How to Build an MPage Component

…In 10 Easy Steps

October 6th 2011

David Stone

Contents

1. Introduction – About this Guide

We'll begin by defining what this "guide" is not. This is not a guide to building an entire MPage. This guide will not take you through the process of creating a whole MPage from scratch. Cerner host classes and education session about this topic and there are many other resources available if that is your goal. It's a good goal, but not what we're going to be discussing. This is also not a guide to creating an entire standard-compliant component "framework". We'll discuss what a framework is to help provide a background, but we won't tell you how to build one. If you're interested in developing a complete framework to host MPages and components you will want to review the component standard in detail as well as the reference frameworks that are provided. Building a framework is a complex programming task that requires advanced JavaScript expertise and ideally an understanding of object-oriented programming concepts.

The goal of this guide is much simpler than either of the two undertakings, but in some ways much more powerful for a beginning MPage developer. Instead of teaching you how to build an MPage from scratch (which is simple in its basics, but complex in best practices and reality), we are focusing on how to add value to the MPages that are already available from Cerner (and others). This guide will teach you how to build a component that can easily be added to an existing MPage. More specifically, we are going to teach you how to build a component that can be added to an MPage that is designed specifically to support components like this.

You may be wondering how many MPage actually support custom components. After-all, if you have already built your own MPage (maybe just a starter page), it's unlikely that your MPage is going to support these components as it is currently built. It's true that there aren't a lot of MPages that currently support these components, but that will be changing quickly. Specifically, Cerner is working to integrate support for the component standard used in this guide. They will be adding support to their "MyPages" solution, which allows a site (or user) to design and configure many MPages from the basic building blocks: MPage Components. In addition, many of the leading MPage development client organizations are internally working to develop MPage frameworks. Many of them are likely to be made available through the App Store or other means. This means that building your own framework is not a necessary undertaking.

This guide is also not laid as a set of rules for building a component. Instead, this book is laid out in the form of how-to exercise intended to illuminate how a user would go about building a component to solve a particular problem or meet a specific need. The guide starts with the basics, a "Hello World" example. It then progresses to more complex examples, such as customizing the data source, or building a collection of components that share data. There are rules to be followed, and they are interspersed within the guide within each example. At

the end of the guide, you will find a more complete list of rules and best-practices that can help guide you to developing more complex components.

There are a few other things that this guide will not do. This guide will not show you how to build a fully working "Medication List" component. We are not focused on producing specific components that you will actually want to use at your site. The underlying tools are presented in this book, but it is up to you (the component developer) to use your creativity and design skills to build the perfect component for your needs. This guide is also not intended to help guide the user in how to develop a CCL script or where to find the data. It also will not cover HTML or CSS markup (we assume you already know most of this). This is not a guide to JavaScript programming (the primary technology for building MPage Components). Luckily, basic components require very little JavaScript knowledge and if you wish, it can be avoided almost altogether (and we'll show you how). You will definitely benefit from a strong knowledge of JavaScript. After all, we are talking about programming and development, so at some level, having the tools in your belt will only make you more powerful.

Lastly, this is a NOT a guide about how to USE MPage components. This will differ from framework to framework. In Cerner's MyPages solution, you will use a component by first importing it to the list of available components and then by adding the component to a particular page using Bedrock. In another sites framework implementation, you will add the component to the page using a little bit of programming and ExtJS tools. In each case, the framework developer will include simple instruction on how to use their framework to add components to a page. Because the component interface is standardized for developers, this will always be fairly similar and easy.

2. Introduction – What is a Component?

So what exactly are MPage Components? There are lots of good analogies. One way to think of a component is kind of like a Lego building block. You can take a "class" of Legos (i.e., black 6x2 blocks) and reuse them along with other building blocks (i.e., gray castle walls) to construct an entire building. In this an analogy, you cannot only use the black blocks, but you can reuse them in many solutions (you usually have more than one at your disposal). You can also use them different ways, turning them lengthwise, or sometime even on their side. This is a bit of an over simplification, but is not too far from reality. MPage Components usually follow a few rules as to how they are used that are more specific than Lego blocks that can be combined any way you like. Unlikely Legos, MPages Components don't work directly with each-other to build more complex structures. Instead, MPage Components are arranged on a page, more like a set of tiles to build an entire MPage which is a collage of the individual components.

Technically, speaking, MPage Components are JavaScript classes. By defining a class, you are able to instantiate many individual component object instances. Depending upon how the class is constructed, each component object may work exactly the same way, or the objects functionality may differ based upon the configuration parameters. If you don't know what a class is, see online references such as Wikipedia[1]. When we refer to components in this guide, we could be referring to classes or objects. The difference being that when you design and develop an MPage component it is a CLASS. When you actually use the component on an actual page, in an actual location on that page, it is an OBJECT instance. Since the purpose of this book is help you build a new component, we will be focusing on the definition of Component CLASSES. For most purposes, you can assume we are talking about defining a component class, unless we specifically refer to it as a Component Object Instance.

Another important distinction to understand is that MPage Components are a very specific type of component class. MPage components are designed to be placed on an MPage to display content in their own "frame" on the page. Think of it like a collage picture frame[2]. You're component goes in one of the sub-frames and renders its content for display. This is not too surprising. If you've seen the summary pages developed by Cerner or other sites, they all tend to have a fairly common approach to their layout and structure. The page consists of many "components" each with a header and placed on the page either in columns or occasionally free-form. The reason I bring this up, it that the term "component" can actually mean many things and you might find MPage developers using specialized components for other tasks on their MPage. This distinction is important.

[1] http://en.wikipedia.org/wiki/Class_%28computer_science%29
[2] The type of picture frame that has spaces for many pictures within the one main frame.

When we refer to **MPage Components**, we are talking about the "pictures" that go in the "frames" on the page. In order to easily add the components to the MPage, the MPage Components must conform to a very specific standard that we will discuss in the next section. Any other type of JavaScript "component" class can be used to help build your components internal functionality and standardize it across a library (such as ExtJS, jQuery, MooTools, etc.). However, there are no (or few) specific rules defined as part of this guide or the MPage Component "Standard" as to how they are used. There are best-practices for each tool and in general for JavaScript class development, but we are not covering that in this guide. Instead, it is left up to each framework or component developer to use JavaScript components or tools for their own development purposes as needed. As example in this analogy would be a component that adds a funny "call-out" comment to a picture. You might use the same component to add all sorts of funny labels to your pictures within the collage. Reusing them might make your components easier to develop, but we're not going to tell you that you have to use them or even how. In the end, we are mostly concerned with getting your picture into the frame correctly so the whole picture frame works.

3. Introduction – The "Standard" and a "Framework"

So how is this all going to work? If we have many different programmers building components and then adding them to a single page (it's still on web page running in IE), they all have to work well together. This is a big part of the two-pronged goal of our effort:

1. Design and build MPage Components that can be easily added to an MPage to enhance its value.
2. Make sure that all of the MPage Components added to a page work well together and don't "blow-up".

If you're familiar with HTML web page development; you know that variable scoping is not its strength (HTML web pages assume that two developers working on a page are coordinating their effort, which is not how components work). You also probably have seen many web pages that have JavaScript variables defined with global scope all over the place. Luckily for us, JavaScript can be really tightly scoped if you follow specific rules and best-practices.

To make this all work, about a dozen client organizations AND Cerner worked for the course of about six months to discuss, define and approve an MPage Component Standard. The standard is fairly complex and inclusive. It covers the methods, properties and events supported by each component. It defines the base-class functionality that we will discuss in the next section. It also defines how a framework will be built to interact with the base-class and component. It defines variable scoping rules, CSS scoping rules, naming conventions and standard functionality. If you want to understand how it all works, by all means read the entire standard. That being said, there are a lot of internal details that you don't need to know if you are just going to build a component.

Finally, we've already mentioned the concept of a "framework", but we haven't really defined what that precisely means. The name is fairly literal. A framework is a mechanism (JavaScript and some CCL code) that build the MPage and "puts" each component on the page. You can think of it much like the actual wood or metal frame of a picture collage. A framework exposes the functionality that the component developer uses to build their MPage Component. A framework manages the layout of the page as a whole. A framework helps to manage scope so that components don't accidentally interact with each-other and "blow-up". Is there a "The" framework? Not really, the framework that your site or your MPage is using may differ from site-to-site and page-to-page. There can only be one framework uses on a specific page, but there can be many different frameworks. For example, Cerner's MyPages is "A" framework. Trinity Health, UNM, Seattle Children's and many of the clients defining the component standard are building their own framework.

If you're interested in building your own framework, more power to you!!! It can be a worthwhile undertaking for advanced MPage development organization that want to expose more functionality and leverage greater power in their development. That being said, it is not easy, nor is it necessary. In many ways, it is simpler to simply use the Cerner MyPages framework or perhaps the basic reference framework that the standards group provides. In the end, you have to decide if you want to focus your resources developing more building blocks, or defining and supporting how the page level framework functions. If you wish to develop a framework, you will need to very carefully follow all of the standards outlined in the standards document. In addition, this will require fairly advanced JavaScript programming skills to do correctly. But like I said, this is not something you must to do…

4. Step 1 – Getting Started (Hello World)

This is the first practical chapter in the guide. In this section, we are going to build our very first component. Like most programming guides we are going to start with a "Hello World" component to help explain some of the basics. To begin with, we define **MyComponent** as a type of MPage Component. YOU WILL ALWAYS USE THIS SAME BASIC PATTERN WHEN YOU BEGIN DEVELOPING YOUR COMPONENT.

```
if (myorg == undefined){ var myorg = new Object();}
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;
```

In looking at the code shown above, the first thing you will notice is that we are defining my `myorg` as a new object. Why are we doing this? This is our first requirement for variable scoping. For each organization that builds components, they must define a namespace variable that all of their components will be scoped to. This should generally be your unique client mnemonic. For the University of Washington, they will use "univwa". You will also note that the `myorg` variable is only defined if it previously does not exist. This is done as a best-practice so that two components from your site will not redefine the variable once it already has content defined under it.

After defining the organization's namespace variable, we define `MyComponent` as an instance function of `myorg`. We then define `MyComponent` as inheriting from the `MPage.Component()` class. This is a very standard JavaScript pattern of defining a sub-class that inherits from a base-class. Why do we do this? When you first define the component using `myorg.MyComponent = function(){};` the component has not functionality. It is simply an empty class. Normally, you might go about defining its internal functionality next (methods, properties, events, etc.). However, in the case of MPage Components, we always start with some basic functionality. This is made possible by inheriting from a standard base-class. This is a technical concept that is central to the concept of object-oriented programming[3]. In our example, by inheriting from the MPage base-class you always have some functionality available in your component and you can then either (a) add functionality, or (b) override and change the standard functionality.

You're probably wondering what functionality is made available by the base-class. Once more, this is where the standard comes in. The base-class is actually built as part of the framework that you are using. Its internal logic and programming may differ from framework to framework, but the functionality that is exposed to your

[3] http://en.wikipedia.org/wiki/Inheritance_(computer_science)

component to use is always the same. In many ways, this guide is about that "exposed" functionality and how to use it. You don't need to know how the base class works, but you will need to know how to use its exposed functionality.

We could explain each feature first and then show you how to use it, but in the end, seeing the functionality in use is probably easier to understand. After showing you an example, we'll explain what is happening.

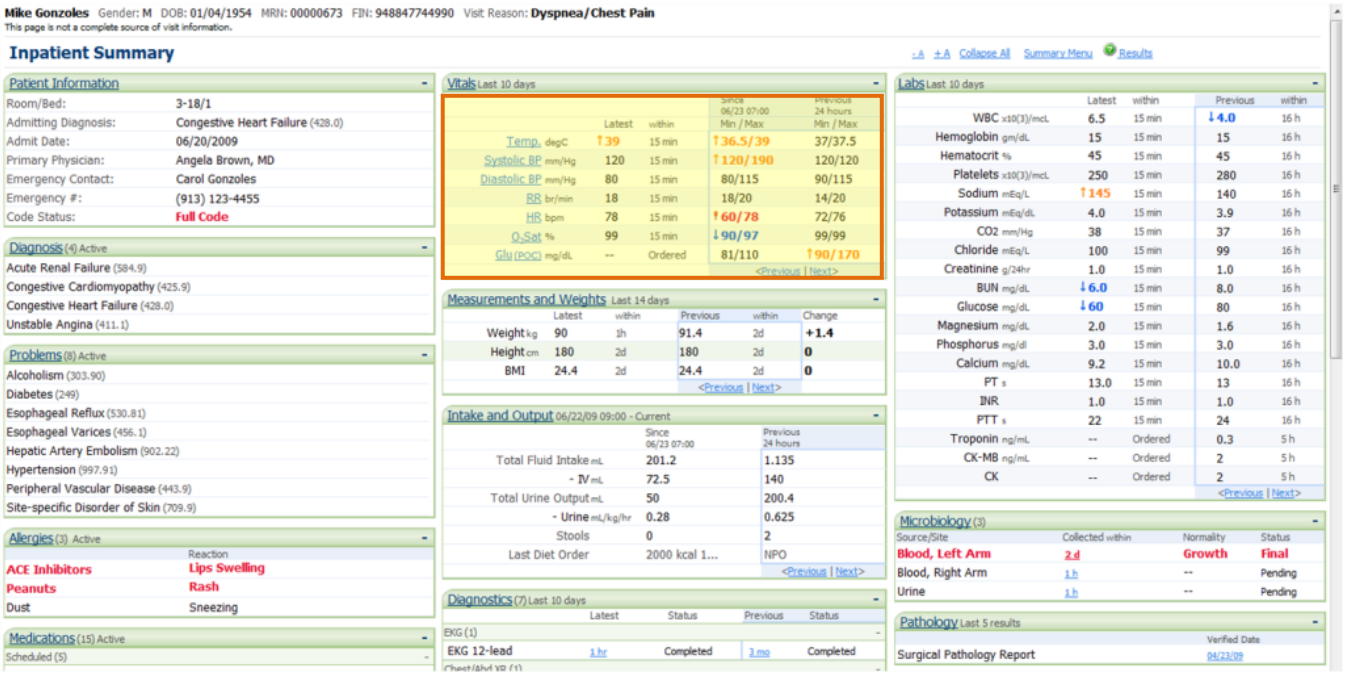
```
if (myorg == undefined){ var myorg = new Object();}
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;
myorg.MyComponent.prototype.render = function(){
	var oDiv = this.getTarget();
	oDiv.innerHTML = "Hello World";
};
```

This is an overly simple example of a component, but it accomplishes our basic task. This component will display the words "Hello World" in the component frame on a page when used. This also highlights a few important concepts that will be used through-out the guide. The `render()` function is actually defined by the base-class. It has default functionality to render any "string" content returned from the data source on the page. That being said, the whole point of a component is to render what you want on the page. This is where it starts. By re-defining the `render()` function, you can control exactly what renders on the page. One important point to note: whenever a base-class function is re-defined (i.e., overridden) you do so using the `prototype` constructor. This maintains the proper prototype chaining in the class inheritance.

Within the function, you will also notice that the program gets a DOM object by calling `this.getTarget()`. In JavaScript, this is a reference to the object that is currently in scope. In an example like the one above, this refers to the object instance of your component. It can be used to call and access methods and properties exposed by the base-class. It can also be used to access custom functions or variables that are defined as a public methods or properties of your component. In the case of `getTarget()`, this is actually a standard function exposed by the base-class. When you call this function in your component it will return the DOM object reference where you should be rendering your content. To render the component, you must set the `innerHTML` value for your target DOM object. This can be done many ways, but setting innerHTML is one of the easiest.

The idea that your component must render to its specific DOM object's innerHTML is very important and deserves a quick aside. When your component renders to the page, it should only be touching the page where it

is specifically allowed to. This is part of a black-box design principle that underlies the entire standard. Your component should NEVER update the HTML page outside of its predetermined scope. In the diagram below, you component is expected to render to the area of the page high-lighted in yellow. This is the DOM object that is returned to your component when `getTarget()` is called. You will also notice that each component has a standard header that includes a title and possibly other features. This header is NOT part of the component scope. You should never attempt to update it directly. There are base-class functions for doing this (such as setting the title at run-time, etc.).



Standards

1. You must define a namespace variable for your component. It should be your organization's unique mnemonic.
2. Your component must inherit from the MPage.Component() base-class.
3. The render() method should be overridden and used to render content to the page.
4. Any time you override a base-class method or property, you must use the prototype constructor method.
5. Your component should only render content to the innerHTML of the DOM object returned by this.getTarget().
6. You should never render content outside of the DOM target object, including the component header (which is part of the page and frameworks scope).
7. this.getTarget() should only be called within the render function (the DOM object may not exist prior to render being called).

5. Step 2 – Getting Data (Hello CCL)

Ok, great, we just build a component that does nothing? It didn't get any patient data. Sure we could make it do some pretty cool stuff using HTML and JavaScript, but the whole point of MPages is to integrate patient data. How do we pull patient data? There are actually many ways to get data for your component; we are going to start with the simplest. In most cases, you will want to have your component call a CCL script which will return data that you then render. The example below shows how to do this using the built-in CCL data request mechanism. In later chapters we'll show you how to customize your data requests.

```
if (myorg == undefined){ var myorg = new Object();}
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;
myorg.MyComponent.prototype.init = function(){
	this.cclProgram = "1_myorg_get_data";
	this.cclParams[0] = "MINE";
	this.cclParams[1] = this.getProperty("personId");
	this.cclParams[2] = this.getProperty("encounterId");
	this.cclParams[3] = this.getProperty("userId");
	this.cclParams[4] = "laboratory";
	this.cclDataType = "TEXT";
};
myorg.MyComponent.prototype.render = function(){
	var oDiv = this.getTarget();
	oDiv.innerHTML = this.data;
};
```

The above example shows a handful of concepts. First, the component overrides the `init()` function. This function is one of the three cores methods that your component will override. We've already seen `render()`. The other function that we will address shortly is `loadData()`. When the framework builds the page (that is its main job), it will create an instance of your component and then in sequence call `init()`, `loadData()` and `render()`. By overriding these methods you are able to control how the component works. The `init()` method allows the component developer to define internal variables as well as public properties that will be used later on. In the example above, it is used to set public properties that will then be used by the base-class `loadData()` method to make a call to CCL.

The public property `this.cclProgram` is a standard base-class property that defines what CCL program will be called. The public property `this.cclParams` is actually an array. As you are likely aware, all CCL programs take input parameters that vary from script to script. You can define those parameters by assigning them to the array (base 0 indexing). Like CCL, you can set either a number or a string. This determines how that data will be passed into CCL and if the data will be quoted or sent in as a numeric expression. The public property

`this.cclDataType` is set to TEXT. This tells the component that when the data is returned it should be treated like and stored as plain text. The other options are JSON (the default) and XML, which we will cover later.

You will also note that three of the parameter values are set to the person, encounter and user id using a special function. In order to access patient, encounter or user specific data, you will need to know what patient, etc you are looking at. To access these variables, you always use the `this.getProperty(name)` method. This is a standard base-class function that can be used to access public properties. To use it simply pass in the property name you wish to access. The component standard publishes the names of support properties. Common ones are listed at the end of this section. If you attempt to call `getProperty()` and a value does not exist, it will return undefined. You're program should be able to deal with this. Not all properties are required to be supported by the framework. Some are optional.

Finally, we have changed what our render method produces. Instead of simply inserting the words "Hello World", it will not insert what-ever text is returned from the CCL script using the `this.data` public property. The data property can be used for more complex purposes, but in the simplest use-case, it is simply set with what-ever data is returned from the CCL script. In the case above, the data is stored as plain text. If the data type had been set to JSON or XML, the data property would actually be an object of one of those types. In this case, assigning it to innerHTML directly would have thrown a runtime error. We'll discuss using those data types in the next section.

This is the first actually usable component concept. If you're CCL script had returned HTML content (say a simple) table it would have rendered it to the page. If you are developing a component using only CCL, you would basically use this component as a simple wrapper to allow you to render your content to the page. In the case of many frameworks (including MyPages) this simple component is built into the framework. All you actually have to do is configure the script name and input parameters to render the component.

One thing this guide does not cover is how to write the CCL script referenced above. That topic could cover a book of its own. The primary concern for this guide is how to build a component that can call a CCL script PROPERLY to retrieve the data you need. Besides the concepts addressed above, the primary concern with CCL scripts will be providing unique naming conventions so that two components do not conflict at this level. This will be covered in Step 8 – Naming Conventions. For now it is enough to know that each CCL script will be uniquely prefixed so that components from two sites will not accidentally conflict with each-other.

Standards

1. The init() function should be overridden to define public properties and other variables that will be used later in the component or those that will be used by standard base-class functionality.
2. Init() is called by the page prior to loadData() and render().
3. Init() should never attempt to render to the page or request data from a source using XMLCclRequest or Ajax. Those functions are reserved for loadData() and render().
4. this.cclProgram is a string representing the CCL script to be called.
5. this.cclParams is an array of the parameters that will be passed to the CCL script. They can be either strings or numbers.
6. this.cclDataType determines how the data returned from the CCL script will be treated. Either as TEXT, JSON or XML.
7. The default implementation of loadData() will use the cclProgram, cclParams and cclDataType properties to request and store data from a single CCL program.
8. The data returned from the CCL script will be assigned to this.data by default. The data type will be determined by the cclDataType property.
9. this.getProperty(name) is used to access a property by name (common examples are listed below).
10. this.getProperty() will return the value if it exists. Because some properties are not required to be supported, it may return undefined. Your component should deal with this gracefully.

Required Properties

personId
encounterId
userId

Optional Properties

PPRCode
positionCd
location (current users device)
defaultLocation (WTS)
appName
headerTitle
headerSubTitle
headerShowHideState
headerOverflow

6. Step 3 – Using JSON or XML

The simple example above requests plain text from the CCL script and renders it to the page. This is a functional model, but blurs the lines between the back-end data access and the front-end display. As a general programming best-practice this is not ideal. In general, your data access layer (i.e., CCL) should be concerned with getting and processing data. The front-end (i.e., your component) should be concerned with displaying content and managing user interactions. The standard does NOT require this, but many programmers will prefer this approach. Luckily, this is very easy to support using the example from above.

```
if (myorg == undefined){ var myorg = new Object();}
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;
myorg.MyComponent.prototype.init = function(){
	this.cclProgram = "1_myorg_get_data";
	this.cclParams[0] = "MINE";
	this.cclParams[1] = this.getProperty("personId");
	this.cclParams[2] = this.getProperty("encounterId");
	this.cclParams[3] = this.getProperty("userId");
	this.cclParams[4] = "laboratory";
	this.cclDataType = "JSON";
};
myorg.MyComponent.prototype.render = function(){
	var oDiv = this.getTarget();
	var sHTML = "<table>";
	for (var i=0; i < this.data.MYRECORD.ALLERGY.length; i++){
		sHTML+="<tr><td>" + this.data.MYRECORD.ALLERGY[i].VALUE + "</td></tr>";
	}
	sHTML+="</table>";
	oDiv.innerHTML = sHTML;
};
```

In the example above, the data type is set to JSON instead of TEXT. This tells the default implementation of `loadData()` to do one additional thing. After retrieving the CCL script's data it also attempts to eval() it as a JSON object[4]. Finally, instead of simply assigning `this.data` to the target DOM object, it is treated as a JSON object and the data is used to build a table string that injected into the page. It could also build a table object and inject that into the page. Both have the same result.

You'll notice in this example that data has a child object of MYRECORD, then an array ALLERGY with a property of VALUE. It is common practice to use CCL to populate a record structure with the relevant data (since this is a native data type that is structurally equivalent to JSON and mappable to XML) and then use cnvtrectojson() in CCL to convert it to JSON string output. This has two consequences. The name of the record is the first object

[4] http://en.wikipedia.org/wiki/Json

name after evaluation. Also, all of the object names, arrays and properties are in uppercase (which matters in JavaScript).

The example above uses JSON. However, XML works essentially the same way. The only major difference being that instead of setting `this.data` to a native JavaScript object (i.e., via JSON standard), it is instead stored as a XML DOM object. As a result, how you use that object differs from JSON.

Standards

1. If **this.cclDataType** is set to JSON, the default functionality will be to eval() the CCL response before assigning it to **this.data** as a native JavaScript object.
2. If **this.cclDataType** is set to XML, the default functionality will be to evaluate it as an XML DOM object before assigning it to the **this.data** property.
3. If either process fails during evalution, this.data will be left null and an error will be raised (more on error handling later).
4. Using JSON or XML is considered best-practice. The default value for cclDataType is JSON as this is become the de-facto web programming norm.

7. Step 4 – Flexing Components with Options

In the prior examples, the components are designed to do one thing and one thing only. This is ok if you don't have many broad needs. However, if you are trying to solve many problems (which we usually are) you may want to have a component that can be re-used. Sometimes, you might re-use the component on a different page with slightly different needs. In other case, you will re-use the component on the same page, but display different information. You could simply create two copies of the component and modify them slightly. This would work, but it is bad programming practice. Instead, you should make you code a little more modular and flexible.

This can easily be done with MPage Components. When a component is added to a framework, the user will have an option of defining some configuration parameters. This will vary from component to component, but they will always be stored in a JSON compliant format. When the component is created by the framework as an instance object of the class, the options will be assigned to the component under the `this.options` public property. This is shown below.

```
myorg.MyComponent.prototype.init = function(){
	this.cclProgram = "1_myorg_get_data";
	this.cclParams[0] = "MINE";
	this.cclParams[1] = this.getProperty("personId");
	this.cclParams[2] = this.getProperty("encounterId");
	this.cclParams[3] = this.getProperty("userId");
	this.cclParams[4] = this.options.event_set_name;
	this.cclParams[5] = this.options.days_back;
	this.cclDataType = "TEXT";
};
```

In the above example, a component developer assigns the "event_set_name" option to the ccl parameter property. In this case the actual value of "event_set_name" would be determined by the person who setup the component in the framework and defined the properties.

The properties for a component are always stored in a JSON object that is defined by the component developer. This usually consists of a flat object with name-value paired properties (such as "event_set_name"). However, in more complex objects, this could also consist of more advanced structures such as objects and arrays. Two examples are shown below (the first example being the one used in the code above).

```
{"event_set_name":"laboratory","days_back":3}

{"event_set":[
	{"name":"chem7","days_back":2},
	{"name":"cbc","days_back":2},
	{"name":"micro","days_back":7}
]}
```

Standards

1. The **this.option** string will contain a JSON compliant object based upon the options configured for that component.
2. Each component will determine and publish the acceptable options format and structure.

Best Practices

1. Each component should be designed to accept basic option to make the component more flexible and easily configurable.
2. The component should NOT assume that all of the options are entered, and should check for the existence before using or merge the options object with a default options object.

8. Step 5 – The Component DOM "Frame" (Overflow and Show/Hide)

In the prior sections we defined the scope of the component to specifically constrained to the DOM element that is returned from the getTarget() method. We also stated that the component should update the innerHTML of the DOM object. This implies a few things to us as well. The component should not attempt to modify the target objects properties such as its class, style, id, etc. In this interpretation, the component should not be able to control if its container is even visible or how big it is. This is intentional.

The standard expects the component developer to render the content within the target element as best it can. The component developer can access certain properties to help optimize the display (such as the target DOM elements height and width). The developer can even listen for resize events (covered later) to help re-optimize the rendering should the contain change size. However, the component developer should not attempt to write to the contains properties, including styles.

It is up to the page to determine the size of each target DOM container that the element will fit within. In some cases, this may involve provided a fixed width and height. In other cases the target DOM container will have a relative width and automatic resizing height. This was the design pattern to allow the page developer to control how component should be laid out on the page and to provide more predictable behavior.

The same thinking applies to show/hide and overflow properties. As explained below, the component developer may be able to access these states through properties. However, the component developer should never attempt to implement show/hide or overflow functionality for the WHOLE target DOM container. This is up to the page to determine how this layout will be managed. This is not to say that the component developer cannot implement overflow and show hide feature inside their individual component. That is OK as long as it is not intended to replicate the target DOM functionality. For example, a portion of a component may implement show/hide, but it just can do it for the WHOLE component. This was done to help maintain consistent functionality across components on a page.

Standards

1. Components should not write to the target DOM properties. Only the innerHTML should be set.
2. Components should not set the size of the target DOM. They can read from it to optimize their sizing, but they should not attempt to control the container size. Instead they should attempt to best render to the space given.
3. Components should not implement show/hide or overflow functionality at the target DOM container level. This is the responsibility of the framework.

9. Step 6 – Working with the Component Header

In the prior sections we saw some simple examples of how to build a component that will request information from CCL and write content to the DOM target element. What if we need to interact with the component header? In the prior sections we stated that you cannot write directly to this portion of the page's DOM. That would be a BIG no-no. What should you do if you need to change the component title or sub-title, or check if the component is currently being displayed or is overflowing?

All of these are managed by the page and are outside of the normal scope of the component. However, there is a way to work with them if they are exposed by the framework you are working with. If the framework exposes them as properties, you should be able to read and in some cases update these values for your component. To do so, you use the familiar concept of a property and the related this.getProperty() and this.setProperty() methods.

```
myorg.MyComponent.prototype.init = function(){
	var bSuccess = this.setProperty("headerSubTitle", "Last 3 days of results");
};
myorg.MyComponent.prototype.render = function(){
	var oDiv = this.getTarget();
	if (this.getProperty("headerOverflowState") == true){
		//do something
	}
};
```

In the above example, `setProperty()` is used to set the set the header subtitle to show that the component is displaying the last 3 days of results. `setProperty()` always returns a reference to the component object instance. This allows you to use the concept of chaining. You can literally chain together an array of `setProperty()` functions acting on the same object.

```
this.setProperty(name1, value1).setProperty(name2, value2).setProperty(name3, value3);
```

The `getProperty()` function can likewise be used to read in the current state of the header such as overflow state. In this case, the overflow state tells the component if the framework is currently overflowing and scrolling within its div. If a property is requested and does not exist, the function will return undefined.

There are four header properties that may optionally be supported by the framework in the 1.0 version of the Standard. They are listed below.

Header Properties

1. **headerTitle** is the main title that displays in the component header for each component.
2. **headerSubTitle** is the sub title that displays next to the component title in the header.

3. **headerShowHideState** determines if the component is currently showing or hidden. The possible values are true and false. This is usually implemented with + and – buttons allowing the user to show or hide the component.
4. **headerOverflowState** determines if the components target DIV is currently being limited in its display by overflow the content vertically. This is usually done either by using the CSS overflow property with a scrollbar or in some cases hiding the overflow content and including a “show all” button to expand it.

10. Step 8 – Naming Conventions

One of the most important goals of building components is the ability to support multiple components on a single page without having them inadvertently interact and "blow-up" the page. Using JavaScript classes for the components and scoping variables is a large part of what makes this work. However, it doesn't cover everything.

In the end, all components are going to be writing HTML content to the page and may also reference their own CSS definitions. If you've spent much time with either, you know that many of the relevant declarations in HTML and CSS are globally scoped. For example, when you defined a CSS class definition, it applies to the entire page. The only limiting factor is what DOM elements are assigned that class. In addition, two elements cannot share the same ID without causing potential conflicts.

Since we cannot namespace the declarations, we need to address the potential conflict another way. It's not quite as elegant as the class approach, but the next best option is to use a well-defined naming convention. In this way, we can be assured that two components won't accidentally share the same names. This doesn't require too much explanation. For the most part, you just need to follow the rules below and you'll be OK. For we get to the exact rules, there is one property that is worth discussing. In order to provide a unique naming convention, each component will need a way to access or generate a unique string for that instance of the component. Luckily, this is easy to do. Every component has access to a string that is unique to each component instance. This is accessed using the method `this.getComponentUid()`. Of note, this is NOT the same string as the component target's DOM id.

```
var sUnique = this.getComponentUid();
```

Naming Convention Rules

1) Always prefix an element id with the uniqueElementId AND include a description of the element it that makes it unique to your component instance as well (globally unique).

```
sElementId = this.getComponentUid() + "_element_1_2";
```

2) Always prefix an element name with the uniqueElementId (unique to each component instance).

```
sElementName = this.getComponentUid() + "_radiobutton";
```

3) If a CSS class is shared by all components at your organization, always prefix the class name with your organization's mnemonic (unique to your organization).

```
.myorg_tall {line-height: 20 px;}
```

4) If a CSS class is shared by all instances of one components, always prefix the class name with your organization's mnemonic AND your component's name (unique to your component).

```
.myorg_myComponent_tall {line-height: 20 px;}
```

5) NEVER define a class at the element level!!!

```
table * td {line-height: 20 px;}  !!! DON'T DO THIS !!!
```

6) If a CSS class is used by one instance of a component, always prefix the class name with you're the uniqueElementId (unique to each component instance). Of note, this CANNOT be done in a style sheet. There is no way to predict each unique component instance. Instead this is likely to be done during the rendering by assigning a class for functional purposes (i.e., jQuery matching).

```
sClassName = this.getComponentUid() + "_functional_class";
```

7) All class names must follow one of the above rules with two exceptions:
   a. If the class name is supported by an external library, that is OK, we have no control over it.

```
sClassName = "jquery_calendar_smoothness";
```

   b. If the class name is "Common CSS" name supplied with the MPage framework, you can use it without restriction.

```
sClassName = "mpage_result_abnormal";
```

8) All CCL programs called by your component should be prefixed with your 1_ and your client mnemonic (unique to your organization).

```
create program 1_myorg_get_user_prefs
```

9) If the CCL program is unique to your component class, it should also be prefixed with 1_ and your component name (unique to your component).

```
create program 1_myorg_mycomponent_get_data
```

11. Step 8 – Standard Versioning and Reverse-Compatibility

One thing that we have not yet addresses is changes to the standard. If there is a new standard, how do we know if your component supports it? If your component is added to an older framework, but requires a new feature, what will happen? How can you build a component to be reverse-compatible and not blow up?

These are all important questions, and the standard was designed with them in mind…

When you are building a new component, you need to address the minimum required standard that a framework must support for your component to work. For example, if a future version of the standard supports a new feature (reverseTwistKick feature – for fun…), perhaps your component won't work without it. You need to be able to tell this to the framework, so it will warn the user and not load the component (which could cause unintended consequences). You do this by setting the `componentMinimumSpecVersion` property. This tells the framework the minimum spec version that must be supported to load the component. When you define this value, you must do so during the definition of your component class and not at runtime. This is because your component instance may not exist when the framework is doing compatibility checking.

```
if (myorg == undefined){ var myorg = new Object();}
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;
myorg.MyComponent.prototype.componentMinimumSpecVersion = 1.2
```

If your component is able to support a lower spec version as a fall-back, it should publish that as the minimum. For example, if your component is able to check for the "reverseTwistKick" feature and implement it only if it exists and has an acceptable fall-back, then the lower spec version is the actual minimum. This allows advanced components to support reverse-compatibility. As shown in the above example, the component spec version is stored as a real number (###.#). This is in keeping with the standard version convention.

In the section above we mentioned having your component check if an advanced feature is supported before attempting to use it. This is a great way to maintain reverse compatibility. This can sometime be checked at runtime by inspecting JavaScript. In other case, you may wish to know the spec version that the framework supports. For example, if it supports version 2.3, you might want to use the "reverseTwistKick" feature. Otherwise, you might want to use a fallback mechanism. This is also easy to do. You can check the supported version at runtime by accessing the public property `this.baseclassSpecVersion`.

It should be noted if the framework detects that a components minimum required spec version it higher than the supported version, it will usually provide some sort of warning and NOT load the component. This is why

programming reverse-compatible components that keep the lowest minimum required spec version is important.

Standards

1. Each component must publish a minimum required spec version.
2. This should be done by setting the property componentMinimumSpecVersion to a real number.
3. This must be set with the class definition and not at runtime.

Best Practices

1. Code reverse-compatible components that have a fallback for advanced and future features where possible. This will help keep he minimum required spec version lower and allow them to work in more pages.

12. Step 9 – Putting it All Together (an Example)

The example below shows a complete implementation of a component that high-lights the tools that we addressed above. This component should work in any framework. It includes three files, which is a common practice for many components.

1) A JavaScript file containing the component definition and other organization level helpers
2) A CSS file containing class definitions for look and feel and layout
3) A CCL program file that contains a script for accessing data

Finally, you will notice that some of the internal functionality is defined by creating additional functions. This is OK and general good practice. Like overriding a base-class method, the internal methods should be defined using prototype. However, in addition, they should ALSO be named with an underscore "_" as a prefix. This identifies them as private and avoid namespace conflicts.

myorg_mycomponent.js

```
/**************************************
Supported Options:
       {"username":"DSTONE"}
**************************************/

//define client mnemonic namespace
if (myorg == undefined){ var myorg = new Object();}

//inherit component from base-class
myorg.MyComponent = function(){};
myorg.MyComponent.prototype = new MPage.Component();
myorg.MyComponent.prototype.constructor = MPage.Component;
myorg.MyComponent.prototype.base = MPage.Component.prototype;

//set minimum spec required
myorg.MyComponent.prototype.componentMinimumSpecVersion = 1.0;

//override init
myorg.MyComponent.prototype.init = function(){
       //set CCL program to query
       this.cclProgram = "1_myorg_mycomponent_get_data";
       this.cclParams[0] = "MINE";
       this.cclParams[1] = this.getProperty("personId");
       this.cclParams[2] = this.getProperty("encounterId");
       this.cclParams[3] = this.getProperty("userId");
       this.cclParams[4] = this.options.username
       this.cclDataType = "JSON";

       //set a header sub-title value
       if (this.options.username != undefined){
              var bHeaderSubTitleSuccess = this.setProperty("headerSubTitle",
                     "Username: " + this.options.username);
       }

       //do something that is future proofed (avoid runtime error)
       if (this.baseclassSpecVersion == 3.0){
```

```
            //only exists in the FUTURE
            this.doSomethingAwesome();
      } else {
            //let user know that awesomeness is not supported in this framework version
            this.setProperty("headerSubTitle", this.getProperty("headerSubTitle") +
                  " awesomeness not supported in " + this.baseclassSpecVersion + "!!!");
      }
};

//override render
myorg.MyComponent.prototype.render = function(){
      //get div to render to
      var oDiv = this.getTarget();

      //get unique id
      var sUniqueId = this.getProperty("uniqueElementId");

      //build output
      this._drawOutput(this.data.RESPONSEDATA);
};

//private draw component
myorg.MyComponent.prototype._drawOutput(oData){
      //build output
      var sHTML = '<table class="myorg_table_header">'+
            '<tr><th>Username</th><th>Name</th><th>Person Id</th></tr>';
      if (oData != undefined){
            if (oData.FOUND == "yes"){
                  sHTML += '<tr><td class="myorg_mycomponent_username">'+
                        oData.USERNAME+
                        '</td><td>'+oData.NAME+
                        '</td><td>'+oData.ID+
                        '</td></tr>';
            } else {
                  sHTML += '<tr><td>'+ oData.USERNAME+
                        '</td><td>Not Found</td><td>Not Found</td></tr>';
            }
      } else {
            sHTML += '<tr><td>'+this.options.username+
                  '</td><td>Error</td><td>Error</td></tr>';

            //raise error for framework to handle (see Advanced Error Handling)
            this.throwNewError("Error finding user: " + this.options.username);
      }
      sHTML += '</table>';

      //render
      oDiv.innerHTML = sHTML;
}
```

myorg_mycomponent.css

```
.myorg_table_header * th {
      background-color: #dddddd;
      font-weight: bold;
}

.myorg_mycomponent_username {
      background-color: yellow;
}
```

1_myorg_mycomponent_get_data.prg

```
drop program 1_myorg_mycomponent_get_data go
create program 1_myorg_mycomponent_get_data

prompt
       "MINE" = "MINE",
       "Username" = ""
with OUTDEV, USERNAME

record responsedata (
       1 found = vc
       1 username = vc
       1 name = vc
       1 id = f8
)
set responsedata->found = "no"
set responsedata->username = $USERNAME

select into "NL:"
from prsnl
where username = $USERNAME
detail
       responsedata->found = "yes"
       responsedata->name = trim(name_full_formatted,3)
       responsedata->id = person_id
with time=5
declare strJSON = vc
set strJSON = cnvtrectojson(responsedata)

select into $OUTDEV
from dummyt
detail
       col 0, strJSON
with format=variable, maxcol=50000, formfeed=none, maxrow=0

end
go
```

13. Step 10 – Publishing Your Component

Publishing your new component should be pretty easy. This guide does not cover how to publish content to the App Store. Instead, we are concern with how you should document and wrap up your component for delivery to another organization. To do this, simply follow the simple rules listed below.

1) Collect the required files.
    a. JavaScript for the component
    b. CSS for the component
    c. CCL program files for the component
2) Collect any dependencies
    a. JavaScript libraries
    b. Shared CSS
3) Document the code requirements and dependencies
    a. Use the form attached to this guide.
    b. This includes: requirements, descriptions, options, licensing requirements, etc.
4) Zip all of the files together (for easy delivery)
5) Deliver the code or host it on the App Store

The key to making this work is really in the documentation. The documentation should be sufficient to tell another organization the class name, any options that are supported, the minimum version number, any special requirements or instructions to localize the component, and any dependencies that must be supported.

A standard template for publishing your component is made available as an attachment at the end of this guide (***Publication Template Attachment***).

14. Advanced – Customizing loadData()

So far we've seen examples that involve requesting data directly from CCL using the base-class loadData method. In these examples, our goal was simple:

1. Define a CCL script used to retrieve data
2. Initialize the query based upon configuration option
3. When the data returns, render our content to the page

This is enough to meet MOST needs. However, what happens if our data source is not CCL? Or, what would we do if we needed to request data from multiple scripts, or perhaps even a combination of the two. In this case, we are permitted to override the loadData() method and define our own version of data retrieval. This is described below. It should be noted that this requires much more complex programming tools including an understanding of call-backs and basic multi-threaded processing flow (very similar to AJAX). Before showing an example, it's worth outlining the technical requirements:

1. loadData contains one input parameter: a callback function that expects the input parameter to be a reference to this component.
2. Any request to a data source must be made asynchronously using AJAX or XMLCclRequest.
3. Requests to XMLCclRequest are best made using loadCcl(), which will handle the asynchronous request.
4. After the data is returned (via a callback due to its asynchronous nature), the original loadData callback function must be called with the component object instance as an input.

That's pretty much it… However, the requirements can be deceptively complex. To help illustrate all of this take a look the example below.

```
myorg.MyComponent.prototype.loadData = function( onLoadedData ){
	oMyObject = this;

	var afterLoadData1 = function( cclData ){
		var sRequestParam = cclData.SOME_VALUE;
		this.loadCcl(
			"myorg_get_more_data",
			["MINE", this.getProperty("personId"), sRequestParam],
			afterLoadData2,
			"JSON"
		);
	};

	var afterLoadData2 = function( cclData ){
		this.data = cclData;
		onLoadedData(oMyObject);
	};

	this.loadCcl("myorg_get_data",
		["MINE", this.getProperty("personId")],
```

```
                afterLoadData1,
                "JSON"
        );
};
```

The example above demonstrates three major concepts: how to chain two CCL requests together using their callbacks, how to use `loadCcl()` and how to assign the response to `this.data`.

In the example above, the first thing that is done is to set a local variable `oMyObject` as a reference to `this`. This is done because during an asynchronous callback from XMLCclRequest, the reference to this may be lost.

The next step is to define two callback functions that will be passed into `loadCcl()`. These functions are each only executed after `loadCcl()` completes its data request. The first callback function, `afterLoadData1`, has an input parameter that is the data returned by `loadCcl()` (either TEXT, JSON or XML). This callback function then calls `loadCcl()` again, but this time calling a different CCL script. In this example, the second CCL script actually uses a response value from the first script to help seed the second request. This might be a common approach if you are working with modular code where two CCL scripts are needed to get all of your data. Upon completing the second `loadCcl()` request, the second callback, `afterLoadData2`, is called (because it is passed into the function). This final callback then assigns the data that is returned to the `this.data` property to be used later by render.

The final step occurs in the last callback/thread in loadData. As the final step in the processing, the loadData callback function, `onLoadedData()`, is called with a reference to the object instance. This steps returns control to the framework and allows it to continue processing.

In looking at this structure, you may wonder why it is important to (a) make asynchronous calls and (b) call back to the framework. The asynchronous calls make sure that the page does not freeze during a long request. If you image many components making requests for data on a page, you would not want the requests to run one after another. In that case, the page would take the sum of the load times of each individual component to load completely. If one component was slow, this could take a very long time. By running the processes asynchronously, the components can load in parallel and independent of each other. The call back structure is related to this. When the framework calls loadData() for your component, it also needs to know when it is time to call render(). This has to occur after loadData() is completed. If the page simply called loadData() immediately followed by render(), this would not work. The asynchronous request for data would still be processing independently and the data would not yet have returned. The callback structure solves this problem

and is come with AJAX type requests in JavaScript. Finally, the callback also allow the page to chain together subsequent loadData() requests for independent components. If the page simply called all of them at once, the maximum supported threads would execute both on the front-end in IE as well as in CCL. With IE8, this would be 6 threads. By using a callback structure, it is possible to develop a framework that controls how many components are loading in parallel.

The example above shows how to override loadData() to make two chained CCL requests. The other primary reason to override loadData() is to make requests to alternative data sources. For example, you may wish to request or write information to a MOWA data service, a CareAware API or an entirely different data service outside of millennium. This guide does not cover HOW to accomplish those requests, but it should be noted that AJAX or another mechanism is generally required so that the calls can be made across domains and asynchronously. If you are making such a data request, the primary requirement is that it is asynchronous and executes the correct callbacks.

It should be noted that requests for CCL data from an MPages Web Service DO NOT NECESSARILY require you to override loadData(). Some frameworks have a built in detection feature which will route the request to either XMLCclRequest or the MPages Web Service depending on where the request originated (i.e., PowerChart or a web browser). In the future, this is likely to become a standard feature of all frameworks.

Standards

1. **loadData()** has one input parameter a callback function.
2. The **loadData()** callback function must be called as the LAST step in any processing after any asynchronous requests have returned.
3. The input parameter for the callback function must be a reference to the component object.
4. Calls to CCL must use **loadCcl()** as it will optimize the request and manage any routing.
5. The input parameters to **loadCcl** are:
    a. The CCL script name (string)
    b. The CCL script parameters (array – same as this.cclParams)
    c. A callback that will be executed after the CCL request completes.
    d. The CCL data type (TEXT, JSON or XML – same as this.cclDataType)
6. The **loadCcl()** callback function will include one parameter, the returned data (after conversion to the correct data type).
7. All data requests must be asynchronous and the final callback must execute the loadData() callback per requirement #2.
8. The loadData() function should never render content to the page. Any data should be stored in local variable or in this.data.
9. The loadData() function should NOT call render(). The framework will determine when render() should be called.

15. Advanced – Listening to Page Events

There are two page-level "events" that are part of the standard. In this particular, case they are implemented as overridable functions as opposed to event listeners. This does not limit their functionality as they are specific to each component and there is no need to multiple registered listeners. Instead the page is responsible for calling each of the appropriate component "event" functions when appropriate.

The first event is unload(). By default, each components base-class has an unload function that does nothing (it just returns null). The framework, will call this function when the page or component attempts to unload. This can be useful as it allows a component developer to attempt to free up any memory or events as needed. To utilize this event, the component program simply overrides what the function does when called.

```
myorg.MyComponent.prototype.unload = function(){
        sId = this.getComponentUid();
        oLink = document.getElementById(sId+"_my_linik");
        oLink.onClick = nothing
};
```

The second event is resize(). The resize function works like unload(). It is called when the size of the component changes in a meaningful way. This function has to parameters: width and height. The component can override resize() to help re-render the component or check if it still fits appropriately.

```
myorg.MyComponent.prototype.resize = function(width, height){
        if (width < (myPriorWidth * 0.8)){
                this.drawComponent();
        }
};
```

Standards

1. The component developer can overload the two functions: **resize()** and **unload()** to trigger when these events occur.

Best Practices

1. Depending upon how you have rendered your component, it may be best practices to unbind certain events or de-allocate memory.
2. If you component is rendered specific to the size of the components target DOM element, it should override the **resize()** method to re-optimize its display when appropriate.

16. Advanced – Error Handling

Errors are inevitable in programming. No matter how well you test your component it will throw errors. In general, when an error occurs your component should attempt to deal with it gracefully. If the component is able to recover from the error and no further notification is required, then processing can move forward normally. This would usually be accomplished using a try/catch routine in JavaScript.

If, however, the error cannot be fully recovered from, then the error must be bubbled up to the framework. Why do we require this? Frameworks will implement error handling mechanism that will allow them to capture the error that is raised and deal with it appropriately. This may include highlighting the problem component. It may also include logging an error to a debugging file. There are two ways this can be accomplished; however, the first is definitely the best practice

Throw New Error Method (Best Practice)

Your component can identify and throw an error that the framework will handle at any time during process or during future runtime (i.e., DOM events). This is done by calling a method `throwNewError()`. This method has two input parameters. The first is a description string which is required. The second is an error object which is optional. This error object would only exist if a try/catch routine was used to capture the actual runtime error. This method is advantageous in the following ways:

1. It can be called asynchronously during an AJAX request, setTimeout or DOM event.
2. It allows the current component process to continue if needed.
3. It can easily be used to register logical non-runtime errors.

Throw or Bubble Up

If you throw a new error (using throw new Error()) or an error occurs during processing, it will naturally bubble up to the framework where it will be caught and dealt with. However, there are a few places where this will NOT work:

1. Within an asynchronous AJAX process (i.e., XMLCclRequest.onreadystatechange{}, etc.)
2. Within an asynchronous setTimeout() or setInternal() call.
3. Within an asynchronous DOM event (i.e., onClick="")

In these cases, it is critical that the component developer includes a try/catch routine and then raises the error using this.throwNewError(). This is required because the framework cannot listen and capture errors that occur asynchronously.

As you may have noticed, there is no need to use the second method for raising errors. It is included essentially as a backup. Instead, you should always attempt to use `throwNewError()` as shown below.

```
myorg.MyComponent.prototype.loadData = function( onLoadedData ){
	oMyObject = this;

	var me = this;
	var afterLoadData1 = function( cclData ){
		try {
			me.data = eval('(' + cclData.JSONDATA + ')');
		} catch (err) {
			me.throwNewError("Error evaluating JSON data: " +
				cclData.JSONDATA, err);
		}
		onLoadedData(oMyObject);
	};

	this.loadCcl("myorg_get_data",
		["MINE", this.getProperty("personId")],
		afterLoadData1,
		"JSON"
	);
};
myorg.MyComponent.render = function(){
	if (this.data.LIST.length == 0){
		this.throwNewError("No data returned.  LIST length = 0.");
	}
};
```

The above example shows how to catch errors in an asynchronous callback as well as in the synchronous call to render using `throwNewError()`.

Finally, your component should NEVER directly alert the user that an error has occurred using the `alert()` function. This interrupts processing logic and affects the page as a whole. Instead, any visual alert should be handled by the page or embedded in the components display as appropriate by calling `throwNewError()`.

Standards

1. Call throwNewError() to raise error events when they occur.
2. Never alert() users to errors. The framework should handle this.
3. Errors can bubble up from your component, but NEVER during asynchronous processes.

Best Practices

1. Use throwNewError() instead of event bubbling.

17. Advanced – Repeated Execution and Appending

What would happen if a component was reloaded by a framework? For example, what if the configuration options were update by a user and the component was supposed to reload its data. In this case the framework will simply call init(), loadData() and render() again in sequence.

In most cases, this should work just fine. However, if you program your component to append content to the target DOM element, then you might suddenly see two version of it. When you are programming your component, you should keep this in mind and always insert elements at the top level to avoid this problem. When data is returned from/to loadData() it should always set the value of any localized data and not append to it. The two examples below would be a problem if the component was reloaded.

```
this.data.push(cclData);

$(this.getTarget()).append("<table>");
```

Standards

1. Code your components assuming that init(), loadData() or render() could be called again.
2. Be careful appending data or content on the page as this can cause repeat display if the functions are called again.

18. Advanced – Dependencies (JavaScript, CSS, etc.)

If you've spent much time working in JavaScript, you've no doubt discovered that it's faster, better and easier to work with pre-existing libraries to help accomplish your tasks. These are often freely available on-line under open source licenses. Sometime, they may require a license be purchases. In either case, they can be very useful. Some of these libraries are simpler helpers like the date.js and time.js library that improve JavaScripts native date handling. In other cases, they are complex libraries like jQuery, extJS and Mootools that handle a broad array of complex development tasks.

In either case, it is OK to use these with your component. The challenge that we will face is this… Some of these libraries may conflict with each other. In some cases, there is not much we can do other than publish the libraries uses so we can track these conflicts. In other cases, libraries may share a global namespace such as the $ variable. However, they may also have the ability to run the library in a safe mode thereby limiting the risk of two libraries using the $ variable. Either way, a risk still exists. To help mitigate this we have include a few required practices and best practices.

Required Practices

1. List ALL of your dependencies when publishing your library.
2. Include the full range of supported versions.
3. Ideally, list any dependent functions that may be a conflict issue.

Best Practices

1. Limit the use of external dependent libraries to those that avoid global namespace pollution (i.e., libraries that scope all of their functionality under one variable).
2. If a library uses the $ or other common global namespace functions, attempt to run it in a safe mode by using the full function namespace (i.e., jQuery).
3. Many programmers will use jQuery, so avoid using dependent libraries that overload the $ function (de-facto best practice due to usage volumes).

The last reason that we list dependencies when publishing a component is that the person setting up the framework or page will need to reference your dependency so that your component will correctly load. You will notice that we require you to include the full range of supported versions. This is important in case two component developers are using the same library on the same page. To make this work, the lowest common denominator library will likely be included on the page (since two versions likely cannot be included on one page).

19. Advanced – Adding Other Public Interfaces

As you're building your component, you may be tempted to add additional methods and functions to your component to accomplish and internal task or make the component easier for a developer to use. The first action is perfectly acceptable. You can define any number of private functions that are used by the component to accomplish internal tasks. In addition, you can do this at the mnemonic level with shared functions (discussed later). However, the second use case is strictly prohibited.

> **IT IS ABSOLUTELY PROHIBITED TO DEVELOP ADDITIONAL PUBLIC INTERFACES FROM A COMPONENT THAT WILL BE CONSUMED BY A USER OF THE COMPONENT.**

This because this will cause interoperability and interchangeability issues. You might be tempted to add a single "Generate()" function that will execute init(), loadData() and render() all in one call. Although the intention of making the component easier to use is noble, the approach is wrong. Your component must overload the base-class public interfaces only and cannot expose additional public interfaces.

Making it easier to use a component is not up to the component developer. Instead, this is up to the framework developer!!!

The framework renders the page, instantiates the components and can provide an interface to page developers to add a component to a specific page instance. This is where any "Generate()" function or related helper mechanism should come into play. Luckily for you, this is not the worry of the component developer. You simply build your component to do its job well and conform to the specs. The framework developers will make them easier to use.

Does this mean that you need to also build an easy to use framework? Well that depends… If you plan to build your pages all from scratch, the answer is probably yes. The good news is that the component standard committee has started you out with an open source framework example that you can extend or just plain use. More importantly, for most organizations, they will be using the Cerner MyPages, which is an advanced standard compliant framework in-and-of-its-self. MyPages goes one step further than defining a "Generate()" function at the JavaScript level. It allows a page developer to simply configure the page in Bedrock and handles the JavaScript generate tasks itself.

20. Advanced – Sharing Code across Components

If you organization builds a lot of components, you will quickly realize that each component is repeating some common functionality. Building this in each class is far from efficient or supportable. A better way to do this would be to include the functionality in a standard library. This can easily be done using your client organization's mnemonic namespace. Just like we scope each component class to the client namespace, we can also scope shared functions and variables to this namespace.

Share variables can be used to components together with shared functionality and data. This will be shown in the next section. Shared functions can be used to re-use functionality across components. This is really quite simple. Simply define your function as a child of your client mnemonic namespace object variable. This is shown below.

```
if (myorg == undefined){ var myorg = new Object();}
myorg.addNumbers = function( num1, num2 ){
       return(num1 + num2);
}
```

This is a rather simple example, but then again, so are there requirements for shared functions…

If two organizations wish to share functions, then they should work together to create a library and scope it a shared name. After that, it is essentially treated like any shared external library and should be published as such.

Standards

1. Scope all shared functions under your mnemonic namespace variable.

Best Practices

1. If you develop many components, try to share common functionality using shared functions at the client namespace level.

21. Advanced – Linking Components

Occasionally, you will be faced with a complex development task that requires you to have one or more components work together on a page to accomplish a task. For example, a group of components on a page may wish to share the same data and display it differently. This could be the case, where there are many orders summary components and a single CCL call to get all active orders is quicker than multiple calls to get active orders by clinical category. By having one component retrieve the data and share it with the rest is quick. Additionally, one component may wish to trigger an event that causes the other linked components to respond.

This should set off alarm bells… After all, one of the guiding principles of the component standard is that each component is an independent black-box. We require that the component stores variables and other information within the component and not at the page level scope. Fortunately, the standard does leave open an avenue for this type of development. Although components should NEVER store information at the page level (i.e., global scope), it is OK for a component to store information and interact with functions defined at the client mnemonic level. This was outlined in the prior example.

This provides a sufficient mechanism for components to share information and link functionality. For instance, two components: A and B might be designed to share the same information (so that it is only queried once), but display it differently. To make this work, you will need to develop code at the mnemonic level that links the components data together. There are many ways to accomplish this within the boundaries of the standard's requirements. We won't provide specific code, but as an example…

1. A client could define a function or namespace at the mnemonic level.
2. Each component, A and B, would have their loadData() method overloaded so that:
    a. It would check the mnemonic to see if the data is stored there
    b. If it is, it would simply pull it locally and use the data
    c. If it isn't, it would make a loadCcl() call to retrieve the data and then store it both locally and under the namespace

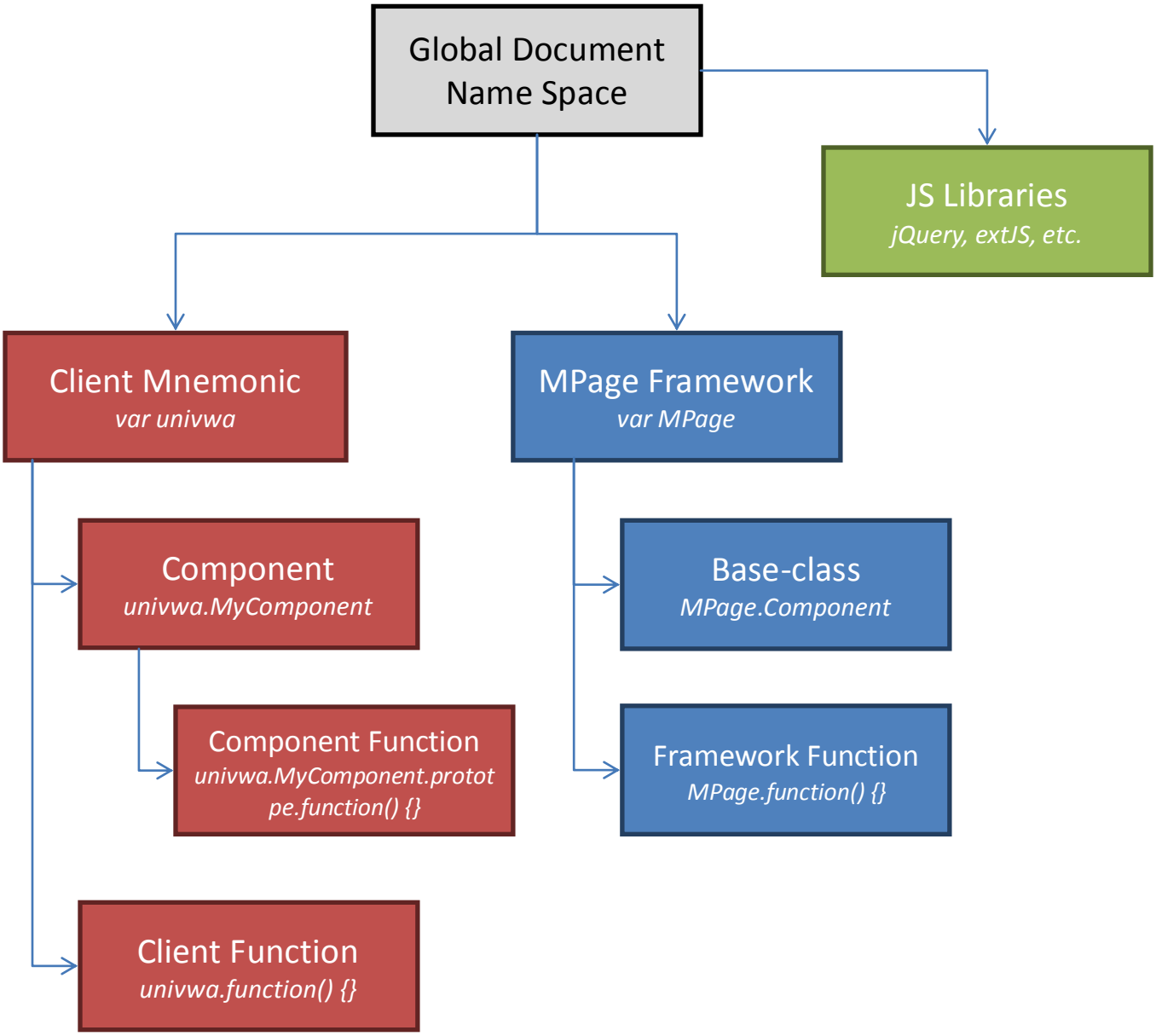
To make this work given the asynchronous nature of the data pulls, it may require that the first component to make a data call also sets a flag saying it is in progress. The second component would then register a callback if the query was in-progress so that when the data returned, functionality at the mnemonic level could trigger that call-back to allow the second component to continue processing inside loadData().

There are a few very important implications of how this is allowed and done.

1. COMPONENTS MUST BE INTENTIONALLY DESIGNED TO WORK TOGETHER. You cannot link functionality between components unless the developers intentionally built them that way.
2. THE STANDARD DOES NOT DEFINE HOW COMPONENTS SHOULD WORK TOGETHER. This is purely up to the developers to build components that work together.
3. In general, components that are linked should come from the same client and hence exist under the same mnemonic. It is possible to cross client mnemonics, but this would require that both clients are working together on an ongoing basis to support this and this will create many dependencies.
4. YOU MUST DOCUMENT ALL LINKED COMPONENT DEPENDENCIES. Linking components will create more dependencies, hence requiring a single page to have access to more common and shared code.
5. **AS A BEST PRACTICE, THIS SHOULD BE AVOIDED** UNLESS THE COMPONENTS ARE ALWAYS GOING TO BE WORKING TOGETHER ON THE SAME PAGE. Otherwise, we have made it much harder to mix-and-match components on a single page in an interoperable way.

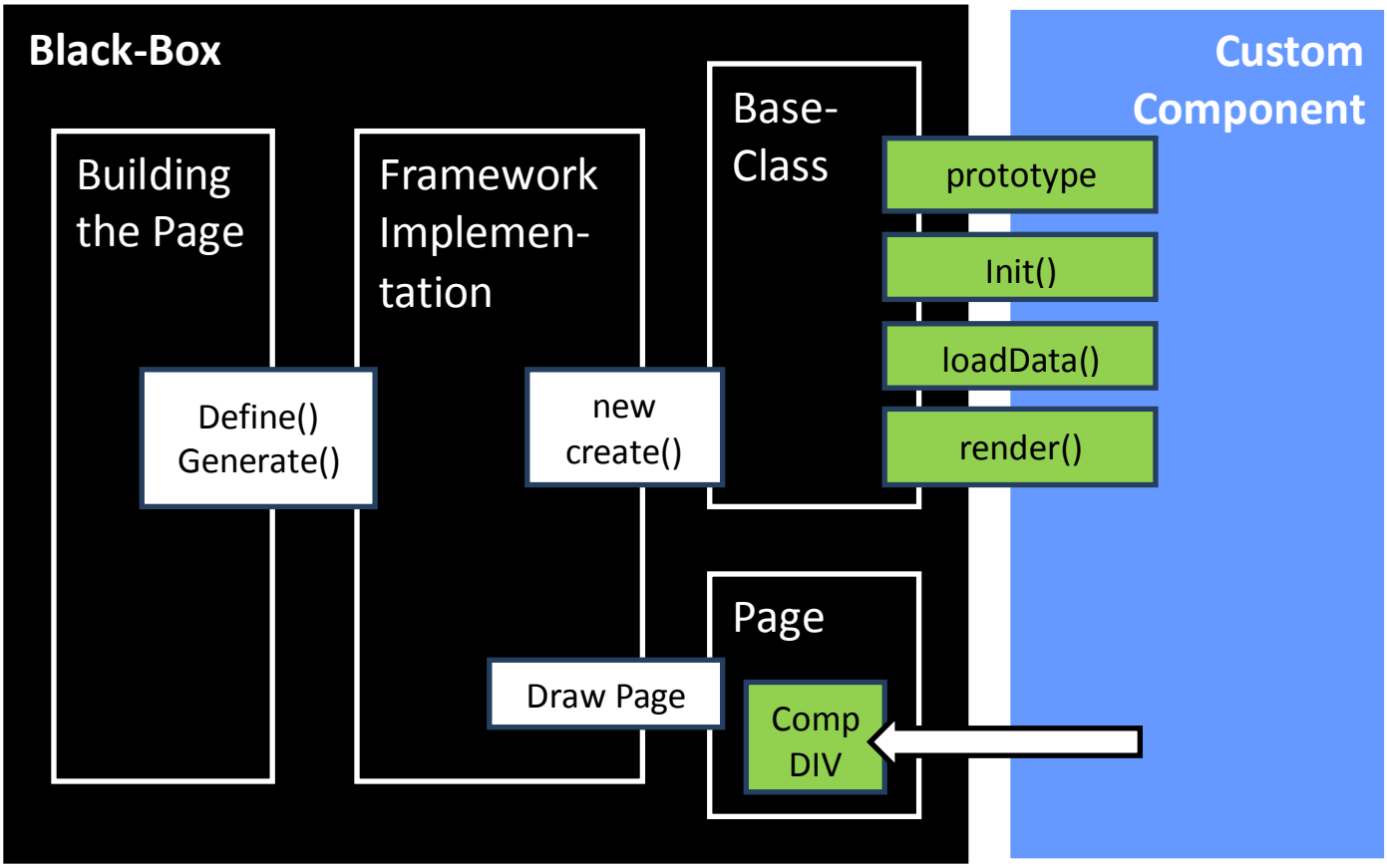
Appendix 1 – Name Space Map

The following diagram outlines the name-spacing that must be followed when developing components a framework or defining client specific functions, etc.



Appendix 2 – Framework Black-Box Design Principle

The following diagram illustrates the black-box design principle that separates a component from the framework and page. One important piece to note, the person building the page (far left) may be interacting with a framework that loads the components, but rarely the components themselves. From a component development perspective, you should NOT concern yourself with how your component will be implemented on a specific page. The Framework developer will expose functions to make this task easier.



Appendix 3 – Publication Template Attachment

<table>
<tr><td>Component Name:</td><td colspan="5">My Component</td></tr>
<tr><td>Description:</td><td colspan="5">My component pulls the current uses name and id and displays it for debugging purposes.</td></tr>
<tr><td>Client Organization Mnemonic:</td><td colspan="5">myorg</td></tr>
<tr><td>Class Name:</td><td colspan="5">myorg.myComponent</td></tr>
<tr><td>Component Version:</td><td colspan="5">10.1</td></tr>
<tr><td>Min. Required Framework Version:</td><td colspan="5">1.5</td></tr>
<tr><td>Framework Version Supported:</td><td colspan="5">3.5</td></tr>
<tr><td>Licensing Requirements:</td><td colspan="5">Client licenses as freely available through App Store using MIT license</td></tr>
<tr><td>CORE Component Specific Files:
(JS and CSS and CCL)</td><td colspan="5">myorg_mycomponent.js
myorg_mycomponent.css
1_myorg_mycomponent_get_data.prg</td></tr>
<tr><td>Other Dependent Files:
(JS and CSS and CCL)</td><td>File Names</td><td>Inc?</td><td>Licensed Under</td><td>Version Required</td><td>Tested With</td></tr>
<tr><td></td><td>myorg_common.js</td><td>Yes</td><td>Client</td><td>1.0</td><td>1.0</td></tr>
<tr><td></td><td>myorg_common.css</td><td>Yes</td><td>Client</td><td>1.0</td><td>1.0</td></tr>
<tr><td></td><td>myorg_check_prefs.prg</td><td>Yes</td><td>Client</td><td>ALL</td><td>1.0</td></tr>
<tr><td></td><td>jquery.js</td><td>No</td><td>MIT</td><td>2.0+</td><td>2.3.1</td></tr>
<tr><td>Dependency Descriptions:</td><td colspan="5">jquery is used for basic functionality and should be compatible with all versions 2.0 and above.</td></tr>
<tr><td>Special Instructions:</td><td colspan="5">None</td></tr>
<tr><td>Special Notes:</td><td colspan="5">None</td></tr>
<tr><td>Options Parameters:</td><td colspan="5">{“username”:””,”active_only”:1}</td></tr>
<tr><td></td><td>Parameter</td><td colspan="2">Use</td><td colspan="2">Values / Examples</td></tr>
<tr><td></td><td>Username</td><td colspan="2">Set the username that will be sent to the component</td><td colspan="2">Ex: DSTONE</td></tr>
<tr><td></td><td>active_only</td><td colspan="2">Only quality active users based upon active_ind = 1</td><td colspan="2">1 = active only
0 = all</td></tr>
</table>