components of the HTML `<head>` entity. The main purpose of this facet is to output the stylesheets and scripts that are specific to the app. Using this facet ensures that these stylesheets and scripts are inserted into the generated HTML after all of the integrated CSS and JavaScript that is included in the Link HTML5 SDK. Inserting your own stylesheets after the Link SDK stylesheets allows your stylesheets to override the Link SDK's built-in styles.
9. The `pm:header` entity creates a header bar across the top of the app screen that is a convenient place to add navigation buttons and to display a title for the current screen. The header for the “ListView” page, for example, includes a back button that navigates back to the “AllLists” screen.

The specific markup for each page is not shown in the example above. However, this example does illustrate how organizational features of JSF, such as `ui:include`, are fully compatible with the Link SDK.

Full Screen Layouts

Native apps, mobile or otherwise, are presented to the user as full screen layouts with an arrangement of individual components within the screen layout. Different components may scroll horizontally or vertically. By contrast, web pages are commonly a single, long body of content that is meant to scroll within the browser screen. Mobile apps built with the Link SDK attempt to combine the convenience of HTML5 with the look and feel of native apps. To that end, screen layouts in the Link SDK are full screen. What this specifically means is that the SDK forces the `overflow: hidden` style on the `div` corresponding to each screen in the mobile app. Hence, the screen itself cannot scroll. Components within the screen can scroll, though, as described in the “Scrolling Div” section below.

To assist with creating full screen layouts that adapt to different screen sizes, the Link SDK defines the style `hx-layout-full-screen`. This style, when applied to a component, essentially means that the marked component should scale to fill in the rest of the available screen space. For example, a screen layout may have a header for navigation, a button bar below that header with buttons corresponding to operations available on that page, then a list of data. The size of the header and the button bar are fixed, either as a hard-coded pixel size or a hard-coded portion of the screen. The data list, though, should

simply scale to fill in whatever screen real estate is left. That is exactly what marking the data list with the `hx-layout-full-screen` style class achieves.

The Scrolling Div

Implementing full screen layouts requires the ability to create independently scrolling components in the layout. The scrolling div component allows developers to do exactly that – to mark any container component (generally a div, but it does not need to be a div) – as a component that should scroll independently, either horizontally, vertically, or both. In addition to the scrolling, this component also allows developers to implement the standard iOS “double tap to zoom” on individual components within a page. The default double tap behavior (without this component) is to zoom in on the entire page.

The scrolling div component is implemented by applying one of the following style classes to a container component:

- **`hx-scroller`**: indicates that the enclosed markup will scroll vertically and will allow double-tap to zoom on the contents of the enclosed markup
- **`hx-scroller-nozoom`**: indicates the enclosed markup will scroll vertically but not allow double-tap to zoom
- **`hx-scroller-zoomonly`**: indicates that the enclosed markup will not scroll but will allow double-tap to zoom
- **`hx-scroller-horizontal,hx-scroller-horizontal-nozoom,hx-scroller-horizontal-zoomonly`**: same as above, but for horizontal scrolling

Container components must adhere to the following restrictions to work properly:

- The container should have exactly one child element in the DOM.
- The container itself must have a fixed height. If the container is the last child element in a full screen layout then this fixed height will be assigned automatically. Otherwise it must be assigned explicitly.
- The child element's height must match the height of the entire set of data to be displayed in the scrolling component. The child element, hence, “spills out” of the container. The container's scrolling capability allows the user to navigate to hidden content that is not visible within the limited height of the wrapper. To ensure that the child element's height matches the height of all data to be displayed within the scroller, do not specify any height attributes on the child component.

JSF Integration

The JSF integration enables the generation of a scrolling div surrounding arbitrary markup. The example below illustrates how to generate a scrolling div and the attributes available to control the

generated markup:

```
<hx:scrollingDiv
      orientation="horizontal"
      zoom=”true”>
        <ui:include src="subdirs.xhtml"/>
 </hx:scrollingDiv>
```

The Split View Component

The split view component enables two column views that are very common in tablet-based user interfaces. The left column is generally a list of data, while the right presents further detail for the selected component. A perfect example is the native iPad e-mail client's inbox, with a list of e-mails down the left column and the message detail to the right. The Link split view component is an adaptive component, which means that a phone the view automatically adjusts such that the left column occupies the full screen and, upon tapping, the user opens the detailed view that is in the right column as a full screen view.

Split views are implemented via a jQuery plugin that is invoked as follows:

```
$('selector').splitView(genLeftContent, genRightContent);
```

The selector refers to the parent component of the split view columns, and the `genLeftContent` and `genRightContent` parameters are single-argument callbacks invoked with a jQuery object encapsulating the parent div for the left and right components respectively as the only argument. The markup for the left/right views can should be attached to this jQuery object using jQuery's DOM generation APIs.

JSF Integration

The Link JSF integration makes it easy to specify a split view declaratively as follows. This example is a simple layout for an e-mail inbox. Each side of the split view has a number of action buttons placed at the top of that portion of the view, allowing users to compose and respond to e-mails.

```
<mh:splitView>
    <mh:splitLeft width="30"> <!-- left view occupies 30% of width. -->
        <pm:buttonBar orientation="Horizontal"
                      fixed="true">
            <mh:iconButton value="#{bundle.GetMessagesLabel}"
                           image="mh-get-mail"
```

```
                        onclick="reloadFolder()"/>
                <mh:iconButton value="#{bundle.ComposeLabel}"
                        image="mh-compose"
                        onclick="doCompose()"/>
            </pm:buttonBar>

            <ui:include src="messagelist.xhtml"/>
        </mh:splitLeft>

        <mh:splitRight>
            <pm:buttonBar orientation="Horizontal">
                <mh:iconButton value="#{bundle.ReplyLabel}"
                        image="mh-reply"
                        onclick="doReply()"
                        align="right"/>
                <mh:iconButton value="#{bundle.ReplyAllLabel}"
                        image="mh-reply-all"
                        onclick="doReplyAll()"
                        align="right"/>
                <mh:iconButton value="#{bundle.DeleteLabel}"
                        image="mh-delete"
                        onclick=”doDelete()”/>
            </pm:buttonBar>

            <ui:include src="messageview.xhtml"/>
        </mh:splitRight>
    </mh:splitView>
```

The Offline Data List

The offline data list is a component designed to render an arbitrary list of data. It is built on top of the jQuery Mobile datalist component, and all features of the jQuery Mobile datalist are available in this component. Beyond the jQuery features, though, the datalist component is designed to seamlessly display and interact with data that is synchronized to the device for offline access.

The datalist component can be constructed directly in JavaScript using the following jQuery plugin:

```
$('selector for div').hxDatalist(<Datalist options>);
```

Aside from the standard jQuery Mobile options controlling the rendering of the datalist, the following configuration options are added to the datalist to manipulation and display of offline data:

- i**temList**: PersistenceJS QueryCollection containing all items that should be displayed in the list
- **condition**: JavaScript boolean condition that determines if the datalist should be hidden or should be displayed. This option allows the SDK to dynamically determine if the list should be hidden or visible.
- **sorts**: Mapping from field names that should be available to the user for sorting the datalist to display names for those fields. The fields are column names in the PersistenceJS schema. The display names are used in the user interface to allow the use to control sorting.
- **sortBy**: Name of the field used for sorting the list when it first loads and prior to any user interaction with the generated sort menu.
- **sortOrder**: The default sort order, either “ASCENDING” or “DESCENDING”.
- **grouped**: Specifies that the data list is grouped with named dividers or collapsible dividers in between groups. If a list is specified as grouped, then the **groupName** (required) and **groupMembers** (optional) options are also used.
- **groupName**: A callback invoked on each row in the **itemList** to map that item to its group.
- **groupMembers**: A callback invoked on each group name to retrieve all members of a group. This option should be used when the supplied data is hierarchical – in other words, the **itemList** is a list of rows corresponding 1-to-1 to groups, and a sub-query should be executed to retrieve the rows for each group. When this option is not specified, all items with the same computed group name are automatically grouped together, and group dividers are inserted automatically when the group name changes. In this latter case, the list must be sorted by the field corresponding to the group name.
- **rowRenderer**: Callback used to create markup for a single list row. Arguments supplied to the call back are the enclosing markup for the row, the datalist object, the row of data to be rendered, the index of the row in the **itemList** collection, and an array of strings, derived from the **strings** option below.
- **strings**: A comma-separated list of strings that are passed through the the row renderer. The main purpose for this option is allow the JSF integration to specify localized strings for use in the client-side rendering of the component.
- **emptyMessage**: String to display when the datalist has no rows.
- **emptyGroupMessage**: String to display when a group in a grouped list has no members.
- **itemContextMenu**: ID of a jQuery Mobile context menu that should be dislayed on tap-hold of a single item in the datalist.

In addition to the options available when creating the datalist, the datalist is a dynamic component that can be updated via an API on the client. The most important function is the **refreshList** function, which re-renders the entire list based on the updated **itemList** (argument 1), **condition** (argument 2), and **sorts** (argument 3), and completion function (argument 4) supplied to this routine.

JSF Integration

The JSF integration encapsulates the options above in convenient markup. For example, the datalist markup below is used to specify the left-hand column of Link's app for browsing SharePoint lists:

```
<mh:dataList
      selectable="true"
      itemList="getListItemsFromList(currentList)"
      rowRenderer="renderListItemSummary"
      widgetVar="spList"
      selectAction="loadListItem(row)"
      tableStyleClass="leftListBarTable"
      sortBy="modifiedDate"
      sortOrder="descending"
      emptyMessage="#{bundle.NoItemsInList}"/>
```

The markup above renders a list of summaries of items in a SharePoint list that is stored for offline access in local storage. Whenever an item is selected by the user, it is loaded dynamically with the `loadListItem` action, which is also used to update the left side of the split screen view that encapsulates both the datalist and the detailed view. The markup also specifies the renderer for each row, a client-side variable for referencing the hxDatalist object, and strings that can be localized by the web server to deliver localized messages to end users. As the content in the list evolves during the lifetime of the application, the **widgetVar** name can be used to reference the datalist object and call its refresh method as described above.

The Editor

The editor component is a rich text editor that leverages the “design mode” feature of modern browsers to allow users to create HTML formatted text. The markup for this text can then be retrieved by the app and sent back to the server as appropriate for the app. For example, the editor could be used in an e-mail application to compose HTML emails.

A component may be marked as an editor in plain HTML by invoking the PrimeFaces.cw function supplying the appropriate arguments:

```
$('selector for div').editor(<Editor options>);
```

Options available for the editor include:

- **width**: the pixel width of the editor
- **height**: the pixel height of the editor

- **fonts**: comma-separated list of fonts to use the fonts drop-down menu in the editor
- **bodyStyle**: style attribute to apply to the body of the iFrame used for creating rich text
- **change**: callback to invoke when the contents of the editor change. The `this` object in the callback refers to the editor object. The HTML in the editor is accessible via `this.getHTML();`
- **isFullHeight**: indicates that the editor should occupy all available vertical space on the page, as opposed to a fixed height
- **isFullWidth**: indicates that the editor shoud occupy all available horizontal space on the page, as opposed to a fixed width.

JSF Integration

An editor may be placed in the generated HTML via the `hx:editor` entity. The attributes available for this entity correspond to the options listed above.

Optimizing Apps for Performance and Security

Optimizing App Performance

One of the many challenges in developing a high performance mobile app is the inherent limitations of the mobile network. The mobile network has high latency under most conditions, is generally unreliable, and is prone to fluctuations in bandwidth. One very coarse conclusion from those general conditions is that apps that minimize the amount of data sent over the network and the number of times they access the network are likely to perform far better than those that use the network more freely.

When building an HTML5 app with the Link SDK, the jQuery Mobile page model encourages all of an app's HTML markup to be encapsulated into a single HTML page. This model allows the entire HTML markup for the app to be downloaded to the device with a single GET request. However, a standard browser running outside of the Link system will have to subsequently fetch all referenced stylesheets, images, and JavaScript files that are referenced from that HTML page. The best practice for managing these resource downloads is to (a) combine and minify all CSS stylesheets into one master stylesheet, (b) combine and minify all JavaScript files into a single master script, and (c) combine all images into a single image using CSS sprites. Following this best practice, downloading an app still requires 4 GET requests – one each for the markup, the combined CSS, the combined JavaScript, and the image sprite.

Securing Apps

Security is of paramount importance in any enterprise app, and the Link SDK includes a number of features to encourage secure development. Beyond those developer-targeted features, when delivered and executed using the Link system, apps are transparently secured as described below. In this section we focus exclusively on the security of the app front end (i.e., the HTML5 piece) – we do not address security considerations in developing web services or business logic code that resides within the enterprise network.

Creating secure HTML5 apps requires securing the following critical aspects of the app:

10. Authentication – it is essential to ensure that only authorized users are able to access your app
11. Protecting data in transit via HTTPS
12. Protecting data at rest via AES-256
13. Following best practices for secure coding

Starting with the first point, any HTML5 app developed for execution outside of the Link system must implement authentication. When executing inside the Link system, the Gateway prevents any unauthenticated access to your internal network as described below. Following the jQuery page model, we recommend the following practice for securing app access:

- Create a separate page for authentication – do NOT make authentication code a component of the main app page. This ensures that HTML markup, JavaScript, CSS, images, and other potentially sensitive information is not exposed simply by downloading the app.
- Monitor session access using secure session IDs. Ensure that these are at least 32 characters

(256-bits) and are generated using a true secure random number generator.

- On the server side, use a filtering mechanism to prevent access to the main app page and all other services that your app will access via JSF. Almost all server-side infrastructures like Java EE and .NET provide a filtering mechanism that ensures that all communications can be checked for the presence of a valid secure session ID. When possible, leverage what the server-side application server provides out of the box.

Once you have ensured that your app only allows access for authenticated users, the next step is to ensure that all communications sent to and from your app are protected appropriately. The best way to do is via HTTPS. The best practices for adding HTTPS to your app are:

- Only allow communication over HTTPS. Do not automatically redirect HTTP traffic to HTTPS. Simply display a page on any HTTP contact that informs the user that he/she must use HTTPS instead. All HTTP traffic is vulnerable to compromise, including the automatic redirect.
- Ensure that you install a signed certificate on the server hosting your app. This certificate not only eliminates unfriendly warning messages from desktop and mobile browsers when accessing your app, it also is a certification of the authenticity of your company. The Link Container will not accept untrusted certificates, and intelligent users will not do so either.
- For apps developed for in-house use, consider using a certificate generated by your own enterprise CA. The Link Container allows you to override the device's list of trusted CAs with your own alternative. The Link system uses this mechanism to guarantee that all apps that run in the container originate from the enterprise network and not any 3rd party with a valid signed certificate. This simply locks down access to the container one layer further.

Once you have ensured that communications to and from your app are safe, the next step is to protect data that you store offline. In the Link system, this encryption is implemented transparently without involvement from your app. When the app runs outside of the Link system, encryption must be implemented in the app layer.

- Encrypt all sensitive data that is stored on the device via the Link SDK's offline storage capabilities. While we do not yet directly integrate this library, we highly recommend CryptoJS (http://code.google.com/p/crypto-js/) for JavaScript based encryption. Use a password-based encryption key to protect the main encryption keys used to protect sensitive data. This level of indirection allows users to change passwords without decrypting and re-encrypting all data.
- Use AES-256 for all encryption. AES-256 is not the only good cryptography algorithm out there, but it is well understood, well tested, well documented, and easy to adopt via CryptoJS. There are no good reasons NOT to use AES-256.
- Follow encryption best practices – including, in particular, using a distinct Initialization Vector for each individual piece of data that you encrypt.

Finally, make sure that your app code itself is secure. The two most important things that you can do to ensure that your app code is secure are:

- Use the Link SDK's PersistenceJS-based API for persistent storage. This framework always uses prepared queries for direct database manipulation, and it prevents programmers from needing to generate SQL queries directly from strings. Constructing SQL queries from strings is almost always a mistake that will lead to SQL injection vulnerabilities. While SQL injection

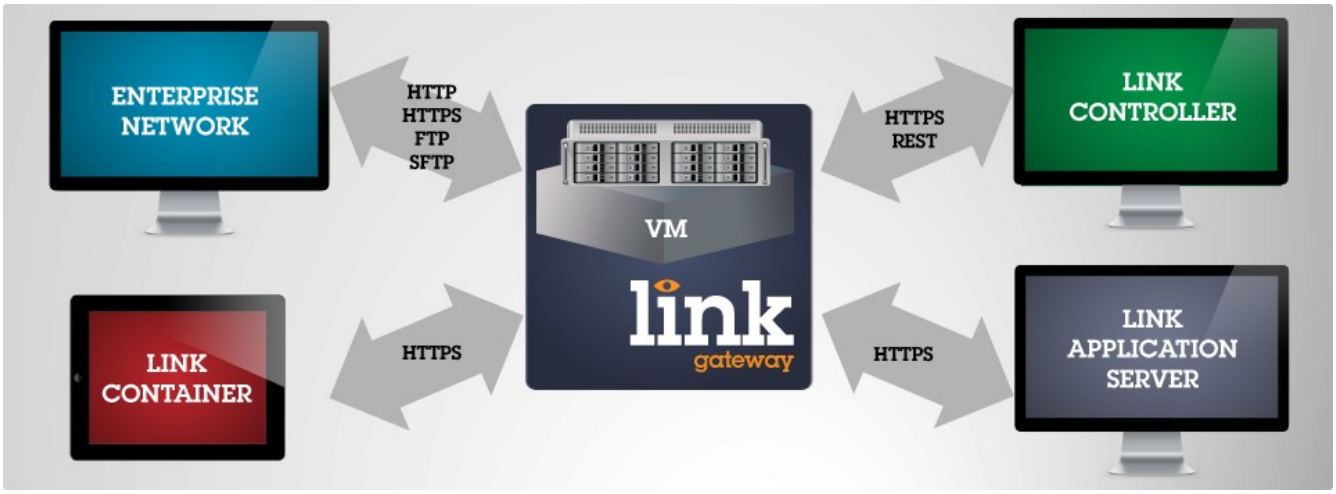
against a client database is nowhere near as devastating as an attack against a database server with multiple users' data, it still could corrupt a user's local database in such a way that would either render the app unusable or might cause the app to misbehave. The exact effects depend on the application, but you are always safer avoiding the issue entirely.

- Protect against XSS attacks by properly escaping text before you append it into the DOM. For a good discussion of how to do so, consult this blog post: http://rbalajiprasad.blogspot.com/2012/09/xss-attack-prevent-using-jquery-encoder.html. While the risks of XSS against an enterprise app running behind the corporate firewall are nowhere near what they are in a consumer facing app that allows attackers to input data into the app, it would still be a grave mistake to ignore XSS. For example, protecting against insider attacks that might expose data that is authorized for a privileged group of users to non-privileged users is extremely important. XSS attacks that inject arbitrary markup into your app can retrieve data stored by the app or sent to the app and send it anywhere within the enterprise network. The Link system helps further mitigate this issue by using a reverse proxy to restrict traffic that is destined for the enterprise network, however, a well crafted XSS attack could work within the proxy's limitations to great effect.

Following these security practices is a huge leap forward in protecting HTML5 apps built with the Link SDK. If security is an important consideration to your organization, then consider adopting the Link system to deliver apps. Some of the key features offered by the integration with the Link system are outlined below. For more information about Link, please visit http://www.mobilehelix.com.

Integration with the Link System

What is Link?



The Mobile Helix Link system is comprised of 4 components:

The Link Gateway, which is a reverse proxy that is the central entry point to the enterprise network. The Gateway controls and optimizes traffic between the external, public Internet and the private, corporate Intranet. The Gateway is deployed in the DMZ in an on-premises deployment, and a large-

scale deployment will have many gateways for redundancy and load balancing.

The Link Controller is the system management console. The Controller manages all users, all devices, and all apps available on a mobile device via the Link system. The Controller integrates with Active Directory (AD) or LDAP to import users and groups, and the Controller manages authentication via AD, LDAP, or any SAML-based single sign-on solution. The Controller uses a role-based security and policy architecture to enable IT administrators to set flexible device, applications, and data policies from a central console. The Controller also monitors the Link system and collects audit data and analytics in a central database.

The Link Application Server hosts Mobile Helix's pure HTML5 apps. Each app is built using the Link HTML5 SDK and is hosted in a standard Java EE container. Examples of Link apps include Link Content Share and Link E-mail. Enterprises can host their own HTML5 apps built with or without the Link HTML5 SDK and legacy web applications available for mobile access using existing web server, application server, or portal infrastructure. The Link Application Servers are firewall protected and integrate with standard HTTPS load balancers.

The Link Container runs on mobile devices and ensures secure access to corporate assets by encrypting data at rest and in motion. The Container includes browser technology supporting critical HTML5 features including offline storage and video. In addition, the Container is enhanced with Apache Cordova (formerly PhoneGap) to enable browser access to device features that are not available across platforms via the HTML5 standard.

Link may be deployed fully on-premises or in a hybrid deployment with selected components in the Cloud. In a cloud deployment, the Link cloud can either integrate with existing VPN infrastructure to access the corporate network or it can use a relay architecture to transmit encrypted data via a persistent, outbound connection from the corporate network to the cloud for routing to the mobile device.

Does the Link SDK require the Link System?

Quick answer: no. As described at the very beginning of this document, the Link SDK is fully standards-based and open. Apps built using the Link SDK can be delivered via standard web infrastructure, and they can run in both desktop browsers and mobile browsers. Apps that choose to use Apache Cordova can be run in the open source distribution of Cordova available from the Apache Software Foundation.

The Link system transparently wraps HTML5 apps via the Link Container. The Link Container is designed to integrate with the Link Controller for authentication and the Link Gateway for communication to and from the enterprise network. In addition, the Container includes a cryptography and data management layer that transparently manages encryption of data in motion and at rest and data retention policies specified in the Link Controller. These features do not require any changes to the app – the only requirement is that the app runs inside the Link Container rather than running in an open source container or in a standard browser.

Benefits of the Link System

The Link system provides a number of important benefits and features to HTML5 apps built with the Link SDK and delivered via the Link system. These benefits include:

- Performance acceleration of HTTPS traffic to and from the Link Container
- Authentication, authorization, and encryption to protect apps and data
- Data retention policies specified in the Link Controller and applied transparently to all apps running in the Link Container
- The Link web services for accessing SharePoint, Exchange, and Windows file shares

While these features are not directly part of the Link SDK, it is worth understanding each piece as they are important when building apps that are designed to run in the Link system.

Performance Acceleration in Link

The Link Gateway implements several features to accelerate the performance of HTTP/HTTPS traffic and, hence, the responsiveness of mobile apps deployed via Link. These optimizations include compression, resource pre-fetching, conditional HTTP requests, and redirect handling. We describe the latter three techniques here.

To minimize the number of roundtrips from the device back to the enterprise network, the Link Gateway scans HTML markup on the fly to identify external references to CSS, JavaScript, and images. These resources are then fetched and packaged with the HTML markup, compressed into a single optimized packaged, and sent to the device. After the first load of an app, the Link Container will cache each component (JavaScript files, CSS files, and images) separately. The Gateway maintains a symmetric cache of data signatures that are retained on the device. On the next load, if the resources included by the app have not changed they are not re-sent to the device.

To further optimize the end-to-end performance of app loads, the Link Gateway maintains a cache of E-Tags or last-modified dates for each cache-able resource retrieved on behalf of a particular user. Before retrieving an object (e.g., a JavaScript file) and checking if its signature matches a signature that is available in the Link Container, the Gateway sends a conditional HTTP request to the originating server. If the object has not changed, the Gateway will thereby avoid downloading the same object again from the server hosting the app.

Finally, the Link Gateway further optimizes HTTP/HTTPS communication by transparently handling redirects in the Gateway. If an app uses a redirect for any reason, rather than incurring the cost of a roundtrip to and from the device to execute the redirect, the Gateway will intercept the redirect and retrieve the referenced object.

Security

The Link system is designed to transparently secure HTML5 apps, freeing app developers from the responsibility of handling most aspects of security in their apps. These features of the Link system include:

- The Link Controller and the Link Container ensure that only authenticated users can access HTML5 apps running inside the Link Container. Link's authorization features allow Link administrators to determine which apps are available to which users and to set appropriate session management policies based on a user's role, a user's device, a user's location, and whether the app is being accessed online or offline. On the back-end, the Link Gateway

integrates with enterprise identity systems, including Active Directory, LDAP, or any SAML-compliant identity provider, to verify a user's identity.

- All communication from the Link container to the Link Gateway is secured via HTTPS. The container validates the authenticity of the Gateway using an enterprise-specific certificate, such that the container can only establish connections to gateways managed by your company. The Link Gateway uses secure session management to ensure that only authenticated users can access resources hosted in the enterprise network.
- All data stored by the Link Container is secured via AES-256. This includes app-specific data stored via the WebSQL standard and, by extension, the PersistenceJS-based data storage API built into the Link HTML5 SDK.
- The Link Controller allows Link administrators to define data retention policies that are encrypted and stored on the device. These policies are applied transparently by the Link Container to ensure that sensitive data is appropriately managed on the device.
- All communications to the enterprise network are monitored in case of any potential breach. This audit log is stored by the Link Controller and can be downloaded via web services as needed.

The Link system implements most of the best practices for security outlined previously in this chapter. However, it does not fully free developers from being mindful of SQL injection and cross-site scripting. The Link SDK mitigates the risk of SQL injection as long as developers work within the design of the SDK. The restrictions placed on communications by the Link Gateway and the Link Container allow Link Administrators to significantly decrease the risk of an effective XSS attack, but it is still essential that developers write code that is robust to XSS by properly escaping any markup that is generated including user input.

Data Retention Policies

Data retention policies in the Link system particularly refer to the lifetime of data on the device and whether that data is available online, offline, or both. These policies are specified in the Link Controller and documented in the Controller documentation. Policies are enforced transparently without placing any requirements on app developers. The most important point for developers to keep in mind is that an app should not assume that offline data persists indefinitely. At any point that data may, due to policy, be deleted from the device. In this case, the app will behave as if it were installed again for the first time.

Link Web Services

Link web services provide a number of RESTful services for accessing common enterprise systems. These systems include:

- Email, contacts, calendar, tasks, and notes stores in Microsoft Exchange
- Access, manipulate (i.e. convert from Microsoft formats to PDF; encrypt via Microsoft's password-based encryption standards), and store documents stored in Microsoft SharePoint
- Access, edit, and create list items stored in SharePoint lists

- Access, manipulate, and store documents and files stored in CIFS file shares

These web services are documented separately.

Summary

An effective HTML5 app combines an intuitive user experience built with the Link HTML5 SDK with an infrastructure designed to optimize performance of the app and ensure the security of sensitive data. Whether this infrastructure is built in house or integrated by incorporating the Link system into your enterprise infrastructure, the supporting infrastructure is as important to an app's success as the app itself. Apps built with the appropriate best practices for performance and security deliver a fantastic user experience without compromising the safety of sensitive corporate data.