markit™

Open Financial Framework
DRAFT

July 2012
version 0.7

Introduction to the Open Financial Framework

The Open Financial Framework, designed and developed by Markit, is an open standard for a web application framework that is specifically designed for the financial markets. This specification enables multiple information providers to entitle both apps and content to be combined in a seamless experience.

The Problem

The Problem We Are Trying to Solve

The reasons for developing the open standard for a financial app ecosystem are simple: one does not exist today and Markit feels there is a great need. Additionally, much of the installed base of financial software is run on the desktop, which means that even if there is no physical media, there is an installation process that is difficult for users to do. By design, this Open Financial Framework implies a one-step installation process, direct from the Internet, with no ”installed software” decisions to be made by the user. Further, the financial information industry has spent most of its history bundling content and tools into packages which include far more than most users need. Perhaps the greatest benefit of the Open Financial Framework is that – through its *Store* component – it will allow users to buy only the content or tools they need. Said plainly: If John can’t get what he wants from Microsoft, he’s going to buy it from Apple. And John wants the best content and the best tools so he’ll buy apps from Apple.

The primary goal of the Open Financial Framework standard and this specification is to provide App developers the resources they need to create apps. Developers who adhere to this standard will make it possible for multiple apps, developed independently by different organizations, to function together creating a seamless and integrated “desktop” experience.

Note:
The term “App”, in popular web parlance, is a module or widget or component. In the Open Financial Framework, Apps are small web pages and consist of HTML, CSS, JavaScript and, of course, data. This specification furnishes App developers with JavaScript code which – among other things – not only allows their Apps to run on the desktop but also provides APIs for communication between the desktop and nearby Apps.

As an extension, these main goals below form the underpinnings of the Open Financial Framework.

Capabilities

- Apps From Multiple Providers Displayed at the Same Time
    - A financial app ecosystem needs to display apps (or modules of data) from multiple information providers at the same time. In order to do this, there needs to be a standard for context passing from app to app, and from the desktop to all apps. As part of the context passed to Apps, App developers should expect to receive information about how their App will be displayed (width, height, etc.).
- Separation of Apps from Content
    - Within Apple’s iTunes Store, some types of content can be used by multiple apps at the same time such as iPhoto or iPod. Within the Open Financial Framework, Apps need to be entitled with content like streaming Level II quotes, or able to handle multiple sources of entitled content, like a news or research aggregation app.
- An App Store that can handle several types of entitlements
    - The App Store within the Open Financial Framework will need to handle more than just individually purchased Apps or Content (known as “entitlements”). In addition to handling cash transactions, the App Store will need to be able to handle entitlements granted to them by a brokerage firm, by a Yahoo! advertiser, by the App developer, or by a content provider. This Framework would make it so that a

user's purchases and entitlements can move to any desktop adhering to this standard, regardless of who is hosting the desktop.

Who Loses & Who Wins

App developers will focus their time, talents, and resources on creating apps for the App community that they believe will give them the largest return. By making this standard, the desktop providers guarantee that no single desktop provider will win the race to become the most attractive marketplace for App developers. The desktop providers also guarantee that they won't lose. Many desktop or platform providers, like Markit, will also be App developers. In Markit's case, what it doesn't get by trying to win the race for the most compelling App community, it would win by having a larger market for its own apps.

Building the Framework

If we have the goal of the Framework being successful and accessible, then we have only one choice in the technology we use: it has to be HTML5.

Open Standard	HTML5 is an open standard. It is free to use. There are thousands of organizations that are developing tools and libraries using HTML5 and HTML5-related technologies.
Cross-Platform	Unlike Apple's iOS, HTML5 allows developers to write code that works on both mobile devices, and desktops.
Manufacturer Independent	HTML5 enables Apps to be written for any device, including Apple, Android and Windows Phone devices* in addition to PCs.
Offline Capability	When configured and when used in a supporting web browser, HTML5 related technologies enable an App to be run even when the device or computer is not connected to the Internet. This allows the user, for example, to read market research on a submarine or airplane.

*Support for HTML5 related technologies on *all* mobile devices are limited. The future mobile landscape is a three horse race between Apple, Google and Microsoft. Apple's iPhone and iPad are leaders in HTML5 support, while Samsung's newest devices built for Google's Android OS are a close second. Microsoft's new mobile devices, including the Surface tablet, have feature support, too.

Using the Open Financial Framework to Solve the Problem

The Components of the Framework

1. The Container
2. The App
3. The App Store
4. The Developer Site

The Container

The Container is most simply described as the user interface and the location where all Apps reside. More specifically, the Container is a web page which is "aware" of its contents (the Apps) and plays the role of a traffic cop managing Context passing between Apps. Further, the Container can have any variation of intelligence on a wide spectrum.

Markit's intention in standardizing the Container concept is to keep the Container as "thin" as possible (not unlike a traditional thin client).

The Grid

Apps will be loaded onto the Container and confirm to the desktop's Grid. The Grid is a HTML document based on a 16-column fluid layout. Users can resize their browser windows and therefore their desktops to any dimension their devices' display supports.

There is only one limitation when designing in App: Apps should be designed to be no narrower than 320 pixels in width (the screen width of the most popular smartphones). The Container itself will not support a resolution narrower than 320 pixels on either desktop or mobile displays. (The optimal Container resolution is 1280 pixels and 'full screen' is one better.) Except in the case of when an iframe is used to load an App on the Container, Apps can determine their own height on the desktop and automatically "stretch" until all content in the App is visible to the user.

Interactions

Apps have Context. Context can be shared between other Apps (children) and the Container (parent). The Container is responsible for managing the Context passing; see "Context" for more information.

As for interactions and user experience, the design guidelines and App API will contain information on how to provide users with a consistent presentation layer. When it comes to in-app menus or settings panes, this is crucial.

The App

The App is a web page or microsite, written by an App developer that includes its own (entitled) data. Further, an App can be purchased (or entitled) from the App Store, and is displayed on the desktop or "Container". The App developer is the person or company that designs, develops, and hosts the app.

Simplistically, an App is one of two things:

Presentation App	A presentation App is an HTML document and contains JavaScript, cascading style sheets, images and other objects. It displays data to users in a manner determined by the App designer and must adhere to this specification.
Data App	A data app is a content feed available in industry-standard formats including JSON, JSONP, RSS or App-defined XML.

App Context

Regardless of type of app, a presentation or data App has “context” – that is to say an App “knows” about who is viewing it and its content. It is aware of a specific user’s data entitlements as well as information about the user (name, email, company, etc). Additionally, the App is capable of sharing Context with the Container and nearby Apps. This means if Susan wants to create a ticker-focused workspace so she can watch her Apple stock decline in value during Apple’s next product event, the Container can send “symbol context” to Apps and the Apps will be smart enough to refresh focusing on AAPL.

Creating a Common Look and Feel

In designing an App, app designers and developers should follow the Responsive Web Design Methodology. At a high level, this allows Apps to be flexible on both desktop and mobile workspaces. Wikipedia defines Responsive Web Design (RWD) as “a web site [that] is crafted to use CSS3 media queries, an extension of the @media rule, with fluid proportion-based grids, to adapt the layout to the viewing environment, and probably also use flexible images. As a result, users across a broad range of devices and browsers will have access to a single source of content, laid out so as to be easy to read and navigate with a minimum of resizing, panning, and scrolling.”

Further, when considering App design, “’Mobile First’ and ‘Progressive Enhancement/Unobtrusive JavaScript’ are related concepts that predated RWD: browsers of basic mobile phones do not understand media queries or JavaScript, and it is wise to create a basic web site then enhance it for smart phones and PCs — rather than attempt "graceful degradation" to try to degrade a complex, image-heavy site to work on the most basic mobile phones.”

Using the Open Financial Framework’s upcoming design guidelines and App API, App Designers and Developers can take advantage of these available resources to develop Apps on their own schedules. The design guidelines will provide a common theme and offer a baseline for consistency between all Apps on the Container.

The App API

Markit is developing App APIs, to provide App design and development teams (as well as Container providers) the tools, concepts and best practices needed to produce high-quality Apps.

In releasing the open App API, the goal is to provide:

1. Technical documentation on how to build and deploy Apps, how Apps are consumed by Containers, how to build and host Containers, and more.
2. Design standards and recommendations for App and Container designers.
3. Articles highlighting best practices and concepts for App and Container developers
4. Getting-started references and sample code

The App Store

The App Store is the place where a user selects (or purchases) either a presentation or data App. Customers will be able to buy an app by using an electronic payment mechanism, like a credit card, but they will also be able to charge their company, or be entitled by a vendor or through another business relationship. Since the App Store is where it all begins, the term is usually used as a substitute for “App Community”.

The Developer Center

The Open Financial Framework specification will be published on a website featuring Framework documentation, App developer resources and sample Apps. This site, known as the Developer Center, will be a web site where all of the resources for App and Container developers can be found. This includes the App APIs which will provide App design and development teams (as well as Container providers) the tools, concepts and best practices needed to produce high-quality Apps.

In addition, a key part of the Developer Center will be the ability for App developers to manage their Apps, register their Apps with Container Providers, issue App updates, and more.

Creating the Open Financial Framework

Phased Approach

Markit is employing a phased approach in the development, testing and release of the Open Financial Framework. The following identifies major milestones targeted for each phase (in no particular order within each phase).

Phase	Objective
1	Publish Open Financial Framework version 1.0 specification on developer resource website hosted by Markit On Demand.
1	Create sample Container for demonstration purposes on developer resource website.
1	Create various sample Apps for demonstration purposes as part of the sample Container on developer resource website.
1	Create and publish JavaScript App APIs on developer resource website. Includes Context sharing and event emitting.
1	Create and publish JavaScript Container APIs on developer resource website. Includes Context sharing and event emitting.
1	Create and publish web services for App developer consumption including XRef, Quote and Timeseries. Includes JSON, JSONP, XML responses.
1	Open-source Markit On Demand-developed ASP.NET C# library for App development. *Editor's note: Open source Element, too?* *Technical note: This C# library is primarily a view engine which smartly looks into your .NET application and stitches together models and views into a neatly packaged serialized object. The content type of the response matches the requested format and self-documents along the way.*
2	Publish Open Financial Framework version 1.1 specification.
2	Create and publish standard single sign on (SSO) mechanism for Container-to-App authentication.
2	Open-source Markit On Demand-developed ASP.NET C# library for Container development. *Editor's note: This must be WIT-less.* *Technical note: This C# library is primarily a view engine with embedded libraries required by the Open Financial Framework.*
2	Create and publish web services for App developer consumption including Directory.
3	Publish Open Financial Framework version 1.2 specification.
3	Launch Open Financial Framework App Store
4	Publish Open Financial Framework version 1.3 specification.
4	Create and publish Container APIs for cross-Container communication. For example, if both Goldman Sachs and UBS host Containers and Steven wants to use his FT.com App in each only after purchasing the FT.com subscription on Goldman's App Store. Goldman's Container now has to be able to tell UBS' Container (or vice versa) that Steven is entitled to the FT.com App.

Disclaimer and Commitment

This specification is a working document currently in draft form. To demonstrate Markit's commitment to perfecting this Framework, Markit development teams are already following most of the technical implementations described in this standard. In an effort to arrive at a completed *open* standard, Markit teams are actively helping to effectively round out this specification.

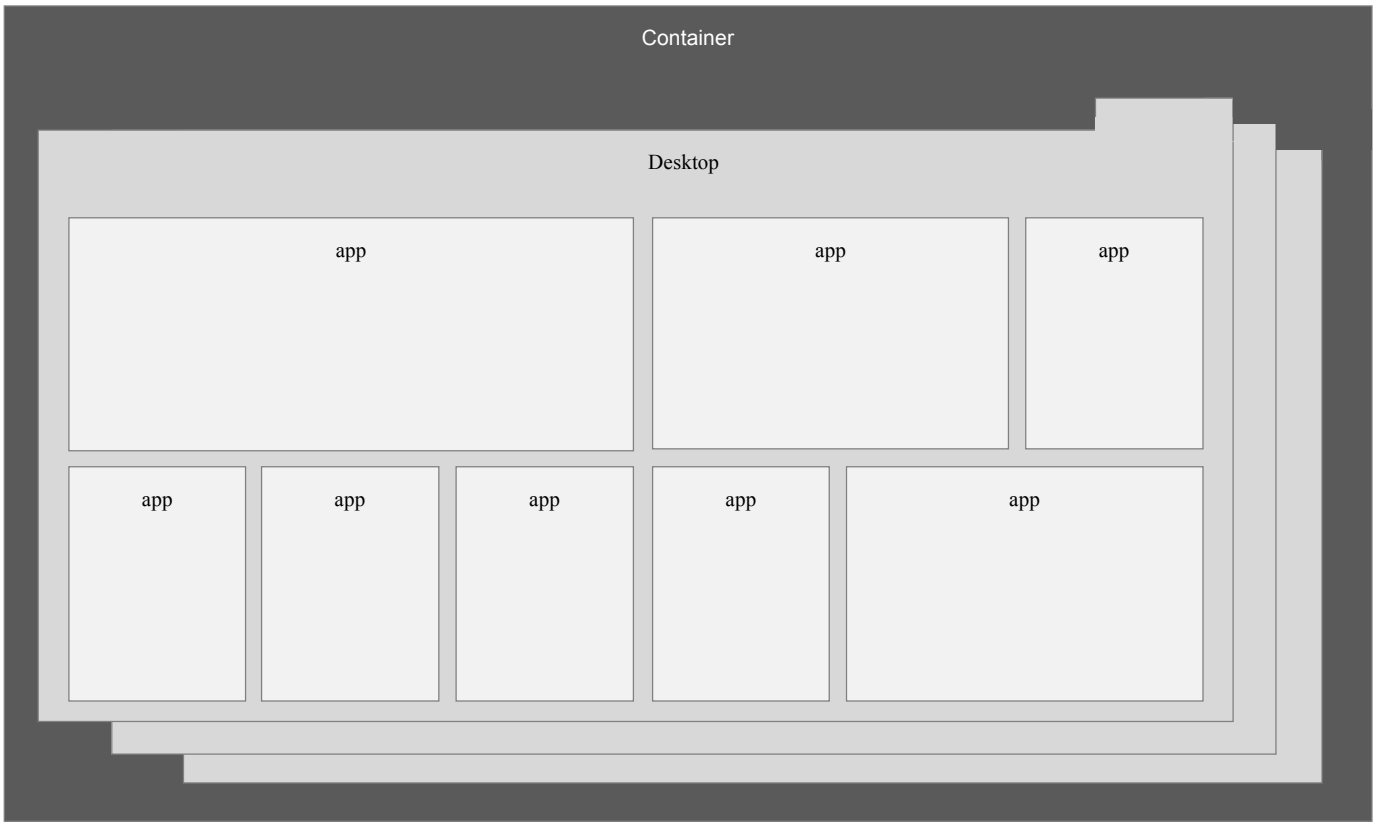
Release Date

Currently, no release date has been set for version 1.0 however Markit is targeting fall 2012.

Together At Last: Containers and Apps

Overview

The diagram below depicts the world's simplest Container – look at all those grey boxes! The purpose of this picture is to demonstrate an App's relationship with a Desktop and in turn the Container. What is not shown here is the user's browser chrome.



As you can see, the Container is nothing without Apps. Similarly, the Apps have no home without the Container. It is imperative they work together and appear *seamless* to users.

"Okay," you say, "who hosts all these grey boxes?" *Great question.*

Hosting

Since the Framework is web-based and it is a primary requirement of this Framework to simultaneously support multiple Apps from different providers, the following are truths:

- Anyone can technically host a Container provided they are willing to develop the infrastructure capable of supporting an app ecosystem which includes authentication, entitlements, the app store, cross-container communication (targeting version 1.3 spec), etc. See *How to Develop a Container* for details.
- Similarly, anyone can host an App. By definition, an App is simply a web page or web site which has a Container-accessible domain name.

Container Providers and App Developers

In designing the Open Financial Framework, an important consideration was maintaining a separation between Container Providers and App Developers. Therefore, after App developers register their Apps with Container Providers,

Containers automatically consume Apps using the Container APIs. App Developers have as much or as little information (as App Context allows) about where their App exists at any given moment in time.

Usage Reporting

App developers are encouraged to leverage their own or third party tools for capturing usage metrics or other MIS. App developers will not have access to Container Provider-owned usage analytics unless it is pre-agreed upon between the two parties.

Developing an App Using the Open Financial Framework

Overview

An App is a web page or microsite, written by an App developer that includes its own (entitled) data. Further, an App can be purchased (or entitled) from the App Store, and is displayed on the desktop or "Container". The App developer is the person or company that designs, develops, and hosts the app.

Simplistically, an App is one of two things:

Presentation App A presentation App is an HTML document and contains JavaScript, CSS, images and other objects. It displays data to users in a manner determined by the App designer and must adhere to this specification.

Data App A data app is a content feed available in industry-standard formats including JSON, JSONP, RSS or App developer-designed XML.

Relationship Between Container and App

Refer to "*Together At Last: Containers and Apps*".

Development Requirements

Each App must have a digital signature or unique identifier in the format of a GUID. This allows App Providers to register their App with the App Store and in turn the App Store can provide reporting metrics on App usage, purchases, etc. Additionally, having a unique ID allows the Container to interact with specific modules on the desktop using Context or in more simple cases using JavaScript's document.getElementById() method.

Beyond a unique ID, Apps have other characteristics that define them within the Framework. The following are required attributes App developers must include in their Apps. The following data points should be represented within the presentation or data App:

AppID – *long (guid)*
Name - *string*
Title - *string*
Context – *object*
AcceptsContext – *bool*
DeveloperEmail - *string*
DeveloperURL – *string*
DeveloperCompanyName - *string*

Since Apps are comprised of mainly HTML, JavaScript and CSS, App development can be as complex as the App developer wishes with one significant limitation: an App cannot be allowed to negatively impact other Apps on the desktop. To prevent accidental impact, Apps are developed inside of JavaScript closure which means an App will not have any public methods and therefore is a closed cell.

Designing the App to Look Integrated with the Container

Design is an important first step in creating a new App. Using the Open Financial Framework's upcoming design guidelines and App API, App Designers and Developers can take advantage of these available resources to develop Apps on their own schedules. The design guidelines will provide a common theme and offer a baseline for consistency between all Apps on the Container.

There is customization available and it will be imperative for App developers to follow Container-provided guidelines for consistency on a case-by-case basis. The 'common theme' provided by this Framework is not a template per se but rather a mechanism to facilitate faster development between numerous App developers.

Loading an App on the Container

Each App must "load" itself by calling a JavaScript method available in the SDK which will be hosted by the Container Provider. The arguments sent to "loadApp" are the App characteristics and any additional name/value data App Developers wish to include as part of their App's Context.

```
Container.loadApp({
	AppId: "1234-5678-9101-1121-3141-5161",
	Name: "Acme Financials",
	Title: "Acme Financials Module",
	Context: {
		Symbol: "MSFT",
		Name: "value",
		List: [1,2,3,4]
	},
	AcceptsContext: true,
	DeveloperEmail: "dev@domain.com",
	DeveloperURL: "http://domain.com",
	DeveloperCompanyName: "Acme, Inc"
});
```

Context

Each Container Provider shall be responsible for hosting the Container JavaScript API. This JavaScript framework provides a consistent means for all App Developers to load their Apps on any Container.

While Apps can have Context themselves, the responsibility for managing Context switching or Context passing falls on the Container. The Container assumes the role of a traffic cop – managing what data goes where. Using an Event Emitter, the Container can "listen" for events sent by Apps on configurable intervals and likewise Apps can listen for events sent by the Container.

This is a sample of a Container sending Context to Apps. Firstly, the Container fires a "ContextUpdate" event.

```
Container.on("ContextUpdate", { Symbol: "MSFT" }, function(ev,callback){
	console.log("Context updated!");
});
```

Apps are responsible for listening to the broadcasted "ContextUpdate" event. App developers can bind custom event handlers based on the emitted event. Using jQuery, a sample to refresh an app with a new symbol:

```
$(".myApp").bind("ContextUpdate", function(ev,data){
	myApp.refresh(data.Symbol);
});
```

Likewise, Apps can send Context to the Container.

```
App.on("ContextUpdateToContainer", { FullScreen: true }, function(ev, callback){
	console.log("Context sent to Container!");
});
```

The Container can then listen for App-emitted events.

```
$("#Container").bind("ContextUpdateToContainer", function(ev,data){
	Container.showFullScreen();
});
```

Apps can also pass Context between Apps. If there are two or more Apps on a Container with similar Context and the ability to receive messages (yes, this is opt-out), Apps can communicate with each other.

For example, on your Container you have “App 1” which is a watch list app alongside “App 2” which is a snap quote app. Embedded within the Content in App 2 could be a button labeled “send to watch list”. Pressing that button would transmit the symbol of the stock currently being viewed in App 1 across to App 2 to be added to the watch list. Achieving that looks programmatically like this:

First, find the App you want to find the apps to which to send the Context to:

```
var $apps = Container.getApps(bool canAcceptContext(true));
```

Secondly, emit an event from App 1:

```
$apps.find(“App1”).on(
	“ContextToApp”,
	{ Symbol”: “MSFT”, Action”: “ADD” },
	function(ev,data){
		myApp.refresh(data);
	}
);
```

And finally listen for that event within App 2:

```
$apps.find(“App2”).bind(“ContextToApp”, function(ev,data){
	console.log(“Context received in App2!”);
});
```

App Formats

Below is a sample request for an App named "Sample" in JSON format.

```
http://www.domain.com/App/Factory/json?params=[{Id:"App1",Width:200,Name:"Value"}]
```

The "`/json`" format specifier in the above request could be safely omitted, as JSON is the default format. The widgets can be returned in a variety of formats as shown in the table below. Inclusion of a widget into an existing site varies based on the format requested.

The following output formats are required of any App:

Name	Description
IFRAME	An inline <script> tag is inserted into the site. When the script is executed, an Iframe is written out to the page. The Iframe output format can be used for complex widgets or widgets that need to be self-contained.
JSON	A JSON object containing the javascript, css, and widgets is inserted into the site via javascript already on the site.
JSONP	Same as JSON but output is wrapped in a Callback function. Used for cross-domain communication and callbacks.
SCRIPT	An inline <Script> tag is inserted into the site. Wen the script is executed, the javascript, css, and html for the widgets is written out to the page.
XML	An XML string containing the javascript, css, and html for the widgets is returned. The XML output format is

	primarily used when bringing widgets into the server-side code before returning to the client.

The JSON response format does not include *type* information. A description of the JSON format can be found at http://json.org. It will not be described in this Spec. A sample JSON response is shown below (with white space and line breaks added for legibility):

```
{
	"InlineScripts" : [],
	"Scripts" : [],
	"Styles" : [],
	"Apps" : [
		{
			"Html" : "\u003cdiv\u003eHello World!\u003c/div\u003e",
			"Data" :
				{
					"Symbol" : "AAPL"
				},
			"Id" : "Sample",
			"Status" : "SUCCESS"
		}]
}
```

The JSONP response format is similar, with the addition of a callback method name.

```
mySampleCallbackName({
	"InlineScripts" : [],
	"Scripts" : [],
	"Styles" : [],
	"Apps" : [
		{
			"Html" : "\u003cdiv\u003eHello World!\u003c/div\u003e",
			"Data" :
				{
					"Symbol" : "MSFT"
				},
			"Id" : "Sample",
			"Status" : "SUCCESS"
		}]
})
```

An example XML representation of the SampleResponse object is shown below:

```
<?xml version="1.0" encoding="utf-16"?>
<AppResponse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <InlineScripts />
  <Scripts />
  <Styles />
  <Apps>
       <App>
            <Html><![CDATA[<div>Hello World!</div>]]></Html>
            <Data>
              <item key="Symbol" value="MSFT" />
```

```
            </Data>
            <Id>Sample</Id>
            <Status>SUCCESS</Status>
        </App>
    </Apps>
</AppResponse>
```

Hosting an App

Since the Framework is web-based and it is a primary requirement of this Framework to simultaneously support multiple Apps from different providers, the following are truths:

- Anyone can technically host a Container provided they are willing to develop the infrastructure capable of supporting an app ecosystem which includes authentication, entitlements, the app store, cross-container communication (targeting version 1.3 spec), etc. See *How to Develop a Container* for details.
- Similarly, anyone can host an App. By definition, an App is simply a web page or web site which has a Container-accessible domain name.

The App Developer or App Provider needs to host their App on a publicly accessible Internet domain. The App should follow the "REST" model for web services. In simple terms, REST is a formal description of the HTTP protocol. Accessing a "REST" App is merely a matter of making a standard HTTP request to a defined resource.

An App can be accessed by an HTTP GET or HTTP POST request. Each request consists of a URI, and a URL encoded list of parameters in JSON. The URI consists of a hostname and base path, a method name, and an optional format specifier. The parameters are appended to the URI on the querystring.

App Content

App Providers can determine which content they wish to make available within their App. It is recommended that content is focused on financial information; however, there is no limitation as such. Content can range from news to research to multimedia, and content should be presented using Progressive Enhancement development strategies. That is to say multimedia content, for example, should be shown plugin-free (using HTML5 video or audio elements) for capable browsers and fallback to Flash-based players for browsers that do not yet support HTML5 related technologies.

If App Providers embed URLs back to their own websites, URLs must be opened in a new window as to not interrupt the experience of someone using the Workspace. If authentication is required on App Providers' site, this can be achieved with pass-through authentication using encrypted URLs as discussed in the Authentication API section of this specification.

Single Sign-On

Providers participating in the Markit App Framework must modify their web sites' authentication mechanism to accept a customer's authentication from a Container Provider without requiring the user to retype their Username and Password. With an authentication methodology in place between the Container Provider and App Provider, when a customer authenticates to a Provider's App, the authentication credentials are passed to the Container Provider. The authentication methodology is also suitable for the reverse, for authenticating a customer requesting content from an App Provider after selecting content from the App within the Container. Authentication information will be passed between App Providers and Container Providers in the form of encrypted URLs.

Entitlements

User Entitlements are the responsibility of the App developer. Many apps will need to be decoupled from the content that they need. This would include apps like research aggregation, news filtering, streaming market data, etc. In order to enable an App to retrieve data from multiple, entitled, content providers in real-time, there will need to be an explicit and trusted mechanism of passing entitlements information between the Store, the data vendors, and the app developers.

Further details around entitlements will be forthcoming as this specification evolves.

Developing the Container

The Container is most simply described as the user interface and the location where all Apps reside. More specifically, the Container is a web page which is "aware" of its contents (the Apps) and plays the role of a traffic cop managing Context between Apps. Further, the Container can have any variation of intelligence on a wide spectrum.

Markit's intention in standardizing the Container concept is to keep the Container as "thin" as possible not unlike a traditional thin client. Wikipedia describes a "thin client" as "a computer or a computer program which depends heavily on some other computer (its *server*) to fulfill its traditional computational roles. This stands in contrast to the traditional fat client, a computer designed to take on these roles by itself. The exact roles assumed by the server may vary, from providing data persistence to actual information processing on the client's behalf.

"Thin clients occur as components of a broader computer infrastructure, where many clients share their computations with the same server. As such, thin client infrastructures can be viewed as the providing of some computing service via several user-interfaces. This is desirable in contexts where individual fat clients have much more functionality or power than the infrastructure either requires or uses."

Context

Each Container Provider shall be responsible for hosting the Container JavaScript API. This JavaScript framework provides a consistent means for all App Developers to load their Apps on any Container.

While Apps can have context themselves, the responsibility for managing Context switching or Context passing falls on the Container. The Container assumes the role of a traffic cop – managing what data goes where. Using an Event Emitter, the Container can "listen" for events sent by Apps on configurable intervals.

This is a sample of a Container sending Context to Apps.

```
Container.on("SendContext", { Symbol: "MSFT" });
```

Apps on the Container can optionally listen for broadcasted messages or events and respond accordingly.

```
$("myApp").bind("SendContext", function(ev,data){
      myApp.refresh();
});
```

Container developers do not need to worry about how to "receive" Context sent from Apps; this logic is handled as part of the Markit-provided Container API.

How to Develop a Container

A Container is most simply described as a web page; therefore developing Containers can be as simple or complex as the Container developer chooses. Markit is developing a Container API which will manage Context on the workspace. Additionally, it will provide App developers an Application Programming Interface, or API, for advanced Container features such as a drag-and-drop, app management (add new apps, remove existing apps), etc.

The simplest implementation of a Container looks like this:

```
<!DOCTYPE html>
<html>
      <head>
            <title>Markit Container</title>
            <meta name="X-UA-Compatible" content="IE=edge,chrome=1" />
            <meta name="viewport" content="device-width" />
            <script src="http://dev.domain.com/ResourceManager/modernizr.js"></script>
```

```
		<!--[if IE]>
		<script src="http://dev.domain.com/ResourceManager/html5shiv.js"></script>
		<![endif]-->
	</head>
	<body>

		<script src="http://dev.domain.com/Container.js?1.0"></script>
	</body>
</html>
```

In developing an advanced Container, the HTML document's body element would contain additional markup and allow for specific positioning or placement of Apps. Such an example might look like this:

```
<!DOCTYPE html>
<html>
	<head>
		<title>Markit Container</title>
		<meta name="X-UA-Compatible" content="IE=edge,chrome=1" />
		<meta name="viewport" content="device-width" />
		<script src="http://dev.domain.com/ResourceManager/modernizr.js"></script>
		<!--[if IE]>
		<script src="http://dev.domain.com/ResourceManager/html5shiv.js"></script>
		<![endif]-->
	</head>
	<body>
		<header>
			<nav>
				<a href=”/home”>Home</a>
			</nav>
		</header>
		<section>
			<h1>Markets Overview</h1>
			<div id=”destinationApp1”></div>
		</section>
		<footer>
			&copy; 2012 Markit.
		</footer>
		<script src="http://dev.domain.com/Container.js?1.0"></script>
		<script
src='http://app.provider.com/App/Factory/script?params=%5B%7B%22Id%22%3A%22Hello%20World%2
2%2C%22DestinationElement%22%3A%22destinationApp1%22%7D%5D'></script>
	</body>
</html>
```

You may notice the App embedded in the above example has querystring “params”. The decoded params value is:

```
?params=[{"Id":"Hello World","DestinationElement":"destinationApp1"}]
```

This params object only demonstrates an example for a Container. For a full explanation on Params and all their possible values, refer to “How to Develop an App” in this specification.

How to Consume an App

A Container developer only needs to be concerned with developing and hosting a Container. Through the Container and App APIs, Containers will consume Apps automatically. The consumption of an App onto the Container is managed

either by individual users (in the configurable Container) or by a Container administrator (in the non-configurable Container).

Depending on the Container design, users are typically shown a menu of Apps from which to choose as they are customizing their workspace(s). When a user chooses to add an App, the App will "load" itself (per the section titled "*Loading an App on the Container*") on the Container.

Search

Each Container Provider shall be responsible for implementing search functionality. Users should be able to search content specific to either a Content provider itself or the Apps as part of the Container. Taking the example of Context, Containers need to be able to allow users to search for financial content – usually starting with the symbol of an instrument.

Financial users will usually be dealing with lists of issues or individual entities. In order to share Context between apps, there will need to be a protocol for specifying the individual entities, issues, or products, as an individual item, or as a list of similar items. In order to do this, both data providers and app providers will need to use a standard and shared identification system.

Markit's reference data technology, or "XRef" (for symbol cross-reference), can be leveraged for this use. Web services can be made available to both App and Container developers to ensure all parties can (technically) converse about the same entities.

Directory

A directory of financial users would be necessary in order to organize and share app store entitlements. Access to this directory could also be an App.

Markit Directory is a centralized, cloud-based service that creates and manages a "golden record" unique ID for an individual's profile details. The processing of profiles, administration interfaces to managing exceptions, reference data, auditing, security, privacy and extensible APIs are all components of the Directory. The Directory enables portability of identity information across security domains and will be the single source for all industry participants for profile information.

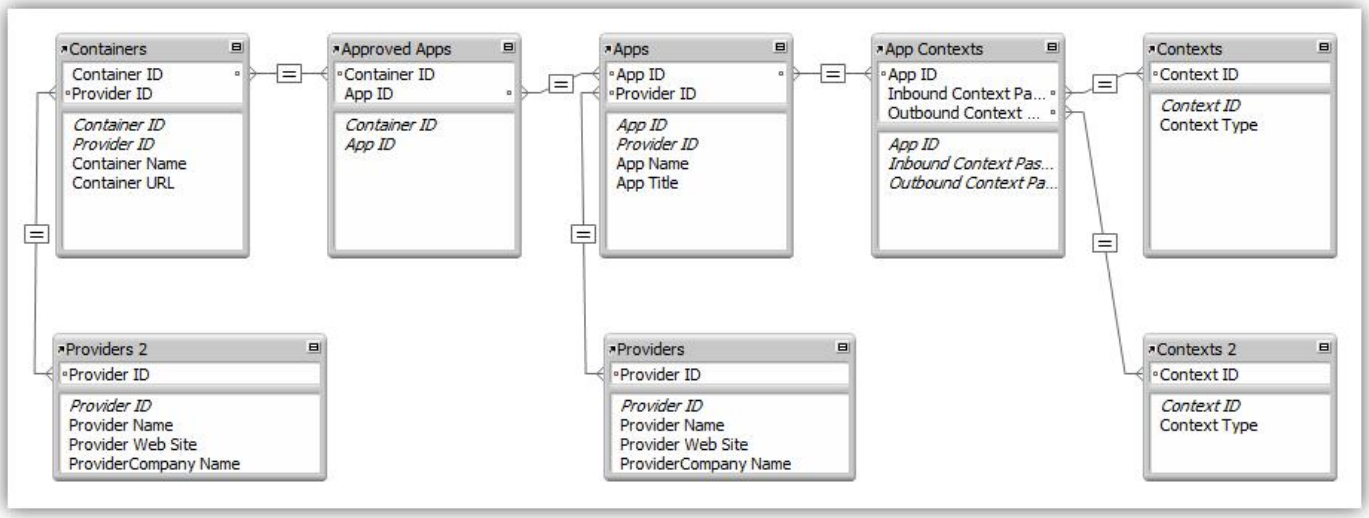
Both App and Container developers could leverage Markit Directory to share user information (as part of Context) between Apps and Containers.

Messaging

Federated Messaging is an app that only Markit can provide by virtue of its exclusive license with NextPlane. Markit's federated messaging App will enable any user, regardless of their firm's instant messaging system, to monitor presence and send message to anyone else, regardless of messaging system.

Container Hosted Data Schema

Below is a proposed high-level database schema for how Apps, App Providers, Containers, Container Providers and Context are managed.



Container Responsibilities

Authentication

Allowing the user into the container

Knowing who the user is.

Storing Entitlements

Passing User Information and Entitlements in the App

Launching An App

Communicating with other Container and Store Developers

Developing the App Store

As part of the Open Financial App Framework specification, there is an App Store component. The App Store is the place where a user selects (or purchases) either a presentation or data App. Customers will be able to buy an app by using an electronic payment mechanism, like a credit card, but they will also be able to charge their company, or be entitled by a vendor or through another business relationship. Since the App Store is where it all begins, the term is usually used as a substitute for “App Community”.

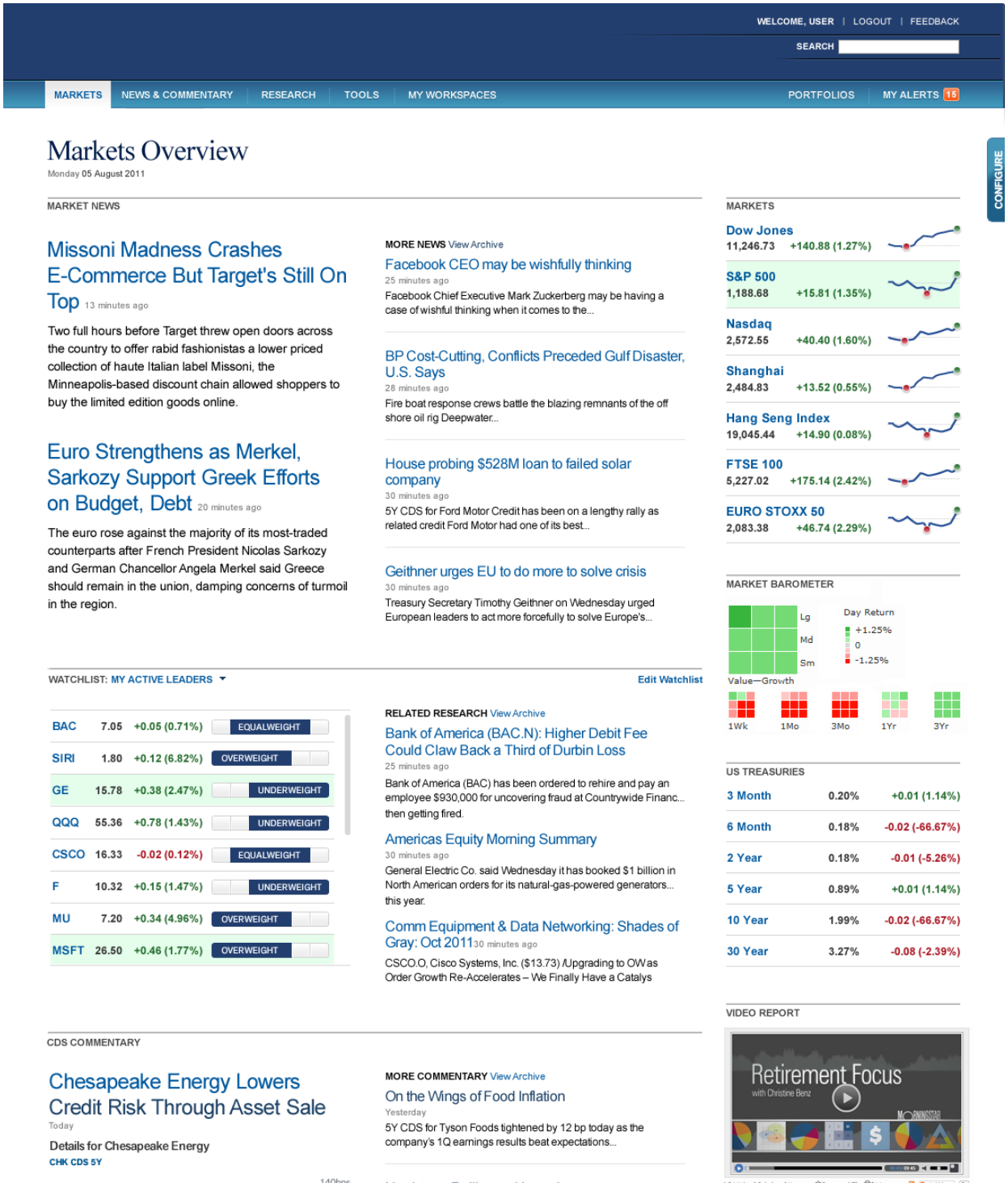
At this point in time, Markit is focusing efforts on developing the App and Container specifications, SDKs, APIs, and examples. Details will be forthcoming as this specification evolves.

Open Framework Developers Working Together

Examples

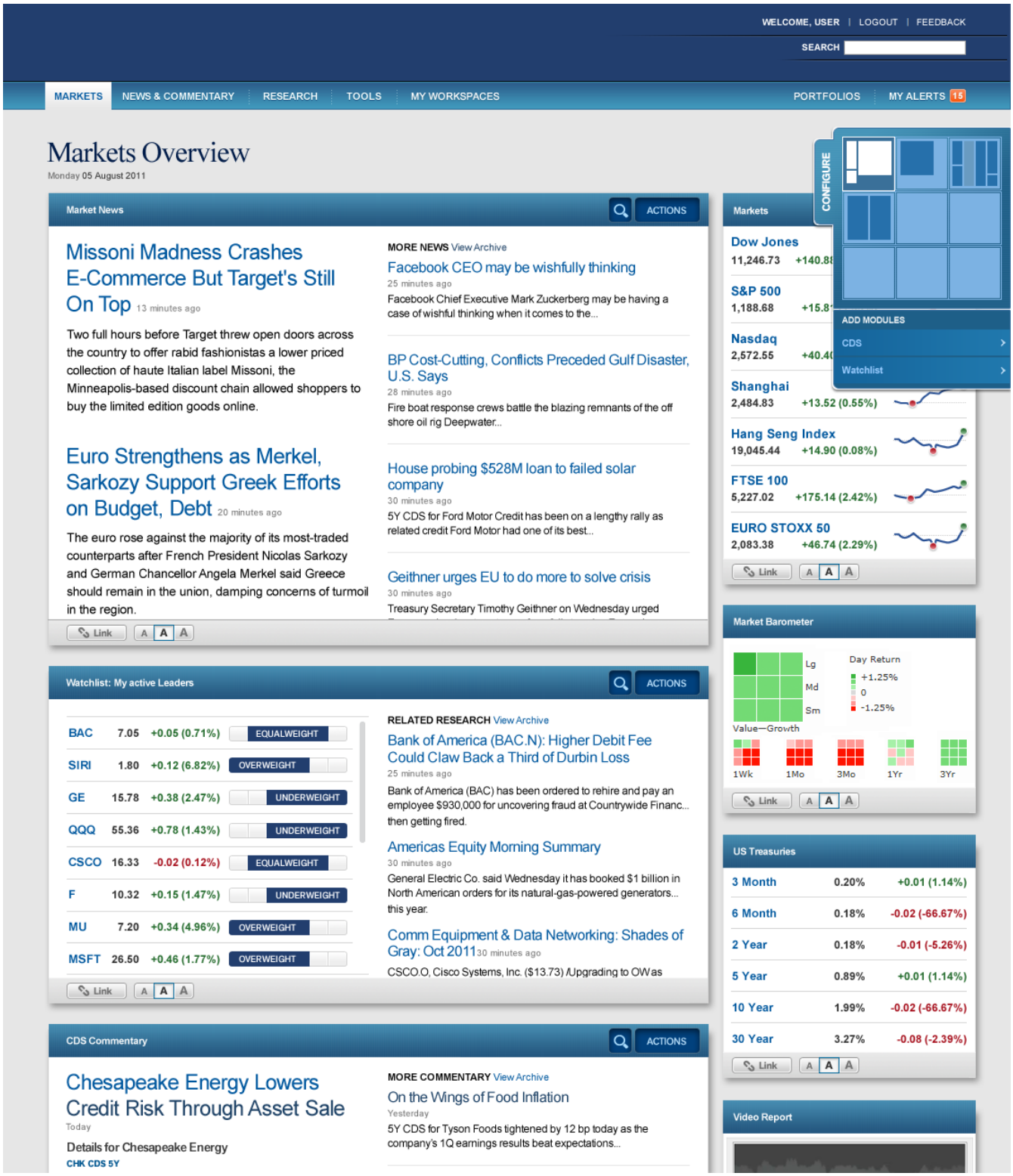
The following are examples of Apps and Containers.

Non-Configurable Markets Overview



Configurable Markets Overview

In this example, each App has “chrome” and a menu of options at the top right of each.



Fully Customizable “Desktop”