QCObjects Documentation

Release latest

Oct 22, 2021

Contents

Fig. 1: logo

Welcome to QCObjects Main Reference Documentation! Check the official page of QCObjects at https://qcobjects.dev

CHAPTER 1

Cross Browser Javascript Framework for MVC Patterns

QCObjects is a javascript framework designed to make easier everything about the MVC patterns implementation into the pure javascript scope. You don't need to use typescript nor any transpiler to run QCObjects. It runs directly on the browser and it uses pure javascript with no extra dependencies of code. You can make your own components expressed in real native javascript objects or extend a native DOM object to use in your own way. You can also use QCObjects in conjunction with CSS3 frameworks like [Foundation] (https://foundation.zurb.com), [Bootstrap] (http://getbootstrap.com) and mobile javascript frameworks like [PhoneGap] (https://phonegap.com) and OnsenUI (https://onsen.io)

Fig. 1: screenshot

1.1 Table of Contents

CHAPTER 2

ALPHA RISE Startup

Fig. 1: alpha

QCObjects was invited to exhibit as an ALPHA Startup in the RISE Conf Hong Kong 2019. RISE attracts the most dynamic startups from around the world. We'll be showing how QCObjects is making a real Global Impact to the JavaScript developers life transforming the way for coding.

If you want to find out more about RISE event check out their website https://riseconf.com

CHAPTER 3

ECMA-262 Specification

See ECMAScript® 2020 Language Specification for reference

CHAPTER 4

Copyright

CHAPTER 5

Demo

CHAPTER 6

Demo Using Foundation

Check out a demo using Foundation components here: Demo Using Foundation

CHAPTER 7

Demo Using Materializecss

Check out a demo using MaterializeCSS here: Demo Using Materializecss

CHAPTER 8

Demo Using Raw CSS

Check out a demo using raw CSS: Demo Using Raw CSS

CHAPTER 9

Another Basic Demo example: The simpliest demo example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Demo</title>
        <script type="text/javascript" src="https://qcobjects.dev/QCObjects.js"></
↪script>
        <script type="text/javascript">
            var canvas1,canvas2,canvas3,container;
        CONFIG.set('relativeImportPath','src/');

            /**
             * Main import sentence.
             */
            Import('cl.quickcorp',function (){

                /**
                 * Super Container MyOwnBody
                 */
                Class('MyOwnBody',HTMLBodyElement,{
                    customAttr:'custom',
                    body:document.body  // breakes default body element and replace␣
↪with them
                });

                /**
                 * Another custom class definition
                 */
                Class('MyContainer',HTMLElement,{
                    width:400,
                    height:400,
                    customAttr:'custom attr container'
                });
```

(continues on next page)

(continued from previous page)

```
                /**
                 * Another custom class definition
                 */
                Class('canvas',HTMLCanvasElement,{
                    customAttr:'custom'
                });

                /**
                 * Another custom class definition
                 */
                Class('MyCanvas2',HTMLCanvasElement,{});

                body = New(MyOwnBody); // binds to body
                body.css({backgroundColor:'#ccc'});

                container = document.getElementsByTagName('container')[0].
↪Cast(MyContainer); // cast any javascript dom object to QC_Object class
                container.css({backgroundColor:'red'}); // access binding in two␣
↪directions to dom objects

                /**
                 * Instance a new custom canvas
                 */
                canvas1 = New(canvas,{
          width:100,
          height:100,
        });
                canvas2 = New(canvas,{
          width:200,
                    height:100,
        });
                canvas3 = New(canvas,{
          width:300,
                    height:50,
        });

                canvas1.css({backgroundColor:'#000000'}); // like jquery and another␣
↪style access
        canvas1.body.style.backgroundColor='#000000'; // standard javascript style␣
↪access
                canvas2.body.style.backgroundColor='#0044AA'; // standard javascript␣
↪style access
                canvas3.body.style.backgroundColor='green'; // standard javascript␣
↪style access

                canvas1.append(); //append canvas1 to body
                canvas2.attachIn('container'); // attach or append to specific tag␣
↪containers
                container.append(canvas3); // append canvas3 to custom tag binding

//              canvas1.body.remove(); // remove canvas1 from dom
                body.append(canvas3); // append canvas3 to body

        // using components
        var c1 = New(Component,{'templateURI':'templatesample.html',cached:false});
        document.body.append(c1); // appends the c1 to the body
```

(continues on next page)

(continued from previous page)

```
            });

        </script>
    </head>
    <body>
        <container id="contentLoader" ></container>
    </body>
</html>
```

CHAPTER 10

Fork

Please fork this project or make a link to this project into your README.md file. Read the LICENSE.txt file before you use this code.

CHAPTER 11

Become a Sponsor

If you want to become a sponsor for this wonderful project you can do it here

CHAPTER 12

Check out the QCObjects SDK

You can check out the QCObjects SDK and follow the examples to make your own featured components

CHAPTER 13

Donate

If you like this code please DONATE!

CHAPTER 14

Installing

CHAPTER 15

Using QCObjects with Atom:

```
> apm install qcobjects-syntax
```

https://atom.io/packages/qcobjects-syntax

CHAPTER 16

Using QCObjects in Visual Studio Code:

https://marketplace.visualstudio.com/items?itemName=Quickcorp.QCObjects-vscode

CHAPTER 17

Installing with NPM:

```
> npm install qcobjects-cli -g && npm install qcobjects --save
```

Fig. 1: screenshot2

CHAPTER 18

Installing the docker playground:

```
docker pull -a quickcorp/qcobjects && docker run -it --name qcobjects-playground --rm␣
↪-it quickcorp/qcobjects
```

Fig. 1: screenshot3

CHAPTER 19

Using the code in the straight way into HTML5:

```
<script type="text/javascript" src="https://qcobjects.dev/QCObjects.js"></script>
```

CHAPTER 20

Reference

CHAPTER 21

Essentials

Here are the essentials symbols and concepts of QCObjects Reference

Basic Type of all elements

With **CompletStorageCache** you can handle a cache for any object and save it in the local storage.

```
var cache = new CompletStorageCache({
                      index:object.id, // Object Index
                      load:(cacheController)=>{}, // A function to execute for the␣
↪first time
                      alternate: (cacheController)=>{} // The alternate function to␣
↪execute from the second time the source coude is loaded
                      });
```

```
var dataObject = {id:1,
                  prop1:1,
                  prop2:2
                };

var cache = new ComplexStorageCache({
    index: dataObject.id,
    load: (cacheController) => {
      dataObject = {
               id:dataObject.id,
               prop1:dataObject.prop1*2, // changing a property value
               prop2:dataObject.prop2
             };
      return dataObject;
    },
    alternate: (cacheController) => {
      dataObject = cacheController.cache.getCached(dataObject.id); // setting␣
↪dataObject with the cached value
      return;
    }
  });
```

(continues on next page)

(continued from previous page)

```
// Next time you can get the object from the cache
var dataObjectCopyFromCache = cache.getCached(dataObject.id);
console.log(dataObjectCopyFromCache); // will show the very same object value than␣
↪dataObject
```

The **asyncLoad** function loads a code once in async mode. This is useful to asure some initial process don't replicate its execution and aren't loaded after sensitive code.

```
asyncLoad(()=>{
  // my code here
},args);
// Where args is an array of arguments, it can be the "arguments" special object
```

```
let doSomething = (arg1,arg2)=>{
  asyncLoad((arg1,arg2)=>{
    console.log(arg1);
    console.log(arg2);
  },arguments);
};

doSomething(1,2); // the code of doSomething will be executed once after the rest of␣
↪asyncLoad queue of functions and before the execution of Ready event.
```

This is NOT the class definition of ECMAScript 2015 (see class ECMAScript 2015 for reference).

Class is a special function to help you to declare a class in an easier and compatible way. It works cross-browser, and I hope ECMA could adopt something like that in the future. To let javascript not to be confuse about this, QCObjects uses "Class" not "class" (note the Camel Case).

```
Class('MyClassName',MyClassDefinition);
```

Where **MyClassDefinition** is an object with a QCObjects **prototype**

```
Class('MyClassName',InheritClass,{
  propertyName1:0, // just to declare purpose
  propertyName2:'',
  classMethod1: function (){
    // some code here
    // note you can use "this" object
    return this.propertyName1;
  },
  classMethod2: function () {
    // some code here
    return this.propertyName2;
  }
});

var newObject = New(MyClassName,{
    propertyName1:1, // this initializes the value in 1
    propertyName2:"some value"
});
console.log(newObject.classMethod1()); // this will show number 1
console.log(newObject.classMethod2()); // this will show "some value"
```

This is a special method inserted to make your life easier when you want to dynamically manipulate the **DOM**. You can insert even a **Component**, a **QCObjects** Object or a **DOM** Element inside another **HTMLElement**.

```
[element].append([object or element]);
```

```
// This will create a QCObjects class named "canvas" extending a HTMLCanvasElement␣
↪with a customAttr property that has a "custom" value
Class('canvas',HTMLCanvasElement,{
  customAttr:'custom'
});

// This will declare an instance canvas1 from the class canvas
let canvas1 = New(canvas,{
            width:100,
            height:100,
          });

// This will append the canvas1 object to the document body
document.body.append(canvas1);
```

When you extend a QCObjects class from another one, you can use _super_ method to get an instance from the main class definition.

```
_super_('MySuperClass','MySuperMethod').call(this,params)
// where this is the current instance and params are method parameters
```

```
Class('MySuperiorClass',InheritClass,{
  propertyName1:0, // just to declare purpose
  propertyName2:'',
  classMethod1: function (){
    // some code here
    // note you can use "this" object
    return this.propertyName1;
  },
});

Class('MyClassName',MySuperiorClass,{
  propertyName1:0, // just to declare purpose
  propertyName2:'',
  classMethod2: function () {
    // The next line will execute classMethod1 from MySuperiorClass
    // but using the current instance of MyClassName1
    return _super_('MySuperiorClass','classMethod1').call(this);
  }
});

var newObject = New(MyClassName,{
    propertyName1:1, // this initializes the value in 1
    propertyName2:"some value"
});
console.log(newObject.classMethod2()); // this will show the number 1
```

Creates an object instance of a QCObjects class definition.

```
let objectInstance = New(QCObjectsClassName, properties);
// where properties is a single object with the property values
```

NOTE: In the properties object you can use single values or getter as well but they will be executed once.

```
Class('MyCustomClass',Object);
let objectInstance = New(MyCustomClass,{
  prop1:1,
  get randomNumber(){ // this getter will be executed once
    return Math.random();
  }
});

console.log(objectInstance.randomNumber); // it will show console.log(objectInstance.
↪prop1); // it will show number 1
```

A single common used QCObjects class definition.

With _Crypt you can encode serializable objects by a passphrase

```
var _string = New(_Crypt,{string:'hello world',key:'some encryption md5 key'});
console.log(_string._encrypt());
console.log(_string._decrypt()); // decodes encrypted string to the source
```
Example (2):

```javascript
_Crypt.encrypt('hola mundo','12345678866');
_Crypt.decrypt('nqCelFSiq6Wcpw==','12345678866');
```

GLOBAL is a special QCObjects class to reach the global scope. It has a set and a get method to help you to manage the internal GLOBAL properties.

```
GLOBAL.set('globalProperty1','some value in global scope');
var globalProperty1 = GLOBAL.get('globalProperty1');
```

CONFIG is a smart class that manages the global settings of your application. You can get the properties either from a config.json or from the memory previously saved by a set() call.

1.- In your initial code set the CONFIG initial values:

```
CONFIG.set('someSettingProperty','some initial value');
```

2.- Then you can access it from anywhere in your code by using the get method:

```
var someSettingProperty = CONFIG.get('someSettingProperty');
```

1.- You need to indicate first that you are using a config.json file by setting the "useConfigService" value to true

```
CONFIG.set('useConfigService',true); // using config.json for custom settings config
```

2.- Once you have set the value above QCObjects will know and look to the next CONFIG settings into the file config.json in the basePath folder of your application.

There is also a way to use an encrypted config.json file in order to protect your settings robots that can steal unprotected data from your web application (like API keys web crawlers).

To encrypt your json file go to https://config.qcobjects.dev, put your domain and the config.json content. The tool will encrypt your json and you can copy the encrypted content to insert it in your config.json file. QCObjects will know the data is encrypted and the process to decode the data will be transparent for you.

waitUntil is a helper just in case you are in trouble trying to run a code before a condition is true. The code inside waitUntil will be executed once.

NOTE: This is useful in some cases but an excessive use is not recommended.

```
waitUntil(()=>{
  // the code that will be executed after the condition is true
},()=>{return condition;});
// where condition is what I want to wait for
```

```
let someVar = 0;
waitUntil(()=>{
  console.log('someVar is present');
},()=>{return typeof someVar != 'undefined';});
// where condition is what I want to wait for
```

Defines a QCObjects package and returns it.

```
Package('packageName',[packageContent]);
```

Where packageContent is an array of QCObjects Classes. If you only pass the packageName param you will get the previously declared package content.

```
'use strict';
Package('org.quickcorp.main',[
  Class('Main',InheritClass,{
    propertyName1:'propertyValue1',
  }),
  Class('MyCustomClass',InheritClass,{
    propertyName2:'propertyValue2',
  }),
]);
```

```
let mainPackage = Package('org.quickcorp.main'); // this will return the previously␣
↪declared content of package 'org.quickcorp.main'
// mainPackage[0] will be the Main class definition.
// This is useful for code introspection
```

Imports a package from another JS file

```
Import (packagename,[ready],[external]);
```

Where packagename is the name of the package, ready is a function that will be executed after the package is loaded, and external is a boolean value that indicates if the JS file is in the same origin or it is from another external resource.

```
Import('org.quickcorp.main');
```

The above code will try to import a JS fila named 'org.quickcorp.main.js' from the path specified in the **relativeImportPath** settings value present in your **CONFIG**. Inside the JS file you have to define a package by using Package('org.quickcorp.main',[Class1, Class2...])

```
Import('org.quickcorp.main',function (){
  console.log('remote import is loaded');
},true);
```

The above code this time is trying to load the same package but using an external path defined by the **remoteImportsPath** setting present in your **CONFIG**

NOTE: In both examples above you have not use or specify the ".js" extension. This it's used by default and can't be changed by security reasons.

Put a symbol (var or function) in the global scope.

```
Export('name of symbol');
```

```
(()=>{
  // this is local scope
  let someFunction = (someLocalParam)=>{
    console.log(someLocalParam);
  };
  Export(someFunction); // now, someFunction is in the top level scope.
})();


// this is the top level scope
someFunction('this works');
```

Use the Cast method of any DOM element to get the properties of another type of object. This is useful to transform an object type to another giving more flexibility in your code.

```
let resultObject = [element or QCObjects type].Cast(objectToCastFrom);
```

Where objectToCastFrom is an object to get the properties from and put it into the result object returned by Cast.

```
Class('MyOwnClass',{
  prop1:'1',
  prop2:2
});

let obj = document.createElement('div').Cast(MyOwnClass);
```

The above code will create a DOM object and Cast it to MyOwnClass. Because of MyOwnClass is a QCObjects type class, obj will now have a prop1 and prop2 properties, and will now be a QCObjects object instance with a body property that is a div element.

Tag is a useful function to select any DOM element using selectors. Tag will always return a list of elements, that you can map, sort, and filter as any other list.

```
var listOfElements = Tag(selector);
```

Where selector is a DOM query selector.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Demo</title>
        <script type="text/javascript" src="https://qcobjects.dev/QCObjects.js"></
↪script>
    </head>
    <body>
    <div class="myselector">
    <p>Hello world</p>
    </div>
    <script>
    Ready(()=>{
      Tag('.myselector > p').map((element)=>{
        element.innerHTML = 'Hello world! How are you?';
      });
    });
    </script>
```

(continues on next page)

(continued from previous page)

```
    </body>
</html>
```

In the above code, a paragraph element was created inside a div with a css class named myselector by html, and then is modified dynamically using the QCObjects Tag function. If you are familiar with query selector frameworks like JQuery, you will love this one.

Assign a function to run after everything is done by QCObjects and after the window.onload event. Use it to prevent 'undefined' DOM objects error.

```
Ready(()=>{
  // My init code here!
});
```

Note that if you define dynamic components by using a HTML "component" tag, the dynamic content load will not trigger Ready events. To catch code everytime a dynamic component is loaded, use a Controller done method instead.

You will use Ready implementation mostly when you want to implement QCObjects in conjunction with another framework that needs it.

A QCObjects class type for components.

[Component].domain Returns a string with the domain of your application. It is automatically set by QCObjects at the load time.

[Component].basePath Returns a string with the base path url of your application. It is automatically set by QCObjects at the load time.

NOTE: If you want to change the components base path, you have to use *CONFIG.set('componentsBasePath','new path relative to the domain')* in your init code.

[Component].templateURI Is a string representing the component template URI relative to the domain. When is set, the component will load a template and append the inner content into the body childs as a part of the DOM. To set this property, it is recommended to use the ComponentURI helper function.

[Component].tplsource Is a string representing the source where the template will be loaded. It can be "default" or "none". A value of "default" will tell QCObjects to load the template from the templateURI content. A value of "none" will tell QCObjects not to load a template from anywhere.

[Component].url Is a string representing the entire url of the component. It is automatically set by QCObjects when the component is instantiated.

[Component].name Is a string representing the name of a component. The name of a component can be any alphanumeric value that identifies the component type. It will be internally used by ComponentURI to build a normalised component template URI.

[Component].method Is a string representing a HTTP or HTTPS method. By default, every component is set to use the "GET" method. In the most of cases, you don't need to change this property.

[Component].data Is an object representing the data of the component. When QCObjects loads a template, it will get every property of data object and bind it to a template label representing the same property inside the template content between double brakets (example: {{prop1}} in the template content will represent data.prop1 in the component instance). NOTE: To refresh the data bindings it is needed to rebuild the component (see the use of [Component].rebuild() method for more details).

[Component].reload Is a boolean value that tells QCObjects when to force reload the content of a component from the template or not. If its value is true, the template content will be replacing the current DOM childs of the component body element. If its value is false, the template content will be added after the las component body child.

[Component].cached Is a boolean value that tells QCObjects if the component needs to be cached or not. When a component is cached, the template content loaded from templateURI will be loaded once. You can set this property either as a static property of the Component Class to set the default value for every next component object instance, or setting the individual value of the property in every component definition. In a world where the performance matters, to give more flexibility to the cache behaviour is needed more than ever.

[Component].routingWay Returns a string representing the routing way. Its value can be "hash", "pathname" or "search". NOTE: To change the routingWay of every component it is recommended to use CONFIG.set('routingWay','value of a valid routing way') in your init code.

[Component].validRoutingWays Returns a list representing the valid routing ways. QCObjects uses this to internally validate the routingWay which was used to build the component routings.

[Component].routingNodes Returns a NodeList object representing the list of nodes that were loaded by the component routing builder.

[Component].routings Returns a list with the component routings built when the component was instantiated.

[Component].routingPath Returns a string representing the current routing path

[Component].routingSelected Returns an object representing the current routing of the component

[Component].subcomponents Returns a list of components that are childs of the component instance.

[Component].body Is a DOM element representing the body of the component. NOTE: Every time a component body is set, it will trigger the routings builder for this component.

[Component].set('prop',value) Sets a value for a component property.

[Component].get('prop') Returns the value of a component property

[Component].rebuild() Rebuilds the component. It will force a call for the componentLoader with this component when it's needed.

[Component].Cast(ClassName or ComponentClassName) Returns the cast of a component definition into another one. This is useful to dynamically merge components definitions.

[Component].route() Forces the component routings builder to reload the routings of the component. This will result in a rebuild call when it's needed.

[Component].fullscreen() Puts the component in fullscreen mode.

[Component].closefullscreen() Closes the fullscreen mode.

[Component].css(css object) Sets the css properties for the component.

[Component].append(component or QCObjects object) Appends a component as a child of the current component body

[Component].attachIn(selector) Attaches a current component body to any element in the given selector.

Is a HTML tag representation of a component instance. Every declaration of a `<component></component>` tag will generate a related instance of a QCObjects component. While a component tag is not an instance itself, you can even define some instance properties by setting the related tag attribute when it is available.

Below is a list of the available attributes for a component tag

''<component name>'' Sets the name of the related component instance built by QCObjects.

Usage:

```
<component name="name_of_component"></component>
```

Example:

```
<!-- index.html -->
<!DOCTYPE html>
<html>
    <head>
        <title>Demo</title>
        <script type="text/javascript" src="https://qcobjects.dev/QCObjects.js"></
↪script>
    </head>
    <body>
      <!-- this will load the contents of ./templates/main[.tplextension] file -->
      <component name="main"></component>
    </body>
</html>
```

‘‘<component cached>‘‘ Sets the cached property if the related instance of a component.

NOTE: Only a value of “true” can be set in order to tell QCObjects that the component template content has to be cached. Any other value will be interpreted as false.

Usage:

```
<component name="name_of_component" cached="true"></component>
```

‘‘<component data-property1 data-property2 ...>‘‘ Sets a static value of a property for the data object in the component instance.

NOTE: Data property tag declaration was thought with the purpose to give some simple way to mocking a dynamic component with template assignments. Don’t use it thinking it is a bidirectional way data binding. While you can get a bidirectional way behaviour accesing a data object from a component instance, it is not the same for the component tag. Data property declaration in component tags is only one way data binding because of components tree architecture.

‘‘<component controllerClass>‘‘ Defines a custom Controller Class for the component instance

Usage:

```
<component name="name_of_component" controllerClass="ControllerClassName"></component>
```

‘‘<component viewClass>‘‘ Defines a custom View Class for the component instance

Usage:

```
<component name="name_of_component" viewClass="ViewClassName"></component>
```

‘‘<component componentClass>‘‘ Defines a custom Component Class for the component instance

Usage:

```
<component name="name_of_component" componentClass="ComponentClassName"></component>
```

‘‘<component effectClass>‘‘ Defines a custom Effect Class for the component instance

Usage:

```
<component name="name_of_component" effectClass="EffectClassName"></component>
```

‘‘<component template-source>‘‘ Sets the tplsource property of the related instance of a component. Possible values are “none” or “default”.

Usage:

```
<component name="name_of_component" template-source="none"></component>
```

‘‘<component tplextension>‘‘ Sets the tplextension property of the related instance of a component. Possible values are any file extension. Default value is “html”

Usage:

```
<component name="name_of_component" tplextension="tpl.html"></component>
```

Is a helper function to let you define the templateURI for a component in a normalised way.

```
var templateURI = ComponentURI({
  'COMPONENTS_BASE_PATH':CONFIG.get('componentsBasePath'),
  'COMPONENT_NAME':'main',
  'TPLEXTENSION':"tpl.html",
  'TPL_SOURCE':"default"
});

console.log(templateURI); // this will show something like "templates/components/main.
↪tpl.html" depending on your CONFIG settings
```

Loads a component instance in a low level, and appends the component template content to the component body. In the most of cases you won’t need to call componentLoader in order to load a component. This is automatically called by QCObjects when it’s needed. componentLoader returns a promise that is resolved when the component load is done and rejected when the component load was failed.

```
[Promise] componentLoader(componentInstance,load_async)
```

Where componentInstance is a component instance created by *‘‘New(ComponentDefinitionClass)‘‘*

```
componentLoader(componentInstance,load_async).then(
  (successStandardResponse)=>{
    // component load successful
    var request = successStandardResponse.request;
    var component = successStandardResponse.component;
  },(failStandardResponse)=>{
    // component load failed
    var component = failStandardResponse.component;
  });
```

Rebuilds every component that is a child element of the DOM element who owns the method. In the most of cases, you won’t need to call buildComponents in order to build or rebuild every component in the DOM. This is automatically called by QCObjects when it’s needed.

```
[element].buildComponents()
```

```
document.buildComponents()
```

A built-in QCObjects Class to define a controller

A built-in QCObjects View to define a view

A built-in QCObjects Class to define a value object

A QCObjects class type for services.

‘Service <#service>‘__.domain Returns a string with the domain of your application. It is automatically set by QCObjects at the load time.

'Service <#service>'__.basePath Returns a string with the base path url of your application. It is automatically set by QCObjects at the load time.

'Service <#service>'__.url Is a string representing the entire url of the service. It can be absolute or relative to the basePath when it applies. It can be also an external url.

NOTE: To load a service of an external resource you need to specify the external parameter to true using serviceLoader.

'Service <#service>'__.name Is a string representing the name of a component. The name of a service can be any alphanumeric value that identifies the service instance. It isn't a unique ID but only a descriptive name.

'Service <#service>'__.method Is a string representing a HTTP or HTTPS method. Possible values are: "GET", "POST", "PUT", ... any other that is accepted by REST services calls.

'Service <#service>'__.data Is an object representing the data of the service. When QCObjects loads a service. It receives the response and interpretes it as a template. So once a service response is obtained, it will get every property of data object and bind it to a template label representing the same property inside the template content between double brakets (example: {{prop1}} in the template content will represent data.prop1 in the service instance).

'Service <#service>'__.cached Is a boolean value that tells QCObjects if the service response needs to be cached or not. When a service is cached, the template content loaded from the service url will be loaded only once. You have to set this value to false for every Service instance you define in order to asure the service is loaded from the resource but not the storage cache.

'Service <#service>'__.set('prop',value) Sets a value for a service property.

'Service <#service>'__.get('prop') Returns the value of a service property

Loads a service instance and returns a promise that is resolved when the service has a successful response load and is rejected when it fails loading the response.

```
[Promise] serviceLoader(serviceInstance)
```

```
Class('MyTestService',Service,{
    name:'myservice',
    external:true,
    cached:false,
    method:'GET',
    headers:{'Content-Type':'application/json'},
    url:'https://api.github.com/orgs/QuickCorp/repos',
    withCredentials:false,
    _new_:()=>{
      // service instantiated
    },
    done:()=>{
      // service loaded
    }
});
var service = serviceLoader(New(MyTestService,{
  data:{param1:1}
})).then(
  (successfulResponse)=>{
    // This will show the service response as a plain text
    console.log(successfulResponse.service.template);
  },
  (failedResponse)=>{

  });
```

Is a built-in definition for a JSON Service Class

'JSONService <#jsonservice>'__.domain Returns a string with the domain of your application. It is automatically set by QCObjects at the load time.

'JSONService <#jsonservice>'__.basePath Returns a string with the base path url of your application. It is automatically set by QCObjects at the load time.

'JSONService <#jsonservice>'__.url Is a string representing the entire url of the service. It can be absolute or relative to the basePath when it applies. It can be also an external url.

NOTE: To load a service of an external resource you need to specify the external parameter to true using serviceLoader.

'JSONService <#jsonservice>'__.name Is a string representing the name of a component. The name of a service can be any alphanumeric value that identifies the service instance. It isn't a unique ID but only a descriptive name.

'JSONService <#jsonservice>'__.method Is a string representing a HTTP or HTTPS method. Possible values are: "GET", "POST", "PUT", ... any other that is accepted by REST services calls.

'JSONService <#jsonservice>'__.data Is an object representing the data of the service. When QCObjects loads a service. It receives the response and interpretes it as a template. So once a service response is obtained, it will get every property of data object and bind it to a template label representing the same property inside the template content between double brakets (example: {{prop1}} in the template content will represent data.prop1 in the service instance).

'JSONService <#jsonservice>'__.cached Is a boolean value that tells QCObjects if the service response needs to be cached or not. When a service is cached, the template content loaded from the service url will be loaded only once. You have to set this value to false for every Service instance you define in order to asure the service is loaded from the resource but not the storage cache.

'JSONService <#jsonservice>'__.set('prop',value) Sets a value for a service property.

'JSONService <#jsonservice>'__.get('prop') Returns the value of a service property

```
Class('MyTestJSONService',JSONService,{
    name:'myJSONservice',
    external:true,
    cached:false,
    method:'GET',
    withCredentials:false,
    url:'https://api.github.com/orgs/QuickCorp/repos',
    _new_:function (){
      // service instantiated
      delete this.headers.charset; // do not send the charset header
    },
    done:function (result){
      _super_('JSONService','done').call(this,result);
    }
});
var service = serviceLoader(New(MyTestJSONService,{
  data:{param1:1}
})).then(
  (successfulResponse)=>{
    // This will show the service response as a JSON object
    console.log(successfulResponse.service.JSONresponse);
  },
  (failedResponse)=>{

  });
```

Is a built-in Class definition to load the CONFIG settings from a config.json file

```
// To set the config.json file relative url
ConfigService.configFileName='config.json'; // it is done by default
CONFIG.set('useConfigService',true); // using config.json for custom settings config
```

CHAPTER 22

SDK

22.1 Quick Start

CHAPTER 23

Step 1: Start creating a main import file and name it like: cl.quickcorp.js. Put it into packages/js/ file directory

```
"use strict";
/*
* QuickCorp/QCObjects is licensed under the
* GNU Lesser General Public License v3.0
* [LICENSE] (https://github.com/QuickCorp/QCObjects/blob/master/LICENSE.txt)
*
* Permissions of this copyleft license are conditioned on making available
* complete source code of licensed works and modifications under the same
* license or the GNU GPLv3. Copyright and license notices must be preserved.
* Contributors provide an express grant of patent rights. However, a larger
* work using the licensed work through interfaces provided by the licensed
* work may be distributed under different terms and without source code for
* the larger work.
*
* Copyright (C) 2015 Jean Machuca,<correojean@gmail.com>
*
* Everyone is permitted to copy and distribute verbatim copies of this
* license document, but changing it is not allowed.
*/


Import ('external/libs');
Import ('cl.quickcorp.model');
Import ('cl.quickcorp.components');
Import ('cl.quickcorp.controller');
Import ('cl.quickcorp.view');

Package('cl.quickcorp',[
  Class('FormValidator',Object,{

  }),
]);
```

CHAPTER 24

Step 2: Then create some services inhereting classes into the file js/packages/cl.quickcorp.services.js :

```
"use strict";
/*
* QuickCorp/QCObjects is licensed under the
* GNU Lesser General Public License v3.0
* [LICENSE] (https://github.com/QuickCorp/QCObjects/blob/master/LICENSE.txt)
*
* Permissions of this copyleft license are conditioned on making available
* complete source code of licensed works and modifications under the same
* license or the GNU GPLv3. Copyright and license notices must be preserved.
* Contributors provide an express grant of patent rights. However, a larger
* work using the licensed work through interfaces provided by the licensed
* work may be distributed under different terms and without source code for
* the larger work.
*
* Copyright (C) 2015 Jean Machuca,<correojean@gmail.com>
*
* Everyone is permitted to copy and distribute verbatim copies of this
* license document, but changing it is not allowed.
*/


Package('cl.quickcorp.service',[
  Class('JsonService',{
    method:"GET",
    cached:false,
    headers: {
        "Content-Type":"application/json",
      "charset":"utf-8"
    },
    JSONresponse: null,
    done:function(result){
      console.log("***** RECEIVED RESPONSE:");
```

(continues on next page)

(continued from previous page)

```
      console.log(result.service.template);
      this.JSONresponse = JSON.parse(result.service.template);
      alert(this.template);
    }
  }),
  Class('FormSubmitService',{
    method:"POST",
    cached:false,
    headers: {
      "Content-Type":"application/json"
    },
    JSONresponse: null,
    done: function(result) {
      console.log("***** CALLED FormSubmitService");
      this.JSONresponse = JSON.parse(result.service.template);
      //TODO success case
      console.log("***** SUCCESS!")
      console.log(this.JSONresponse);
    },
    fail: function(result) {
      //TODO negative case
      console.log("***** ERROR");
    }

  })

])
```

CHAPTER 25

Step 3: Now it's time to create the components (cl.quickcorp.components.js)

```
"use strict";
/*
* QuickCorp/QCObjects is licensed under the
* GNU Lesser General Public License v3.0
* [LICENSE] (https://github.com/QuickCorp/QCObjects/blob/master/LICENSE.txt)
*
* Permissions of this copyleft license are conditioned on making available
* complete source code of licensed works and modifications under the same
* license or the GNU GPLv3. Copyright and license notices must be preserved.
* Contributors provide an express grant of patent rights. However, a larger
* work using the licensed work through interfaces provided by the licensed
* work may be distributed under different terms and without source code for
* the larger work.
*
* Copyright (C) 2015 Jean Machuca,<correojean@gmail.com>
*
* Everyone is permitted to copy and distribute verbatim copies of this
* license document, but changing it is not allowed.
*/
Package('cl.quickcorp.components',[
  Class('FormField',Component,{
    cached:false,
    reload:true,
    createBindingEvents:function (){
        var _executeBinding = this.executeBinding;
        var thisobj = this;
        var _objList = this.body.querySelectorAll(this.fieldType);
        for (var _datak=0;_datak<_objList.length;_datak++){
            var _obj = _objList[_datak];
            _obj.addEventListener('change',function(e){
                logger.debug('Executing change event binding');
                thisobj.executeBindings();
```

(continues on next page)

(continued from previous page)

```
            });
            _obj.addEventListener('keydown',function(e){
                logger.debug('Executing keydown event binding');
                    thisobj.executeBindings();
            });
        }
    },
    executeBinding:function (_obj){
        var _datamodel = _obj.getAttribute('data-field');
        logger.debug('Binding '+_datamodel+' for '+this.name);
        this.data[_datamodel]=_obj.value;
    },
    executeBindings:function (){
      var _objList = this.body.querySelectorAll(this.fieldType);
        for (var _datak=0;_datak<_objList.length;_datak++){
            var _obj = _objList[_datak];
            var _datamodel = _obj.getAttribute('data-field');
        logger.debug('Binding '+_datamodel+' for '+this.name);
            this.data[_datamodel]=_obj.value;
        }
    },
    done:function (){
      var thisobj = this;
      thisobj.executeBindings();
        thisobj.createBindingEvents();
      logger.debug('Field loaded: '+thisobj.fieldType+'[name='+thisobj.name+']');
    }
  }),
  Class('ButtonField',FormField,{
    fieldType:'button'
  }),
  Class('InputField',FormField,{
    fieldType:'input'
  }),
  Class('TextField',FormField,{
    fieldType:'textarea'
  }),
  Class('EmailField',FormField,{
    fieldType:'input'
  })
]);
```

CHAPTER 26

Step 4: Once you have done the above components declaration, you will now want to code your controllers (cl.quickcorp.controller.js)

```
"use strict";
/*
* QuickCorp/QCObjects is licensed under the
* GNU Lesser General Public License v3.0
* [LICENSE] (https://github.com/QuickCorp/QCObjects/blob/master/LICENSE.txt)
*
* Permissions of this copyleft license are conditioned on making available
* complete source code of licensed works and modifications under the same
* license or the GNU GPLv3. Copyright and license notices must be preserved.
* Contributors provide an express grant of patent rights. However, a larger
* work using the licensed work through interfaces provided by the licensed
* work may be distributed under different terms and without source code for
* the larger work.
*
* Copyright (C) 2015 Jean Machuca,<correojean@gmail.com>
*
* Everyone is permitted to copy and distribute verbatim copies of this
* license document, but changing it is not allowed.
*/
"use strict";
Package('cl.quickcorp.controller',[
    Class('MainController',Object,{
        _new_:function (){
            //TODO: Implement
            logger.debug('MainController Element Initialized');
        }
    }),
    Class('MyAccountController',Object,{
        component: null,
        done:function (){
            var controller = this;
```

(continues on next page)

(continued from previous page)

```
            logger.debug('MyAccountController Element Initialized');
            this.component.body.setAttribute('loaded',true);

        },
        _new_:function (o){
            //TODO: Implement
            this.component = o.component;

        }
    }),
]);
```

CHAPTER 27

Step 5: To use into the HTML5 code you only need to do some settings between script tags:

```
<script>
CONFIG.set('relativeImportPath','js/packages/');
CONFIG.set('componentsBasePath','templates/components/');
CONFIG.set('delayForReady',1); // delay to wait before executing the first ready␣
↪event, it includes imports
CONFIG.set('preserveComponentBodyTag',false); // don't use <componentBody></
↪componentBody> tag

Import('cl.quickcorp'); # this will import your main file: cl.quickcorp.js into js/
↪packages/ file path
</script>
```