JasperReports® Server

Visualize.js Guide

Version 9.0.0 | January 2024

cloud SOFTWARE GROUP | Copyright © 2005-2024. Cloud Software Group, Inc. All Rights Reserved.

Contents

Contents 2
API Reference - Visualize.js 6
Requesting the Visualize.js Script 7
Contents of the Visualize.js Script 8
Usage Patterns 9
Testing Your JavaScript 10
Changing the Look and Feel 10
Cross-Domain Security 12
Serving Visualize.js from a CDN 15
API Reference - login and logout 18
Authentication Properties 18
Authentication Functions 19
Login With Plain Text Credentials 20
Login With SSO Token 21
Logging Out 22
Login With Hooks 22
UI for Login/Logout 24
UI for Login/Logout With SSO Token 25
Sharing Credentials Among Calls 25
Using Visualize.js Without Authentication 26
API Reference - resourcesSearch 27
Search Properties 27
Search Functions 28
Finding Resources 30
Reusing a Search Instance 31
Reusing Search Results 32
Discovering Available Types 32

API Reference - report ... 33
Report Properties ... 34
Report Functions ... 36
Report Structure ... 39
Rendering a Report ... 40
Getting the Embed Code of a Report ... 42
Setting Report Parameters ... 44
Saving a Report ... 45
Rendering Multiple Reports ... 45
Resizing a Report ... 47
Setting Report Pagination ... 48
Creating Pagination Controls (Next/Previous) ... 49
Creating Pagination Controls (Range) ... 50
Exporting From a Report ... 51
Exporting Data From a Report ... 53
Refreshing a Report ... 54
Canceling Report Execution ... 55
Discovering Available Charts and Formats ... 56
API Reference - inputControls ... 57
Input Control Properties ... 58
Input Control Functions ... 58
Embedding Input Controls ... 59
Handling Input Control Events ... 61
Resetting Input Control Values ... 64
Embedded Input Control Styles ... 65
Custom Input Controls ... 66
API Reference - dashboard ... 70
Dashboard Properties ... 71
Dashboard Functions ... 72
Dashboard Structure ... 75
Rendering a Dashboard ... 77
Getting the Embed Code of a Dashboard ... 77

- Refreshing a Dashboard ... 79
- Tracking Completion Status ... 79
- Using Dashboard Input Controls ... 80
- Using the Dashboard Undo Stack ... 83
- Setting Dashboard Hyperlink Options ... 86
- Exporting From a Dashboard ... 90
- Closing a Dashboard ... 93

API Reference - adhocView ... 94

- Ad Hoc View Properties ... 94
- Ad Hoc View Functions ... 96
- Ad Hoc View Data Structure ... 97
- Rendering an Ad Hoc View ... 98
- Getting the Embed Code of an Ad Hoc View ... 100
- Setting the Visualization Type ... 102
- Setting Ad Hoc View Filters ... 103
- Accessing Ad Hoc View Hyperlinks ... 104

API Reference - Errors ... 109

- Error Properties ... 110
- Common Errors ... 110
- Catching Initialization and Authentication Errors ... 112
- Catching Search Errors ... 112
- Validating Search Properties ... 113
- Catching Report Errors ... 113
- Catching Input Control Errors ... 114
- Validating Input Controls ... 114

API Usage - Report Events ... 115

- Tracking Completion Status ... 115
- Tracking Report Container Size ... 115
- Listening for Page Totals ... 116
- Listening for the Last Page ... 117
- Customizing a Report's DOM Before Rendering ... 117

API Usage - Hyperlinks ... 118

Structure of Hyperlinks 119
Customizing Links 120
Drill-Down in Separate Containers 121
Accessing Data in Links 122
API Usage - Interactive Reports 124
Interacting With JIVE UI Components 124
Using Floating Headers 127
Changing the Chart Type 127
Changing the Chart Properties 129
Undo and Redo Actions 131
Sorting Table Columns 133
Filtering Table Columns 134
Formatting Table Columns 136
Conditional Formatting on Table Columns 140
Sorting Crosstab Columns 141
Sorting Crosstab Rows 143
Implementing Search in Reports 144
Providing Bookmarks in Reports 145
Disabling the JIVE UI 146
Visualize.js Tools 147
Checking the Scope in Visualize.js 147
CSS Diagnostic Tool 149
Jaspersoft Documentation and Support Services 157
Legal and Third-Party Notices 159

API Reference - Visualize.js

The JavaScript API exposed through Visualize.js allows you to embed and dynamically interact with reports. With Visualize.js, you can create web pages and web applications that seamlessly embed reports and complex interaction. You can control the look and feel of all elements through CSS and invent new ways to merge data into your application. Visualize.js helps you make advanced business intelligence available to your users.

This chapter contains the following sections:

- Requesting the Visualize.js Script
- Contents of the Visualize.js Script
- Usage Patterns
- Testing Your JavaScript
- Changing the Look and Feel
- Cross-Domain Security
- Serving Visualize.js from a CDN

Each function of Visualize.js is then described in the following chapters:

- API Reference - login and logout
- API Reference - resourcesSearch
- API Reference - report
- API Reference - inputControls
- API Reference - dashboard
- API Reference - Errors

The last chapters demonstrate more advanced usage of Visualize.js:

- API Usage - Report Events
- API Usage - Hyperlinks
- API Usage - Interactive Reports
- Visualize.js Tools

Requesting the Visualize.js Script

The script to include on your HTML page is named visualize.js. It is located on your running instance of JasperReports® Server. Later on your page, you also need a container element to display the report from the script.

```
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
...
<!-- Provide a container for the report -->
<div id="container"></div>
```

The content of visualize.js is type='text/javascript', but that is the default so you usually don't need to include it.

You can specify several parameters when requesting the script:

Parameter	Type or Value	Description
userLocale	locale string	Specify the locale to use for display and running reports. It must be one of the locales supported by JasperReports Server. The default is the locale configured on the server.
logEnabled	true|false	Enable or disable logging. By default, it is enabled (true).
logLevel	debug|info|warn|error	Set the logging level. By default the level is error.
baseUrl	URL	The URL of the JasperReports Server that will respond to visualize requests. By default, it is the same server instance that provides the script.

The following request shows how to use script parameters:

```
<script src="http://bi.example.com:8080/jasperserver-pro/client/
             visualize.js?userLocale=fr&logLevel=debug"></script>
```

Contents of the Visualize.js Script

The Visualize.js script itself is a factory function for an internal JrsClient.

```
/**
 *  Establish connection with JRS instance and generate
 *  ready to use client
 *  @param {Object} properties - configuration to connect to JRS instance
 *  @param {Function} callback - optional, successful callback
 *  @param {Function} errorback - optional, invoked on error
 *  @param {Function} always - optional, invoked always
 */
function visualize(properties, callback, errorback, always){}

/**
 *  Store common configuration, to share them between visualize calls
 *  @param {Object} properties - configuration to connect to JRS instance
 */
function visualize.config(properties);
```

You write JavaScript in a callback that controls what the client does. The following code sample shows the functions of the JrsClient that are available to you:

```
{    /**
     * Perform authentification with provided auth object
     * @param auth {object}  - auth properties
     */
    login : function(auth){},
```

```
    /**
     * Destroy current auth session
     */
    logout : function() {},

    /**
     * Create and run report component with provided properties
     * @param properties {object}  - report properties
     * @returns {Report} report - instance of Report
     */
    report : function(properties){},

    /**
     * Create and run controls for provided controls properties
     * @param properties {object}  - input controls properties
     * @returns {Options} inputControls instance
     *
     */
    inputControls : function(properties){},

    /**
     * Create and run resource search component for provided properties
     * @param properties {object}  - search  properties
     * @returns {Options} resourcesSearch instance
     *
     */
    resourcesSearch : function(properties){}
}
```

These functions are described in the remaining API reference chapters.

Usage Patterns

After specifying the authentication information, you write the callback that will execute inside the client provided by visualize.js.

```
visualize({
    server: "http://bi.example.com",
    auth: {
        name : "joeuser",
        password: "joeuser"
    }
}, function(v){

    //'v' is a client to JRS instance under  "http://bi.example.com"
    //session established for joeuser/joeuser

    var report = v.report(...);

}, function(err){
    alert(err.message);
});
```

The following sample shows how to use the always callback:

```
visualize({
    server: "http://bi.example.com",
    auth: {
        name : "joeuser",
        password: "joeuser"
    }
    }).always(function(err, v) {

        if (err) {
        alert(err.message);
        return;
        }

    v.report(); // use v here

});
```

As of JasperReports Server 7.9, the server can provide the code to embed any report, dashboard, or Ad Hoc view. This means you can browse your server's repository, choose the dashboard or Ad Hoc View that you want to embed, preview it, and then select the </> icon in the toolbar to view the sample code. For more information, see:

- Getting the Embed Code of a Report

- Getting the Embed Code of a Dashboard
- Getting the Embed Code of an Ad Hoc View

Testing Your JavaScript

As you learn to use Visualize.js and write the JavaScript that embeds your reports into your web app, you should have a way to run and view the output of your script.

In order to load Visualize.js , your HTML page containing your JavaScript must be accessed through a web server. Opening a static file with a web browser does not properly load the iframes needed by the script.

One popular way to view your Visualize.js output, is to use the jsFiddle online service. You specify your HTML, JavaScript, and optional CSS in 3 separate frames, and the result displays in the fourth frame.

Another way to test your JavaScript is to use the app server bundled with JasperReports Server. If you deploy the server from the installer with the Apache Tomcat app server, you can create an HTML file at the root of the server web app, for example:

<js-install>/apache-tomcat/webapps/jasperserver-pro/testscript.html

Write your HTML and JavaScript in this file, and then you can run Visualize.js by loading the file through the following URL:

http://mydomain.com:8081/jasperserver-pro/testscript.html

Changing the Look and Feel

When you create a web application that embeds Visualize.js content, you determine the look and feel of your app through layout, styles, and CSS (Cascading Style Sheets). Most of the content that you embed consists of reports and dashboards that you create in JasperReports Server or Jaspersoft® Studio, where you set the appearance of colors, fonts, and layout to match your intended usage.

But some Visualize.js elements also contain UI widgets that are generated by the server in a default style, for example the labels, buttons, and selection boxes for the input controls of a report. In general, the default style is meant to be neutral and embeddable in a wide range of visual styles. If the default style of these UI widgets does not match your app, there are two approaches described in the following sections:

- Customizing the UI with CSS – You can change the appearance of the UI widgets through CSS in your app.
- Customizing the UI with Themes – You can redefine the default appearance of the UI widgets in themes on the server.

Customizing the UI with CSS

The UI widgets generated by the server have CSS classes and subclasses, also generated by the server, that you can redefine in your app to change their appearance. To change the appearance of the generated widgets, create CSS rules that you would add to CSS files in your own web app. To avoid the risk of unintended interference with other CSS rules, you should define your CSS rules with both a classname and a selector, for example:

```
#inputContainer .jr-mInput-boolean-label {
    color: #218c00;
}
```

To change the style of specific elements in the server's generated widgets, you can find the corresponding CSS classes and redefine them. To find the CSS classes, write the JavaScript display the UI widgets, for example input controls, then test the page in a browser. Use your browser's code inspector to look at each element of the generated widgets and locate the CSS rules that apply to it. The code inspector shows you the classes and often lets you modify values to preview the look and feel that you want to create. For more information, see Embedded Input Control Styles.

Customizing the UI with Themes

The UI widgets in Visualize.js elements are generated on the server and their look and feel is ultimately determined by themes. Themes are CSS files defined for organizations, and thus appearances are determined by the user credentials submitted by your Visualize.js instance. Certain visual properties of the UI widgets embedded in your app by Visualize.js can be modified by changing themes on the server. For more information about themes, see the JasperReports Server Administrator Guide.

However, not all properties of a UI widget can be modified through themes. If you want to create a fully custom visualization, you might need to access raw data through the Visualize.js APIs and write your own elements and CSS to display them. For more information, see Exporting Data From a Report.

Cross-Domain Security

After developing your web application and embedding Visualize.js, you will deploy it to your users through some web domain, for example bi-enabled.myexample.com. For testing, JasperReports Server accepts Visualize.js requests from any domain, and your web app will display embedded reports and dashboards. However, once you go into production, you should add security and make sure that the server responds only to Visualize.js requests from your web app. This is called cross-site request forgery (CSRF) protection.

JasperReports Server includes a configurable whitelist of domains, and it only responds to Visualize.js requests from those domains. Add your web domain to this whitelist, and no other web apps on malicious domains can access your server. In addition to whitelisted domains, JasperReports Server always responds to requests that originate from the same domain as the server itself.

To configure the cross-domain whitelist

1. Log in as system administrator (superuser).
2. Select Manage > Server Settings then Server Attributes.
3. The server attribute named domainWhitelist contains a regular expression that matches allowed domains. Set it as follows:
    - When your Visualize.js web app is on another domain, such as in this example, create a regular expression to match the protocol, domain name and port numbers. You can also match multiple subdomains or several port numbers as in this example:
      domainWhitelist = http://*.myexample.com:80\d0
      The server translates this simplified expression into the proper regular expression ^http://.*\.myexample\.com:80\d0$. If you want to avoid the translation, put ^ $ around your value.
    - When your Visualize.js web app is on the same domain as your JasperReports Server set the value to <blank> (no value) so that no other domain has access:
      domainWhitelist = <blank>
    - It is also possible to limit the domains accessing the JasperReports Server for each organization. For advanced configuration details, see the section on cross-domain whitelists in the JasperReports Server Security Guide.

For more information about setting attributes see the JasperReports Server Administrator Guide.

Cross-Site Errors

Even when the domain whitelist is configured, some browsers may limit cross-site access for security reasons. In particular, the Safari browser often blocks access to the Visualize.js script because it uses third-party cookies to enable cross-site access.

There are several workarounds in this case:

- Use a different browser. Other browsers may allow access when Safari does not. However, in the future, other browsers may also begin to restrict access to third-party cookies that enable cross-site access.
- Use a proxy to make JasperReports Server appear to be on the same domain as your website. If you have already embedded Visualize.js in a cross-site way, changing to proxying will require changes to your environment (implementing and configuring a proxy service) and to your application.
- Use the same domain but different sub-domains for your application and JasperReports Server, for example:
    - www.myapp.com for your application
    - jaspersoft.myapp.com for JasperReports Server serving the Visualize.js script

When accessing JasperReports Server application resources from another domain, Chrome will treat it as a CORS request. To make this work, both applications (requestor and JasperReports Server) should communicate via HTTPS protocol:

- Server should use HTTPS.
- Client-side application should also use HTTPS.
- For XMLHttpRequest, you need to set *withCredentials=true*

JasperReports Server will send *Secure;HttpOnly Cookies* attributes.

The Chrome browser accepts cookies only with these attributes (ignoring Path=/jasperserver-pro, so it works for CORS requests).

Enabling Private Network Access

When private network access is blocked for AJAX calls from public networks, you can enable private network access by configuring JasperReports Server to send an Access-Control-Allow-Private-Network: true header back to the browser.

To enable private network access

1. Find tomcat/webapps/jasperserver-pro/WEB-INF/applicationContext-security-web.xml
2. Find <bean id="corsProcessor" class="com.jaspersoft.jasperserver.api.security.csrf.JSCorsProcessor">
3. In that bean under <property name="headerUrlPatterns">, uncomment:

```
<entry key="Access-Control-Allow-Private-Network">
        <util:list>
                <value>.*</value>
        </util:list>
</entry>
```

4. And under <property name="headerValues">, uncomment:

```
<<entry key="Access-Control-Allow-Private-Network">
        <util:list>
                <value>true</value>
        </util:list>
</entry>
```

Note that in the property *headerUrlPatterns*, you can set regex which will decide to which URL requests these headers should be set, by default there is .* which means that this header will be set for every request.

Serving Visualize.js from a CDN

This feature allows you to serve Visualize.js scripts from the static host or a CDN.

Setup

To set up:

1. Copy the following folder: *<tomcat>/webapps/jasperserver-pro/scripts/visualize/* (or if you are building Visualize from the source code: *<jasperserver-ui>/pro/jrs-ui-pro/build/overlay/scripts/visualize/*) to the location which will be used as a static resources source.
2. Set up your static resources server or CDN to serve content of this folder.

Using Visualize.js from a CDN

Assumptions:

- The content of the visualize.js folder described above is available at the following URL: *https://somehost.com/visualize*
- The JasperReports Server instance is available at the URL: *https://jrshost.bi/jasperserver-pro*

The following sample shows how to use static Visualize.js:

```
<script src="https://somehost.com/visualize/visualize.js"></script>
 <div id="container"></div>
<script>
  visualize({
    publicPath: "https://somehost.com/visualize/",
    server: "https://jrshost.bi/jasperserver-pro",
    auth: {
      name: "superuser",
      password: "superuser"
    }
  }, function (v) {
    v.dashboard({
      resource: "/public/Samples/Dashboards/1._Supermart_Dashboard",
      container: "#container",
      error: function (e) {
        console.log(e);
      }
    });
  });
</script>
```

- **publicPath** - this property MUST be present if Visualize.js scripts are served from some separate server (not from JRS instance). Its value should be the URL where folder described in a Set Up section. Value should ends with /
- **server** - this property is mandatory when Visualize.js scripts are served from separate server.

You cannot use URL parameters to configure Visualize.js when it is served from a separate server. So in the above example, this will not work:
https://somehost.com/visualize/visualize.js?baseUrl=https://jrshost.bi/jasperserver-pro

How to Avoid Using publicPath

If you are building Visualize.js from the source code, it is possible to hardcode the value of the publicPath property into the Visualize.js scripts during the build, so that the **publicPath** property becomes optional.

To do this, build Visualize.js from the source code (you should set up your build environment by reading the README.md in the source code):

1. Create **.env.loca**l file in *<jasperserver-ui>/pro/jrs-ui-pro* folder.
2. Add the following lines to this file:

```
WEBPACK_INCLUDE_CONFIGS=visualize
WEBPACK_PUBLIC_PATH=https://somehost.com/visualize/
```

3. Build:

```
cd <jasperserver-ui>/pro/jrs-ui-pro
yarn run build
```

4. Use build result from this folder *<jasperserver-ui>/pro/jrs-ui-pro/build/overlay/scripts/visualize/* to serve Visualize.js.

Serving CSS from a CDN

Visualize.js requires a special CSS which is loaded by default from the JRS instance (which is configured by server property).

This allows you to use the themeability feature because the CSS is loaded after authentication process, so the loaded CSS respects the logged in user theme.

But sometimes you might want to serve CSS from the static server (CDN).

Assuming that the CSS is accessible at this url: *https://somehost.com/visualize/css*, then the following CSS files should be accessible:

https://somehost.com/visualize/css/jasper-ui/jasper-ui.css

https://somehost.com/visualize/css/jquery-ui/jquery-ui.css

https://somehost.com/visualize/css/dashboard/canvas.css

https://somehost.com/visualize/css/panel.css

https://somehost.com/visualize/css/webPageView.css

https://somehost.com/visualize/css/pagination.css

https://somehost.com/visualize/css/menu.css

https://somehost.com/visualize/css/simpleColorPicker.css

https://somehost.com/visualize/css/notifications.css

In this case, use the following configuration to use custom CSS files:

```
visualize({
//... other config props
theme: {
href: "https://somehost.com/visualize/css"
}
}, function (v) {
//.....
});
```

API Reference - login and logout

The initialization of the script sets the authentication method and credentials you want to use for accessing JasperReports Server. You can then use the login and logout functions to manage multiple user sessions.

This chapter contains the following sections:

- Authentication Properties
- Authentication Functions
- Login With Plain Text Credentials
- Login With SSO Token
- Logging Out
- Login With Hooks
- UI for Login/Logout
- UI for Login/Logout With SSO Token
- Sharing Credentials Among Calls
- Using Visualize.js Without Authentication

Authentication Properties

The properties argument to the visualize function has all the fields for specifying various authentication methods.

```
{
    "allOf": [
        {
            "$ref": "BIComponentSchema.json"
        },
        {
            "type": "object",
```

```
            "description": "Authentication Properties",
            "properties":{
                "url": {
                    "type" : "string",
                    "description": "Jasper Reports Server deployment URL"
                },
                "name": {
                    "type" : "string",
                    "description": "Name of the user to authenticate"
                },
                "password": {
                    "type" : "string",
                    "description": "Password of the user to authenticate"
                },
                "organization": {
                    "type": ["string", "null"],
                    "description": "Organization of the user to authenticate"
                },
                "locale": {
                    "type": "string",
                    "description": "Default user locale to use for this session. Can be overridden
for particular user action"
                },
                "timezone": {
                    "type": "string",
                    "description": "Default user timezone to use for this session. Can be
overridden for particular user action"
                },
                "token" : {
                    "type": "string",
                    "description": "SSO authentication token. If present, then all the rest
parameters are ignored"
                },
                "preAuth" : {
                    "type": "boolean",
                    "description": "Pre authentication enabled flag. If true, then authentication
request is sent to base JRS URL, otherwise to {baseUrl}/j_spring_security_check"
                },
                "tokenName" : {
                    "type": "string",
                    "description": "SSO authentication token name. Use this parameter to override
default SSO token parameter name"
                },
                "loginFn": {
                    "type": "function",
                    "description": "A function to process login"
                },
                "logoutFn": {
                    "type": "function",
                    "description": "A function to process logout"
                }
            },
          "required": ["url"]
        }
    ]
}
```

Authentication Functions

The Authentication module has functions for logging in and logging out.

```
define(function () {
    /**
     * @param {Object} properties - authentication properties
     * @constructor
     */
    function Authentication(properties){}

    /**
     * Performs JRS login using current Authentication instnce properties.
     * @param {Function} callback - optional, invoked in case successful logout
     * @param {Function} errorback - optional, invoked in case of failed logout
     * @param {Function} always - optional, invoked always
     * @return {Deferred} dfd - login deferred
     */
    Authentication.prototype.run= function(callback, errorback, always){};

    /**
     * Performs JRS authentication.
     * @param {Object} properties - mandatory, contain all the required properties for the
authentication
     * @param {Function} callback - optional, invoked in case successful login
     * @param {Function} errorback - optional, invoked in case of failed login
     * @param {Function} always - optional, invoked always
     * @return {Deferred} dfd - authentication deferred
     */
    Authentication.prototype.login= function(properties, callback, errorback, always){};
    /**
     * Performs JRS logout. Session is dropped.
     * @param {Function} callback - optional, invoked in case successful logout
     * @param {Function} errorback - optional, invoked in case of failed logout
     * @param {Function} always - optional, invoked always
     * @return {Deferred} dfd - logout deferred
     */
    Authentication.prototype.logout= function(callback, errorback, always){};
    return Authentication;
});
```

There are several ways to set the user credentials, based on your environment.

Login With Plain Text Credentials

Use the methods of the Authentication module to set the credentials and perform login and logout operations.

```
var authentication = new Authentication({
    name:"JoeUser",
    password:"supersecret",
    organization:"organization_1"
    locale: "en",
    timezone: "Europe/Helsinki"
});
authentication.run();
...
authentication.logout();
```

Alternatively, you can specify the username, password, organization (if required), and optional parameters in the auth property of the visualize object itself.

```
visualize({
    auth: {
        name: "JoeUser",
        password: "supersecret",
        organization:"organization_1",
        timezone: "Europe/Helsinki"
    }
}, function (v) {
    ...
}, function () {
    alert("Unexpected error!");
});
```

Login With SSO Token

If you have single-sign-on (SSO) implemented and have configured JasperReports Server to use it, you can specify the SSO token in the Authentication module or Visualize.js. This example shows a token from a Central Authentication Service (CAS) server.

```
var authentication = new Authentication({
token: "ST-40-CZeUUnGPxEqgScNbxh9l-sso-cas.prod.jaspersoft.com",
});
authentication.run();
```

Or:

```
visualize({
    auth : { token : "ST-40-CZeUUnGPxEqgScNbxh9l-sso-cas.prod.jaspersoft.com"}
}, function (v){
    alert("You are now logged into JasperReports Server with your SSO token.");
    ...
}, function(err){
    alert(err.message);
});
```

Some SSO implementations require encoding, additional parameters, or both. For example, if your server is configured for pre-authentication, you could use the following example to authenticate from Visualize.js. Note that the encoded fields depend on the specifics of your pre-authentication configuration:

```
var t = encodeURIComponent("u=John|r=Ext_User|o=organization_1|pa1=USA|pa2=1");
visualize({
    auth: {
        token: t,
        preAuth: true,
        tokenName: "pp"
    }
}, function (v){
    ...
});
```

If you have configured token-based pre-authentication with WebLogic Server, there is one additional setting needed for authentication to work properly with Visualize.js. Edit the file applicationContext-externalAuth-preAuth-mt.xml in the JasperReports Server web app, and remove the comments (<!-- -->) on the following line:

```
<!-- <property name="welcomePage" value="/index.htm"/> -->
```

Logging Out

To log out and destroy the current user session, call the logout function and optionally specify any action to take when done.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
    }
}, function (v) {

    ...

    //destroy session
    $("#logout").click(function () {
        v.logout().done(function () {
            alert("You are now logged out of JasperReports Server.");
        });
    });
});
```

Login With Hooks

If you have external authentication providers, you can invoke their login and logout URLs. Again, there are two similar forms, one with the Authentication module, and one with the

auth properties.

The functions you define for login and logout must return a deferred object and accept two arguments:

- properties – An object that contains all the required authentication properties.
- request – A request function your function can use to perform authentication from a website.

```
var authentication = new Authentication({
    name:"JoeUser",
    password:"supersecret",
    loginFn: function(properties, request){
        return request({url: "http://auth.example.com?username=" + properties.name + "&password="
+ properties.password});
    },
    logoutFn: function(properties, request){
        return request({url: "http://auth.example.com/logout"});
    }
});
authentication.run();
...
authentication.logout();
```

Or:

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        loginFn: function (properties, request) {
            // Use a customLogin function to authenticate
            // It must be on the same domain: 'request' works only with JRS instance
            alert("Sending custom login request to 'http://bi.example.com/customLogin'");
            return request({
                url: "http://bi.example.com/customLogin?username=" + properties.name +
"&password=" + properties.password
            });
        },
        logoutFn: function (properties, request) {
            // Use a customLogout function to destroy the session
            // It must be on the same domain: 'request' works only with JRS instance
            alert("Sending custom logout request to 'http://bi.example.com/customLogout'");
            return request({
                url: "http://bi.example.com/customLogout"
            });
        }
    }
}, function (v) {
    ...
});
```

UI for Login/Logout

You can define IDs (#name) with listeners that perform login and logout functions. In your HTML, you can then assign these IDs to the appropriate buttons or links.

```
visualize(
    function(v){
        $("#selected_resource").change(function () {
            $("#container").html("");
            createReport($("#selected_resource").val(), v);
        });

        $("#login").click(function(){
            v.login(getAuthData()).done(function(){
                createReport($("#selected_resource").val(),v);
                showMessage(".success");
            }).fail(function(){showMessage(".error");});
        });
        $("#logout").click(function(){
            v.logout().done(function(){showMessage(".logout");});
        });
        $(':disabled').prop('disabled', false);
    }
);

//create and render report to specific container
function createReport(uri, v) {
    v("#container").report({
        resource: uri,
        error: function (err) {
            alert(err.message);
            }
     });
};

function showMessage(selector){
    $(".message").hide();
    $(selector).show();
};

function getAuthData(){
    return {name: $("#j_username").val(),
            password: $("#j_password").val(),
            organization:$("#orgId").val(),
            locale:$("#userLocale").val(),
            timezone:$("#userTimezone").val()
           }
};
```

UI for Login/Logout With SSO Token

The code is slightly different if you have a login/logout UI and use SSO tokens. Note that the logout uses the .always event instead of .done.

```
visualize(
    function(v){
        $("#selected_resource").change(function () {
            $("#container").html("");
            createReport($("#selected_resource").val(), v);
        });
        $("#login").click(function(){
            v.login(getAuthData()).done(function(){
                createReport($("#selected_resource").val(),v);
                showMessage(".success");
            }).fail(function(){showMessage(".error");});
        });
```

```
        $("#logout").click(function(){
            v.logout().always(function(){showMessage(".logout");});
        });
        $(':disabled').prop('disabled', false);
    }
);

//create and render report to specific container
function createReport(uri, v) {
    v("#container").report({
        resource: uri,
        error: function (err) {
            alert(err.message);
            }
     });
};

function showMessage(selector){
    $(".message").hide();
    $(selector).show();
};

function getAuthData(){
    return {token: $("#token").val()};
};
```

Sharing Credentials Among Calls

Use the visualize.config function to define and store authentication credentials. It uses the same auth structure as the visualize function. You can then create several containers with separate calls to visualize, using the common credentials.

```
visualize.config({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization:"organization_1",
        timezone: "Europe/Helsinki"
    }
});

visualize(function (v) {
    v("#container1").report({
        resource: "/public/Samples/Reports/06g.ProfitDetailReport",
        error: function (err) {
            alert(err.message);
        }
    });
});
```

```
visualize(function (v) {
    v("#container2").report({
        resource: "/public/Samples/Reports/State_Performance",
        error: function (err) {
            alert(err.message);
        }
    });
});
```

Using Visualize.js Without Authentication

Internally, Visualize.js uses the REST API to authenticate and interact with the server. However, the REST client receives and reuses the JSESSIONID cookie that identifies it as authenticated, which is also the same cookie used in regular browser clients. Therefore, users who access the JasperReports Server web app UI and use the same browser to run a visualize.js client before their session expires don't need to authenticate in visualize.js.

If your visualize.js solution includes other browser windows or other authenticated REST calls, then you can simplify your visualize.js and remove the authentication:

```
// This assumes that authentification was made somehow prior
visualize(function (v) {

    //do what you usally do with 'v'
    console.log(v);

});
```

API Reference - resourcesSearch

The resourcesSearch function performs searches in the repository to find content that can be displayed by visualize.js.

This chapter contains the following sections:

- Search Properties
- Search Functions
- Finding Resources
- Reusing a Search Instance
- Reusing Search Results
- Discovering Available Types

Search Properties

The properties structure passed to the resourcesSearch function is defined as follows:

```
{
    "type": "object",
    "properties": {
        "server": {
            "type": "string",
            "description": "Url to JRS instance."
        },
        "q": {
            "type": "string",
            "description": "Query string. Search for occurrence in label or description of
resource."
        },
        "folderUri": {
            "type": "string",
            "description": "Parent folder URI.",
            "pattern": "^/\\w*(/\\w+)*$"
        },
```

```
        "types": {
            "type": "array",
            "description": "Type of resources to search.",
            "items": {
                "type":"string",
                "enum": [
                    "folder", "dataType", "jdbcDataSource", "awsDataSource", "jndiJdbcDataSource",
                    "virtualDataSource", "customDataSource", "beanDataSource", "xmlaConnection",
```

```
                    "listOfValues", "file", "reportOptions", "dashboard", "adhocDataView",
                    "query", "olapUnit", "reportUnit", "domainTopic", "semanticLayerDataSource",
                    "secureMondrianConnection", "mondrianXmlaDefinition", "mondrianConnection",
                    "inputControl"
                ]
            }
        },
        "offset": {
            "type": "integer",
            "description": "Pagination. Index of first resource to show.",
            "minimum": 0
        },
        "limit": {
            "type": "integer",
            "description": "Pagination. Resources count per page.",
            "minimum": 0
        },
        "recursive": {
            "type": "boolean",
            "description": "Flag indicates if search should be recursive."
        },
```

```
        "sortBy": {
            "type": "string",
            "description": "Field to sort on.",
            "enum": [
                "uri",
                "label",
                "description",
                "type",
                "creationDate",
                "updateDate",
                "accessTime",
                "popularity"
            ]
        },
        "accessType": {
            "type": "string",
            "description": "Filtering by type of access, e.g. what was done with resource.",
            "enum": [
                "viewed",
                "modified"
            ]
        },
        "showHiddenItems": {
            "type": "boolean",
            "description": "Flag indicates if hidden items should present in results."
        },
        "forceTotalCount": {
            "type": "boolean",
            "description": "If true, Total-Count header is always set (impact on performance),
                            otherwise - in first page only"
        }
    },
    "required": ["server"]
}
```

Search Functions

The resourcesSearch function exposes the following functions:

```
define(function () {

    /**
     * Constructor. Takes context as argument.
     * @param contextObj - map of properties.
     */
    function ResourcesSearch(contextObj){};

    /**
     * Get/Set 'q' parameter of the query
     * @param contextObj - map of properties.
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.q= function(value){};

    /**
     * Get/Set 'folderUri' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.folderUri= function(value){};

    /**
     * Get/Set 'type' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.type= function(value){};

    /**
     * Get/Set 'offset' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.offset= function(value){};

    /**
     * Get/Set 'limit' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.limit= function(value){};


    /**
     * Get/Set 'recursive' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.recursive= function(value){};
```

```
    /**
     * Get/Set 'sortBy' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
```

```
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.sortBy= function(value){};

    /**
     * Get/Set 'accessType' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.accessType= function(value){};

    /**
     * Get/Set 'showHiddenItems' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.showHiddenItems= function(value){};

    /**
     * Get/Set 'forceTotalCount' parameter of the query
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    ResourcesSearch.prototype.forceTotalCount= function(value){};

    return ResourcesSearch;
});
```

Finding Resources

The following code example shows how to perform a search and store the results in a variable:

```
// Populate the repository list
JRSClient.resourcesSearch({
// server: serverUrl,
folderUri:"/public/Samples",
recursive:true,
types:["reportUnit", "dashboard"],
success:listRepository,
error:function (err)
{ alert(err); }

});
```

The next two examples show different ways of handling results after making a simple repository search in the Public folder.

```
new ResourcesSearch({
```

```
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
    recursive: false
})).run(function(resourceLookups){
    // results here
});
```

```
var search = v.resourcesSearch({
    folderUri: "/public",
    recursive: false,
    success: function(repo) {
        console.log(repo.data()); // resourceLookups
    }
});
```

You can also specify the runImmediately:false parameter so that you can set up the search in the first call, and run it later in a separate call. In the following code sample, the first statement builds a query but makes no request to the server, and the second statement actually sends the request, which executes the query.

```
var query = v.resourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
    recursive: false,
    runImmediately : false
}));

query.run().done(function(results){
})
```

Reusing a Search Instance

If you make multiple searches, for example in different folders, you can create a function to do that using the ResourcesSearch function.

```
var folderContentQuery = new ResourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    recursive: false
}));

// call 1
folderContentQuery.folderUri("/uri1").run(doSomethingWithResultFunction);
...
// call 2 after some time
folderContentQuery.folderUri("/uri2").run(doSomethingWithResultFunction);
```

Reusing Search Results

Code example:

```
var call = new ResourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
    recursive: false
})).run(function(resourceLookups){
    // data() available here
});
// at this point call.data() will return null until the run callback is called.
call.data() === null // -> true

.....
// if some data was obtained earlier, it accessible via data()
var resourceLookups = call.data();
```

Discovering Available Types

You can write code to discover and display the types that can be searched and types of sorting that can be specified.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    buildControl("Resources types", v.resourcesSearch.types);
    buildControl("Resources search sort types", v.resourcesSearch.sortBy);

});
```

```
function buildControl(name, options) {

        function buildOptions(options) {
            var template = "<option>{value}</option>";
            return options.reduce(function (memo, option) {
                return memo + template.replace("{value}", option);
            }, "")
        }

    console.log(options);

    if (!options.length){
        console.log(options);
    }

    var template = "<label>{label}</label><select>{options}</select><br>",
```

```
        content = template.replace("{label}", name)
                .replace("{options}", buildOptions(options));

    $("#container").append($(content));
}
```

API Reference - report

The report function runs reports on JasperReports Server and displays the result in a container that you provide. This chapter describes how to render a report in Visualize.js.

The report function also supports more advanced customizations of hyperlinks and interactivity that are described in subsequent chapters:

- API Usage - Hyperlinks
- API Usage - Interactive Reports

This chapter contains the following sections:

- Report Properties
- Report Functions
- Report Structure
- Rendering a Report
- Getting the Embed Code of a Report
- Setting Report Parameters
- Saving a Report
- Rendering Multiple Reports
- Resizing a Report
- Setting Report Pagination
- Creating Pagination Controls (Next/Previous)
- Creating Pagination Controls (Range)
- Exporting From a Report
- Exporting Data From a Report
- Refreshing a Report

- Canceling Report Execution
- Discovering Available Charts and Formats

Report Properties

The properties structure passed to the report function is defined as follows:

```
{
    "title": "Report Properties",
    "type": "object",
    "description": "A JSON Schema describing a Report Properties",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
        "server": {
            "type": "string",
            "description": "URL of JRS instance."
        },
        "resource": {
            "type": "string",
            "description": "Report resource URI.",
            "pattern": "^/[^/~!#\\$%^|\\s`@&*()\\-+={}\\[\\]:;\"'<>,?/\\|\\\\]+(/
[^/~!#\\$%^|\\s`@&*()\\-+={}\\[\\]:;\"'<>,?/\\|\\\\]+)*$"
        },
        "container": {
            "oneOf": [
                {
                    "type": "object",
                    "additionalProperties" : true,
                    "description": "DOM element to render report to"
                },
                {
                      "type": "string",
                      "description": "CSS selector for container to render report to."
                }
            ]
        },
        "params": {
            "type": "object",
            "description": "Report's parameters values",
            "additionalProperties": {
                "type": "array"
            }
        },
        "pages": {
            "type": ["string", "integer", "object"],
            "description": "Range of report's pages or single report page",
            "pattern": "^[1-9]\\d*(\\-\\d+)?$",
            "properties": {
                "pages": {
                    "type": ["string", "integer"],
                    "description": "Range of report's pages or single report page",
                    "pattern": "^[1-9]\\d*(\\-\\d+)?$",
                    "default": 1,
                    "minimum": 1
                },
                "anchor": {
                    "type": ["string"],
                    "description": "Report anchor"
```

```
            }
        },
        "default": 1,
        "minimum": 1
    },
```

```
    "scale" : {
        "default": "container",
        "oneOf" : [
            {
                "type": "number",
                "minimum" : 0,
                "exclusiveMinimum": true,
                "description" : "Scale factor"
            },
            {
                "enum": ["container", "width", "height"],
                "default": "container",
                "description" : "Scale strategy"
            }
        ]
    },
    "defaultJiveUi": {
        "type": "object",
        "description": "Default JIVE UI options.",
        "properties": {
            "enabled": {
                "type": "boolean",
                "description": "Enable default JIVE UI.",
                "default": "true"
            }
        }
    },
    "isolateDom": {
        "type": "boolean",
        "description": "Isolate report in iframe.",
        "default": "false"
    },
    "linkOptions": {
        "type": "object",
        "description": "Report's parameters values",
        "properties": {
            "beforeRender": {
                "type": "function",
                "description": "A function to process link - link element pairs."
            },
            "events": {
                "type": "object",
                "description": "Backbone-like events object to be applied to JR links",
                "additionalProperties" : true
            }
        }
    },
    "ignorePagination": {
        "type": "boolean",
        "description": "Control if report will be split into separate pages",
        "default": false
    },
```

```
        "autoresize": {
            "type": "boolean",
            "description": "Automatically resize report on browser window resize",
            "default": true
        },
        "chart": {
            "type": "object",
            "additionalProperties": false,
            "description": "Properties of charts inside report",
            "properties": {
                "animation": {
                    "type": "boolean",
                    "description": "Enable/disable animation when report is rendered or resized.
Disabling animation may increase performance in some cases. For now works only for Highcharts-
based charts."
                },
                "zoom": {
                    "enum": [false, "x", "y", "xy"],
                    "description": "Control zoom feature of chart reports. For now works only for
Highcharts-based charts."
                }
            }
        },
        "loadingOverlay": {
            "type": "boolean",
            "description": "Enable/disable report loading overlay",
            "default": true
        },
        "scrollToTop": {
            "type": "boolean",
            "description": "Enable/disable scrolling to top after report rendering",
            "default": true
        },
        "showAdhocChartTitle": {
            "type": "boolean",
            "description": "Enable/disable showing Ad Hoc chart reports title",
            "default": true
        }
    },
    "required": ["server", "resource"]
}
```

When setting a container for Visualize.js, the container element must not be a script element, or an element must not contain script tags for the security reasons.

Report Functions

The report function exposes the following functions:

```
define(function () {

    /**
     * @param {Object} properties - report properties
     * @constructor
     */
    function Report(properties){}

    /**
     * Setters and Getters are functions around
     * schema for bi component at ./schema/ReportSchema.json
     * Each setter returns pointer to 'this' to provide chainable API
     */

    /**
     * Get any result after invoking run action, 'null' by default
     * @returns any data which supported by this bi component
     */
    Report.prototype.data = function(){};

    /**
     * Attaches event handlers to some specific events.
     * New events overwrite old ones.
     * @param {Object} events - object containing event names as keys and event handlers as values
     * @return {Report} report - current Report instance (allows chaining)
     */
    Report.prototype.events = function(events){};

    //Actions

    /**
     * Perform main action for bi component
     * Callbacks will be attached to deferred object.
     * @param {Function} callback - optional, invoked in case of successful run
     * @param {Function} errorback - optional, invoked in case of failed run
     * @param {Function} always - optional, invoked always
     * @return {Deferred} dfd
     */
    Report.prototype.run = function(callback, errorback, always){};

    /**
     * Render report to container, previously specified in property.
     * Clean up all content of container before adding Report's content
     * @param {Function} callback - optional, invoked in case successful export
     * @param {Function} errorback - optional, invoked in case of failed export
     * @param {Function} always - optional, optional, invoked always
     * @return {Deferred} dfd
     */
    Report.prototype.render = function(callback, errorback, always){};

    /**
     * Refresh report execution
     * @param {Function} callback - optional, invoked in case of successful refresh
     * @param {Function} errorback - optional, invoked in case of failed refresh
     * @param {Function} always - optional, invoked optional, invoked always
```

```
     * @return {Deferred} dfd
     */
    Report.prototype.refresh = function(callback, errorback, always){};

    /**
     * Cancel report execution
     * @param {Function} callback - optional, invoked in case of successful cancel
     * @param {Function} errorback - optional, invoked in case of failed cancel
     * @param {Function} always - optional, invoked optional, invoked always
     * @return {Deferred} dfd
     */
    Report.prototype.cancel = function(callback, errorback, always){};

    /**
     * Update report's component
     * @param {Object} component - jive component to update, should have id field
     * @param {Function} callback - optional, invoked in case of successful update
     * @param {Function} errorback - optional, invoked in case of failed update
     * @param {Function} always - optional, invoked optional, invoked always
     * @return {Deferred} dfd
     */
    Report.prototype.updateComponent = function(component, callback, errorback, always){};

    /**
     * Update report's component
     * @param {String} id - jive component id
     * @param {Object} properties - jive component's properties to update
     * @param {Function} callback - optional, invoked in case of successful update
     * @param {Function} errorback - optional, invoked in case of failed update
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
     */
    Report.prototype.updateComponent = function(id, properties, callback, errorback, always){};

    /**
     * Save JIVE components state
     * @param {Function} callback - optional, invoked in case of successful update
     * @param {Function} errorback - optional, invoked in case of failed update
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
     */
    Report.prototype.save = function(callback, errorback, always){};

    /**
     * Save JIVE components state as new report
     * @param {Object} options - resource information (i.e. folderUri, label, description,
overwrite flag)
     * @param {Function} callback - optional, invoked in case of successful update
     * @param {Function} errorback - optional, invoked in case of failed update
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
     */
    Report.prototype.save = function(options, callback, errorback, always){};

    /**
     * Undo previous JIVE component update
     * @param {Function} callback - optional, invoked in case of successful update
     * @param {Function} errorback - optional, invoked in case of failed update
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
```

```
actors nothing will happen.
     * @param {Function} callback - optional, invoked in case of successful resize
     * @param {Function} errorback - optional, invoked in case of failed resize
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
     */
    Report.prototype.resize = function(callback, errorback, always){};

    /**
     * Search for text content through all pages in report
     * @param {String} query - text content to search through all report's pages
     * @param {Function} callback - optional, invoked in case of successful resize
     * @param {Function} errorback - optional, invoked in case of failed resize
     * @param {Function} always - optional, invoked optional, invoked always
     * @return{Deferred} dfd
     */
    Report.prototype.search = function(query, callback, errorback, always){};

    return Report;
});
```

Report Structure

The Report Data structure represents the rendered report object manipulated by the report function. Even though it is named "data," it does not contain report data, but rather the data about the report. For example, it contains information about the pages and bookmarks in the report.

The report structure also contains other components described elsewhere:

- The definitions of hyperlinks and how to work with them is explained in Customizing Links
- Details of the Jaspersoft Interactive Viewer and Editor (JIVE UI) are explained in Interacting With JIVE UI Components.

```
{
    "title": "Report Data",
    "description": "A JSON Schema describing a Report Data",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "totalPages": {
            "type": "number",
            "description": "Report's page total count"
        },
        "links": {
```

```
            "type": "array",
            "description": "Links extracted from markup, so their quantity depends on pages you
have requested",
            "items": {
                "$ref": "#/definitions/jrLink"
            }
        },
        "bookmarks": {
            "type": "array",
            "description": "Report's bookmarks. Quantity depends on current page",
            "items": {
                "$ref": "#/definitions/bookmark"
            }
        },
        "reportParts": {
            "type": "array",
            "description": "Report's parts. Quantity depends on current page",
            "items": {
                "$ref": "#/definitions/reportPart"
            }
        },
        "components": {
            "type": "array",
            "description": "Components in report, their quantity depends on pages you have
requested",
            "items": {
                "type": "object",
                "description": "JIVE components data"
            }
        }
    },
    "definitions": {
        "bookmark":{
            "type": "object",
            "properties":{
                "page": "number",
                "anchor": "string",
                "bookmarks": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/bookmark"
                    }
                }
            }
        },
        "reportPart":{
            "type": "object",
            "properties":{
                "page": "number",
                "name": "string"
            }
        },
        "jrLink": { // see chapter on hyperlinks
        }
    }
}
```

Rendering a Report

To run a report on the server and render it in Visualize.js, create a report object and set its properties. The server and resource properties determine which report to run, and the

container property determines where it appears on your page.

```
var report = v.report({
    server: "http://bi.example.com:8080/jasperserver-pro",
    resource: "/public/Sample/MyReport",
    container: "#container"
});
```

The following code example shows how to display a report that the user selects from a list.

```
visualize({
    auth: { ...
    }
}, function (v) {

    //render report from provided resource
    v("#container").report({
        resource: $("#selected_resource").val(),
        error: handleError
    });

    $("#selected_resource").change(function () {
        //clean container
        $("#container").html("");
        //render report from another resource
        v("#container").report({
            resource: $("#selected_resource").val(),
            error:handleError
        });
    });

     //enable report chooser
    $(':disabled').prop('disabled', false);

    //show error
    function handleError(err){
        alert(err.message);
    }

});
```

The HTML page that displays the report uses a static list of reports in a drop-down selector, but otherwise needs only a container element.

```
<!--Provide the URL to visualize.js-->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<select id="selected_resource" disabled="true" name="report">
    <option value="/public/Samples/Reports/1._Geographic_Results_by_Segment_Report">Geographic
Results by Segment</option>
    <option value="/public/Samples/Reports/2_Sales_Mix_by_Demographic_Report">Sales Mix by
Demographic</option>
    <option value="/public/Samples/Reports/3_Store_Segment_Performance_Report">Store Segment
Performance</option>
    <option value="/public/Samples/Reports/04._Product_Results_by_Store_Type_Report">Product
Results by Store Type</option>
</select>
<!--Provide a container to render your visualization-->
<div id="container"></div>
```

Getting the Embed Code of a Report

As of JasperReports Server 7.9, the server can provide the code to embed any report displayed in the report viewer. This lets you browse the repository, preview a report, and get its basic embed code. You can paste this code directly in your application, and then edit it for your needs. You can also open the report's embed code in JSFiddle to see the effect of your edits in real time, then copy the final code from JSFiddle.

To copy the embed code of a report

1. Log into JasperReports Server and browse the repository to find the report you want to embed.
2. Run the report to view its output in the server's report viewer.
3. Click the </> icon in the menu bar to view the embed code.
4. The Report Embed Code dialog shows you the Visualize.js code and a preview of the report as it is currently saved. Select Copy Code to copy the entire code block to your clipboard. You can also highlight selected parts of the code and use Ctrl-C (Command-C on Mac OS), for example if you want only the main function.

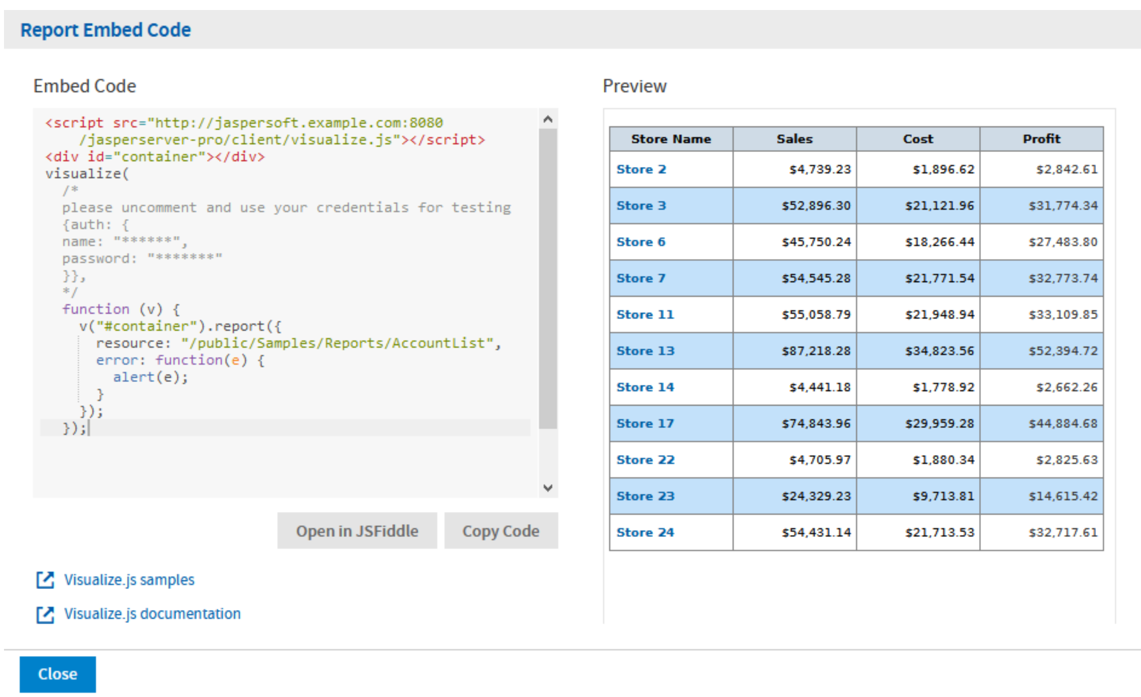

Figure 1: The Embed Code of a Report

The code sample includes comments where you can enter credentials for authentication. You should also change the name of the container to match the one in your application.

5. Alternatively, select Open in JSFiddle to load the same code into a new Fiddle, an online JavaScript viewer and interactive editor. This lets you modify the JavaScript or HTML, and add CSS if desired, then see the results in real time.

Before or after copying the embed code, you can fine-tune the report within the viewer, for example sorting a table colum or changing the chart type. Save your changes to make them available to others. Visualize.js always displays the latest saved version of a report, as determined by the repository URL in the embed code.

When displayed through Visualize.js, a report with a chart includes the icon in the top left corner that allows users to switch between chart types, though not all chart types work with the data in a given report. This selector changes the chart in the user's current session, but the changes can't be saved. For every new session, Visualize.js displays the default appearance of the chart in the report.

Setting Report Parameters

To set or change the parameter values, update the params object of the report properties and invoke the run function again.

```
// update report with new parameters
report
    .params({ "Country": ["USA"] })
    .run();
...
// later in code
console.log(report.params()); // console log output: {"Country": ["USA"] }
```

The example above is trivial, but the power of Visualize.js comes from this simple code. You can create any number of user interfaces, database lookups, or your own calculations to provide the values of parameters. Your parameters could be based on 3rd party API calls that get triggered from other parts of the page or other pages in your app. When your reports can respond to dynamic events, they are seamlessly embedded and much more relevant to the user.

Here are further guidelines for setting parameters:

- If a report has required parameters, you must set them in the report object of the initial call, otherwise you'll get an error. For more information, see Catching Report Errors.
- Parameters are always sent as arrays of quoted string values, even if there is only one value, such as ["USA"] in the example above. This is also the case even for single value input such as numerical, boolean, or date/time inputs. You must also use the array syntax for single-select values as well as multi-select parameters with only one selection. No matter what the type of input, always set its value to an array of quoted strings.
- The following values have special meanings:
    - "" – An empty string, a valid value for text input and some selectors.
    - "~NULL~" – Indicates a NULL value (absence of any value), and matches a field that has a NULL value, for example if it has never been initialized.
    - "~NOTHING~" – Indicates the lack of a selection. In multi-select parameters, this is equivalent to indicating that nothing is deselected, thus all are selected. In a single-select non-mandatory parameter, this corresponds to no selection (displayed as ---). In a single-select mandatory parameter, the lack of selection makes it revert to its default value.

Saving a Report

Once you change the report parameters, you can save the new report in the repository. You can invoke the report.save function without parameters to overwrite the current report. You can also specify a new name or a new folder to save as a different report. The authenticated user must have write permission to the report or to the folder.

```
report.save();

report.save({folderUri:"/public",
            label:"Some report label",
            description:"Some report description",
            overwrite:true});
```

The following schema describes the parameters to the report.save function:

```
{
    "title": "Report save options",
    "type": "object",
    "properties": {
        "folderUri": {
            "type": "string",
            "description": "The URI of a folder to save a report",
            "pattern": "^/.+"
        },
        "label": {
            "type": "string",
            "description": "Report resource label"
        },
        "description": {
            "type": "string",
            "description": "Report resource description"
        },
        "overwrite": {
            "type": "boolean",
            "description": "If true and there is a resource with the same name, then the resource
will be overwritten",
            "default": "false"
        }
    },
    "required": ["folderUri"]
}
```

Rendering Multiple Reports

JavaScript Example:

```
visualize({
    auth: { ...
    }
}, function (v) {

    var reportsToLoad = [
        "/public/Samples/Reports/AllAccounts",
        "/public/Samples/Reports/01._Geographic_Results_by_Segment_Report",
        "/public/Samples/Reports/Cascading_Report_2_Updated",
        "/public/Samples/Reports/07g.RevenueDetailReport"
    ];

    $.each(reportsToLoad, function (index, uri) {
        var container = "#container" + (index + 1);
        v(container).report({
            resource: uri,
            success: function () {
                console.log("loaded: " + (index + 1));
            },
            error: function (err) {
                alert(err.message);
            }
        });
    });
});
```

Associated HTML:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.0/jquery.min.js"></script>
<script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.10.0/jquery-ui.min.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<table class="sample">
    <tr>
        <td id="container1"></td>
        <td id="container2"></td>
    </tr>
    <tr>
        <td id="container3"></td>
        <td id="container4"></td>
    </tr>
</table>
```

Associated CSS:

```
html, body {
}
table.sample {
    width: 100%;
}
td#c1, td#c2, td#c3, td#c4 {
    width: 50%;
}
```

Resizing a Report

When rendering a report, by default it is scaled to fit in the container you specify. When users resize their window, reports will change so that they fit to the new size of the container. This section explains several ways to change the size of a rendered report.

To set a different scaling factor when rendering a report, specify its scale property:

- container – The report is scaled to fully fit within the container, both in width and height. If the container has a different aspect ratio, there will be white space in the dimension where the container is larger. This is the default scaling behavior when the scale property is not specified.
- width – The report is scaled to fit within the width of the container. If the report is taller than the container, users will need to scroll vertically to see the entire report.
- height – The report is scaled to fit within the height of the container. If the report is wider than the container, users will need to scroll horizontally to see the entire report.
- Scale factor – A decimal value greater than 0, with 1 being equivalent to 100%. A value between 0 and 1 reduces the report from its normal size, and a value greater than 1 enlarges it. If either or both dimensions of the scaled report are larger than the container, users will need to scroll to see the entire report.

In every case, the entire report is scaled in both directions by the same amount, you cannot change the aspect ratio of tables and crosstab elements.

For example, to initialize the report to half-size (50%), specify the following scale:

```
var report = v.report({
    resource: "/public/Sample",
    container: "#reportContainer",
    scale: 0.5
});
```

You can also change the scale after rendering, in this case to more than double size (250%):

```
report
    .scale(2.5)
    .run();
```

Alternatively, you can turn off the container resizing and modify the size of the container explicitly:

```
var report = v.report({
    resource: "/public/Sample",
    container: "#reportContainer",
    scale: "container",
    autoresize: false
});

$("#reportContainer").width(500).height(500);
report.resize();
```

To make the Report Viewer responsive, an additional property is introduced in Visualize.js that is reportContainerWidth. You need to set the value of this property to the width of the report container. Then this value is passed to the report execution process as `REPORT_CONTAINER_WIDTH` built-in parameter value.

```
visualize({
    auth: {...
    }
}, function(v) {
     let reportConfig = {
         resource: "/public/ResponsiveReport",
         container: "#container",
         reportContainerWidth: getContainerWidth(),
         events: {
               responsiveBreakpointChanged: function(error) {
                    if (error) {
                        console.log(error);
                     } else {
                         report.destroy();
                         // rerun report
                         reportConfig.reportContainerWidth = getContainerWidth();
                         report = v.report(reportConfig);
                            }
                        }
                    },
                    error: (e) => console.error(e.message || e)
            };
            let report = v.report(reportConfig);

            function getContainerWidth() {
                  return document.getElementById("container").clientWidth;
           }
 });
```

Setting Report Pagination

To set or change the pages displayed in the report, update the pages object of the report properties and invoke the run function again.

```
    report
```

```
        .pages(5)
        .run(); // re-render report with page 5 into the same container

    report
        .pages("2") // string is also allowed
        .run();

    report
        .pages("4-6") // a range of numbers as a string is also possible
        .run();

    report
        .pages({     // alternative object notation
            pages: "4-6"
        })
        .run();
```

The pages object of the report properties also supports bookmarks by specifying the anchor property. You can also specify both pages and bookmarks as shown in the example below. For more information about bookmarks, see Providing Bookmarks in Reports.

```
    report
        .pages({    // bookmark inside report to navigate to
            anchor: "summary"
        })
        .run();

    report
        .pages({    // set bookmark to scroll report to in scope of provided pages
            pages: "2-5",
            anchor: "summary"
        })
        .run();
```

Creating Pagination Controls (Next/Previous)

Again, the power of Visualize.js comes from these simple controls that you can access programmatically. You can create any sort of mechanism or user interface to select the page. In this example, the HTML has buttons that allow the user to choose the next or previous pages.

```
visualize({
    auth: { ...
    }
}, function (v) {

    var report = v.report({
        resource: "/public/Samples/Reports/AllAccounts",
```

```
        container: "#container"
    });

    $("#previousPage").click(function() {
        var currentPage = report.pages() || 1;

        report
            .pages(--currentPage)
            .run()
                .fail(function(err) { alert(err); });
    });

    $("#nextPage").click(function() {
        var currentPage = report.pages() || 1;

        report
            .pages(++currentPage)
            .run()
                .fail(function(err) { alert(err); });
    });
});
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="previousPage">Previous Page</button><button id="nextPage">Next Page</button>

<div id="container"></div>
```

Creating Pagination Controls (Range)

JavaScript Example:

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.report({
        resource: "/public/Samples/Reports/AllAccounts",
        container: "#container"
    });

    $("#pageRange").change(function() {
        report
            .pages($(this).val())
            .run()
                .fail(function(err) { alert(err); });
    });
});
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

Page range: <input type="text" id="pageRange"></input>

<div id="container"></div>
```

Exporting From a Report

To export a report, invoke its export function and specify the outputFormat property. You MUST wait until the run action has completed before starting the export. The following export formats are supported:

"pdf", "xlsx", "rtf", "csv", "xml", "odt", "ods", "docx", "json", "pptx"

For CSV and JSON output, see Exporting Data From a Report. Note that the HTML output of a report is not available through Visualize.js.

```
report.run(exportToPdf);

function exportToPdf() {
    report
        .export({
            outputFormat: "pdf"
        })
        .done(function (link) {
            window.open(link.href); // open new window to download report
        })
        .fail(function (err) {
            alert(err.message);
        });
}
```

The following sample exports 10 pages of the report to a paginated Excel spreadsheet:

```
report.run(exportToPaginatedExcel);

function exportToPaginatedExcel() {
    report
        .export({
            outputFormat: "xlsx",
            pages: "1-10",
            ignorePagination: false
        })
        .done(function(link){
            window.open(link.href); // open new window to download report
        })
        .fail(function(err){
            alert(err.message);
        });
}
```

The following sample exports the part of report associated with a named anchor:

```
report.run(exportPartialPDF);

function exportPartialPDF() {
    report
        .export({
            outputFormat: "pdf",
            pages: {
                anchor: "summary"
            }
        })
        .done(function(link){
            window.open(link.href); //open new window to download report
        })
        .fail(function(err){
            alert(err.message);
        });
}
```

The following example creates a user interface for exporting a report:

```
visualize({
    auth: { ...
    }
}, function (v) {

    var $select = buildControl("Export to: ", v.report.exportFormats),
        $button = $("#button"),
        report = v.report({
            resource: "/public/Samples/Reports/5g.AccountsReport",
            container: "#container",

            success: function () {
                button.removeAttribute("disabled");
            },

            error: function (error) {
                console.log(error);
            }
        });

    $button.click(function () {

        console.log($select.val());

        report.export({
            //export options here
            outputFormat: $select.val(),
            //exports all pages if not specified
            //pages: "1-2"
        }, function (link) {
           var url = link.href ? link.href : link;
           window.location.href = url;
        }, function (error) {
            console.log(error);
        });
    });

    function buildControl(name, options) {
```

```
        function buildOptions(options) {
            var template = "<option>{value}</option>";
            return options.reduce(function (memo, option) {
                return memo + template.replace("{value}", option);
            }, "")
        }

        var template = "<label>{label}</label><select>{options}</select><br>",
            content = template.replace("{label}", name)
                .replace("{options}", buildOptions(options));

        var $control = $(content);
        $control.insertBefore($("#button"));
        //return select
        return $($control[1]);
    }

});
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="button" disabled>Export</button>
<!-- Provide a container for the report -->
<div id="container"></div>
```

Exporting Data From a Report

You can also request the raw data of the report in CSV or JSON format.

The following example shows how to export data in CSV format. CSV output is plain text that you must parse to extract the values that you need.:

```
report.run(exportToCsv);

function exportToCsv() {
    report
        .export({
            outputFormat: "csv"
        })
        .done(function(link, request){
            request()
                .done(function(data) {
                    // use data here, data is CSV format in plain text
                });
                .fail(function(err){
                    //handle errors here
                });
```

```
        })
        .fail(function(err){
            alert(err.message);
        });
}
```

The following example shows how to export data in JSON format. By its nature, JSON format can be used directly as data within your JavaScript.

```
report.run(exportToJson);

function exportToJson() {
    report
        .export({
            outputFormat: "json"
        })
        .done(function(link, request){
            request({
                dataType: "json"
            })
            .done(function(data) {
                // use JSON data as objects here
            })
            .fail(function(err){
                    //handle errors here
            });

        })
        .fail(function(err){
            alert(err.message);
        });
}
```

Refreshing a Report

JavaScript Example:

```
visualize({
    auth: { ...
    }
}, function (v) {

    var alwasyRefresh = false;

    var report = v.report({
        //skip report running during initialization
        runImmediately: !alwasyRefresh,
        resource: "/public/viz/usersReport",
        container: "#container1",
     });

    if (alwasyRefresh){
```

```
        report.refresh();
    }

    $("button").click(function(){
        report
            .refresh()
            .done(function(){console.log("Report Refreshed!");})
            .fail(function(){alert("Report Refresh Failed!");});
    });

});
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<button>Refresh</button>
<div id="container1"></div>
```

Canceling Report Execution

To stop a running report, call its cancel function:

```
    ...
    report
    .cancel()
    .done(function(){
        alert("Report Canceled");
    })
    .fail(function(){
        alert("Report Failed");
    });
```

The following example is more complete and creates a UI for a spinner and cancel button for a long-running report.

```
var spinner = createSpinner();

visualize({
    auth: { ...
    }
}, function (v) {

    var button = $("button");

    var report = v.report({
        resource: "/public/Reports/Slow_Report",
```

```
        container: "#container",
        events: {
            changeTotalPages : function(){
               spinner.remove();
            }
        }
    });

    button.click(function () {
        report
            .cancel()
            .then(function () {
                spinner.remove();
                alert("Report Canceled!");
            })
            .fail(function () {
                alert("Can't Cancel Report");
            });
    });
});

function createSpinner() {
    var opts = {
        lines: 17, length: 3, width: 2, radius: 3, corners: 0.6, rotate: 0, direction: 1,
        color: '#000', speed: 1, trail: 60, shadow: false, hwaccel: false, zIndex: 2e9,
        top: 'auto', left: 'auto', className: 'spinner'
    };
    var container = $("#spinner");
    var spinner = new Spinner(opts).spin(container[0]);
    return container;
}
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://fgnass.github.io/spin.js/spin.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<div id="spinner"></div>
<button>Cancel</button>
<div id="container"></div>
```

Discovering Available Charts and Formats

You can write code to discover and display the types of charts and export formats that can be specified. The following example reads the exportFormats, chart.types, and table.column.types of the given report and dynamically creates a selection dialog for each:

```
visualize({
    auth: {
```

```
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    buildControl("Report export formats", v.report.exportFormats);
    buildControl("Chart types", v.report.chart.types);
    buildControl("Report table column types", v.report.table.column.types);

});

function buildControl(name, options) {

        function buildOptions(options) {
            var template = "<option>{value}</option>";
            return options.reduce(function (memo, option) {
                return memo + template.replace("{value}", option);
            }, "")
        }

    console.log(options);

    if (!options.length){
        console.log(options);
    }

    var template = "<label>{label}</label><select>{options}</select><br>",
        content = template.replace("{label}", name)
                .replace("{options}", buildOptions(options));

    $("#container").append($(content));
}
```

API Reference - inputControls

The inputControls function prepares and displays the input controls, also known as the filters, of a report. Your users interact with the controls to submit input when running reports.

Visualize.js supports much simpler input controls that are rendered directly by the server. All you need to do is specify the report resource and a container destination, and the server generates the UI interface that allows the user to select or enter values. You can then change the appearance of the input controls through CSS as necessary.

The previous mechanism of creating your own input control structures in JavaScript is still available to create custom input control interfaces. When creating custom input controls, your code can retrieve the input control data and generate your own input control UI interface.

This chapter contains the following sections:

- Input Control Properties

- Input Control Functions
- Embedding Input Controls
- Handling Input Control Events
- Resetting Input Control Values
- Embedded Input Control Styles
- Custom Input Controls

Input Control Properties

The properties structure passed to the inputControls function is defined as follows:

```
{
    "type": "object",
    "properties": {
        "server": {
            "type": "string",
            "description": "Url to JRS instance."
        },
        "resource": {
            "type": "string",
            "description": "URI of resource with input controls.",
            "pattern": "^/[^/~!#\\$%^|\\s`@&*()\\-+={}\\[\\]:;\"'<>,?/\\|\\\\]+(/
[^/~!#\\$%^|\\s`@&*()\\-+={}\\[\\]:;\"'<>,?/\\|\\\\]+)*$"
        },
        "params": {
            "type": "object",
            "description": "Parameters for input controls.",
            "additionalProperties": {
                "type": "array"
            }
        },
        "container": {
            "oneOf": [{
                "type": "object",
                "additionalProperties" : true,
                "description": "DOM element to render report to"
            }, {
                "type": "string",
                "description": "CSS selector for container to render report to."
            }]
        }
    },
    "required": ["server", "resource"]
}
```

Input Control Functions

The InputControls function exposes the following functions:

```
define(function () {

    /**
     * Constructor. Takes properties as argument.
     * @param properties - map of properties.
     */
    function InputControls(properties){};

    /**
     * Get/Set 'resource' property - URI of resource with input controls.
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    InputControls.prototype.resource = function(value){};

    /**
     * Get/Set 'params' property - Parameters for input controls.
     * @param value - new value, optional
     * @returns this if 'value' sent to the method,
     *          otherwise returns current value of the parameter
     */
    InputControls.prototype.params = function(value){};

    /**
     * Attaches event handlers to some specific events.
     * New events overwrite old ones.
     * @param {Object} events - object containing event names as keys
     *                          and event handlers as values
     * @return {InputControls} input controls - current InputControls
     *                                          instance (allows chaining)
     */
    Report.prototype.events = function(events){};

    /**
     * Reset input controls current state.
     * @param {Function} callback - optional, invoked in case successful export
     * @param {Function} errorback - optional, invoked in case of failed export
     * @param {Function} always - optional, optional, invoked always
     * @return {Deferred} dfd
     */
    Report.prototype.reset = function(){};

    return InputControls;
});
```

Embedding Input Controls

When you display and use input controls, the server generates the UI widgets for all of the input controls of a report; you specify the container where you want to embed them.

If desired, it is still possible to generate your own UI widgets for each input control, as done in some previous versions; see Custom Input Controls. Previously written JavaScript that created custom input controls still work.

In the simplest embedding case, you specify the URI of the report resource and a container. Only the input controls for the designated report are embedded in the container. The input

controls appear on your page as standard selection boxes and drop-down selectors that the user can interact with and choose new values. The following example shows the HTML and corresponding JavaScript:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<div id="inputContainer"></div>

visualize({
    auth: { ...
    }
}, function (v) {
    var inputControls = v.inputControls({
        resource: "/public/Samples/Reports/Cascading_Report_2_Updated",
        container: "#inputContainer",
        error: function (err) {
            console.error(err);
        }
    });


});
```

Of course, you can add styles to determine the shape and placement of your input controls container, as shown in the following CSS sample. If you wish to change the appearance of the embedded controls within the container, see Embedded Input Control Styles.

```
#inputContainer {
  width: 300px;
  padding-left: 50px;
}
```

The following is a more realistic example that shows how to embed both the report and its input controls into separate containers on your page. The page also provides a button to run the report after the user has made changes to input controls:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<script type='text/javascript' src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<div id="inputContainer" style="width:300px"></div>
<button id="run">
Run
</button>
<div id="reportContainer"></div>
```

The JavaScript embeds the report and the input controls in their respective containers, then it handles the button click to read the current values of the input controls (called parameters in this instance) and run the report using those values:

```
var inputControls="";
visualize({
    auth: { ...
    }
}, function(v) {
    var resourceUri = "/public/Samples/Reports/ProfitDetailReport";
    //render report from resource
    var report = v.report({
        resource: resourceUri,
        container: "#reportContainer"
    });

    //render input controls from resource
    inputControls= v.inputControls({
        resource: resourceUri,
        container: "#inputContainer",
        error: function(err) {
            console.error(err);
        },
        success: function(data) {
            console.log(data)
        }
    });

    //run report again using paramaters from input control
    $("#run").click(
        function(){
            console.log(inputControls.data().parameters)
            inputControls.run(null, function(e) {
        });
        var params = inputControls.data().parameters;
        report.params(params).run()
    });
});
```

In most cases, your input controls will come from the same report that you are running. As in the example above, the URI for the report and the input controls is the same report resource. However, it is possible to use input controls from a different report, as long as the values from those input controls are compatible with the target report. Because input controls are ultimately used to create the query that fetches data for a report, the input controls of one report are compatible with another report if they are based on the same data source or Domain.

For example, you might have two or more reports based on the same data source and embed the input controls from only one of them. You could then use the one set of input controls to update all of the reports simultaneously.

Handling Input Control Events

Because the input controls are separate from the report, events provide a way for you to make them work together or provide additional information. The inputControls instance

generates a change event after the user interactively sets or selects a new input control value using the UI widgets.

The most common use is to listen for input control change events in order to run the associated report with the new input control values. In this example, the HTML sample has the report and input controls containers side by side in a row (styles or CSS not shown):

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<div class="container">
  <div class="row">
    <div class="col-lg-6">
      <div id="reportContainer"></div>
    </div>
    <div class="col-lg-6">
      <div id="inputContainer"></div>
    </div>
  </div>
```

Then, the JavaScript embeds the report and the input controls in their respective containers and adds a listener for change events. If there is no error, the input control values (called params in this instance) are taken from the event and used to run the report again. This is a very common pattern when using input controls:

```
visualize({
    auth: { ...
    }
}, function (v) {

    var resourceUri = "/public/Samples/Reports/Cascading_Report_2_Updated";

    var report = v.report({
        resource: resourceUri,
        container: "#reportContainer"
    });

     var inputControls = v.inputControls({
        resource: resourceUri,
        container: "#inputContainer",
        events: {
            change: function(params, error){
            if (!error){
                report.params(params).run();
            }
          }
        }
    });
});
```

Another use for change events is to update other parts of your page based on the value of an input control. Such a mechanism could be used to update the title above a report, after the report is updated with a change of input controls.

In the following example, the change event triggers an update to text that displays the current value of an input control. The HTML has a placeholder for the text:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<div id="inputContainer"></div>

<div  id="myText" >
    <span>Selection: </span>
    <strong id='output'></strong>
</div>
```

The CSS hides the text at first:

```
#inputContainer {
    width:200px;
}

#myText {
   display:none
}
```

And the JavaScript catches the input control change event, then extracts the value of a given parameter to display it:

```
visualize({
    auth: {...
    }
}, function (v) {
    v.inputControls({
        container: "#inputContainer",
        resource: "/public/Samples/Reports/06g.ProfitDetailReport",
        error: function (err) {
            alert(err);
        },
        events: {
            change : function(params) {
                $("#myText").show();
                $("#output").text(params['ProductFamily'].join(', '));
            }
        }
    });
});
```

The change event can also be used to check whether the input controls entered by the user passed validation checks. Validation is defined on the server and ensures that values entered by the user are of the expected type or within a given range. The following example shows an event handler that checks the validation result:

```
        events: {
            change : function(state, validationResult) {
              if (validationResult) {
                console.log(validationResult)
              } else {
```

```
                console.log(state)
              }
            }
        }
```

Resetting Input Control Values

inputControls provides a function to reset all input control values to their default values. When the reset function is invoked, the display of the input controls refreshes to show their default values, that is, the initial state of the input controls before the user made any changes.

The following example has buttons to run the report and also to reset input controls. The HTML sample has the containers and buttons that are needed:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<script type='text/javascript' src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<div id="inputContainer" style="width:300px"></div>
<button id="run">
Run
</button>
<button id="reset">
Reset
</button>
<div id="reportContainer"></div>
```

The corresponding JavaScript invokes the reset function when the button is clicked, and then it runs the report as well again:

```
var inputControls="";
visualize({
    auth: { ...
    }
}, function(v) {
    var resourceUri = "/public/Samples/Reports/ProfitDetailReport";
    //render report from resource
    var report = v.report({
        resource: resourceUri,
        container: "#reportContainer"
    });

    //render input controls from resource
    inputControls= v.inputControls({
        resource: resourceUri,
        container: "#inputContainer",
        error: function(err) {
            console.error(err);
```

```
        },
        success: function(data) {
            console.log(data)
        }
    });

    function runReport(){
        console.log(inputControls.data().parameters)
        inputControls.run(null, function(e) {
        });
        var params = inputControls.data().parameters;
        report.params(params).run()
    }

    //run report again using paramaters from input controls
    $("#run").click(runReport);

    //reset parameters in input controls then run report again
    $("#reset").click(
        function(){
            console.log("Params before reset: ", inputControls.params())
            inputControls.reset().done(runReport);
    });

});
```

Embedded Input Control Styles

With the embedded input controls, the server generates all of the UI widgets within your container. These UI widgets all have a standard look and feel that is controlled by CSS also generated by the server. If you wish to change the appearance of the generated input controls, you can add your own CSS with the proper class names.

In the following simple example, the HTML has a container and the JavaScript uses it for the input controls:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<div id="inputContainer" style="width:300px"></div>

visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
}, function (v) {
    var inputControls = v.inputControls({
        resource: "/public/viz/Adhoc/Ad_Hoc_View_All_filters_Report",
        container: "#inputContainer",
        error: function (err) {
            console.error(err);
        }
    });
});
```

The following CSS will change the text color of the label of the boolean input control:

```
#inputContainer .jr-mInput-boolean-label {
    color: #218c00;
}
```

The following list shows the CSS base classes and secondary class names for the labels of each kind of input control embedded by the server. Use these in your CSS rules as shown in the example above to change the appearance of your labels.

```
#inputContainer .jr-mInputControlBoolean .jr-mInput-boolean-label
#inputContainer .jr-mInputControlSingleValueText .jr-mInput-label
#inputContainer .jr-mInputControlSingleValueDate .jr-mInput-label
#inputContainer .jr-mInputControlSingleValueTime .jr-mInput-label
#inputContainer .jr-mInputControlSingleValueDatetime .jr-mInput-label
#inputContainer .jr-mInputControlSingleValueNumber .jr-mInput-label
#inputContainer .jr-mInputControlSingleSelect .jr-mInput-label
#inputContainer .jr-mInputControlMultiSelect .jr-mInput-label
#inputContainer .jr-mInputControlSingleSelectRadio .jr-mInput-label
#inputContainer .jr-mInputControlMultiSelectCheckbox .jr-mInput-label
```

If you want to change the style of other elements in the server's embedded input control UI, you can find the corresponding CSS classes and redefine them. To find the CSS classes, write the JavaScript to embed your input controls, then test the page in a browser. Use your browser's code inspector to look at each element of the generated input controls and locate the CSS rules that apply to it. The code inspector shows you the classes and often lets you modify values to preview the look and feel that you want to create.

Custom Input Controls

In the sections above, the UI widgets for embedded input controls are generated by the server and displayed in a given container. An alternate way to create input controls for your reports in visualize.js is to retrieve the input control values and analyze their structure so you can create your own UI widgets. These are called custom input controls.

Custom input controls require more JavaScript coding, and might be specific to a given report's input controls. In addition, handling cascading input controls is more complex and requires you to listen for change events that update the input control structure. The following sections explain how to implement custom input controls.

Input Control Structure

The data() for InputControls is an array of InputControl objects, with the structure shown in this example:

```
[
  {
    "id":"Cascading_name_single_select",
    "label":"Cascading name single select",
    "mandatory":"true",
    "readOnly":"false",
    "type":"singleSelect",
    "uri":"repo:/reports/samples/Cascading_multi_select_report_files/Cascading_name_single_
select",
    "visible":"true",
    "masterDependencies": {
      "controlId": [
        "Country_multi_select",
        "Cascading_state_multi_select"
      ]
    },
    "slaveDependencies":null,
    "validationRules": [
      {
        "mandatoryValidationRule" : {
          "errorMessage" : "This field is mandatory so you must enter data."
        }
      }
    ],
```

```
    "state": {
      "uri": "/reports/samples/Cascading_multi_select_report_files/Cascading_name_single_select",
      "id": "Cascading_name_single_select",
      "value": null,
      "options": [
        {
          "selected": false,
          "label": "A & U Jaramillo Telecommunications, Inc",
          "value": "A & U Jaramillo Telecommunications, Inc"
        }
      ]
    }
  },
  ....
]
```

Fetching Input Control Data

The data being output here has the input control structure shown in the previous section:

```
visualize(function(v){
    var ic = v.inputControls({
        resource: "/public/ReportWithControls",
        success: function(data) {
           console.log(data);
        }
    });
});
```

This example shows an alternate way of fetching input controls:

```
(new InputControls({
    server: "http://localhost:8080/jasperserver-pro",
    resource: "/public/my_report",
    params: {
        "Country_multi_select":["Mexico"],
        "Cascading_state_multi_select":["Guerrero", "Sinaloa"]
    }
})).run(function(inputControlsArray){
    // results here
})
```

Creating Input Control Widgets

This example retrieves the input controls of a report and parses the structure to create drop-down menus of values for each control:

```
visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
},function(v) {

    v.inputControls({
        resource: "/public/Samples/Reports/16g.InteractiveSalesReport",
        success: function (controls) {
            controls.forEach(buildControl);
        },
        error: function (err) {
            alert(err);
        }
    });

    function buildControl(control) {

        function buildOptions(options) {
            var template = "<option>{value}</option>";
            return options.reduce(function (memo, option) {
                return memo + template.replace("{value}", option.value);
            }, "")
        }

        var template = "<label>{label}</label><select>{options}</select><br>",
            content = template.replace("{label}", control.label)
                .replace("{options}", buildOptions(control.state.options));

        $("#container").append($(content));
    }
});
```

Cascading Input Controls

In order to implement cascading input controls, you must implement a change listener on the parent control and use it to trigger an update on the dependent control:

```
var reportUri = "/public/Samples/Reports/Cascading_Report_2_Updated";

visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
}, function (v) {
    var inputControls = v.inputControls({
        resource: reportUri,
        success: renderInputControls
    });

    var report = v.report({ resource: reportUri, container: "#container" });

    $("#productFamilySelector").on("change", function() {
        report.params({ "Product_Family": [$(this).val()] }).run();
    });
});
```

```
function renderInputControls(data) {
    var productFamilyInputControl = _.findWhere(data, {id: "Product_Family"});
    var select = $("#productFamilySelector");
    _.each(productFamilyInputControl.state.options, function(option) {
        select.append("<option " + (option.selected ? "selected" : "") + " value='" +
                        option.value + "'>" + option.label + "</option>");
    });
}
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<select id="productFamilySelector"></select>
<div id="container"></div>
```

Reusing Input Control Instances

Input controls are meant to be dynamic and modified by users. By using the inputControls.params function, you can update the values of input controls and then update the corresponding report.

```
var inputControls = new InputControls({
    server: "http://localhost:8080/jasperserver-pro",
    resource: "/public/my_report"
});

// call 1
inputControls.params({ "Country_multi_select": ["Mexico"] }).run(doSomethingWithResultFunction);
...
// call 2 after some time
inputControls.params({ "Country_multi_select": ["USA"] }).run(doSomethingWithResultFunction);
```

Reusing Input Control Data

You can store the data from the inputControls function and access the data() structure at a later time:

```
var call = (new InputControls({
    server: "http://localhost:8080/jasperserver-pro",
    resource: "/public/my_report"
})).run(function(inputControlsArray){
    // data() available here
});

// at this point call.data() will return null until the run callback is called.
call.data() === null // -> true
...
// if some data was obtained earlier, it accessible via data()
var inputControlsArray = call.data();
```

API Reference - dashboard

The dashboard function runs dashboards on JasperReports Server and displays the result in a container that you provide. Dashboards are a collection of reports and widgets that you design on the server. Dashboards were entirely redesigned in JasperReports Server 6.0 to provide stunning data displays and seamless integration through Visualize.js.

This chapter contains the following sections:

- Dashboard Properties
- Dashboard Functions
- Dashboard Structure
- Rendering a Dashboard
- Getting the Embed Code of a Dashboard

- Refreshing a Dashboard
- Tracking Completion Status
- Using Dashboard Input Controls
- Using the Dashboard Undo Stack
- Setting Dashboard Hyperlink Options
- Exporting From a Dashboard
- Closing a Dashboard

Dashboard Properties

The properties structure passed to the dashboard function is defined as follows:

```
{
    "title": "Dashboard Properties",
    "type": "object",
    "description": "JSON Schema describing Dashboard Properties",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
        "server": {
            "type": "string",
            "description": "URL of JRS instance"
        },
        "resource": {
            "type": "string",
            "description": "Dashboard resource URI",
            "pattern": "^/[^/~!#\\$%^|\\s`@&*()\\+={}\\[\\]:;\"'<>,?/\\|\\\\]+(/
[^/~!#\\$%^|\\s`@&*()\\+={}\\[\\]:;\"'<>,?/\\|\\\\]+)*$"
        },
```

```
        "container": {
            "oneOf": [
                {
                    "type": "object",
                    "additionalProperties" : true,
                    "description": "DOM element to render report to"
                },
                {
                    "type": "string",
                    "description": "CSS selector for container to render report to"
                }
            ]
        },
        "linkOptions": {
            "type": "object",
```

```
            "description": "Dashboard's parameters values",
            "properties": {
                "beforeRender": {
                    "type": "function",
                    "description": "A function to process link - link element pairs."
                },
                "events": {
                    "type": "object",
                    "description": "Backbone-like events object to be applied to JR links",
                    "additionalProperties" : true
                }
            }
        },
        "params": {
            "type": "object",
            "description": "Dashboard parameter values",
            "additionalProperties": {
                "type": "array"
            }
        },
        "report": {
            "type": "object",
            "description": "Properties for reports inside dashboard",
            "additionalProperties": false,
            "required": ["chart"],
            "properties": {
                "chart": {
                    "type": "object",
                    "additionalProperties": false,
                    "description": "Properties of charts inside report",
                    "properties": {
                        "animation": {
                            "type": "boolean",
                            "description": "Enable/disable animation when report is rendered or
resized. Disabling animation may increase performance in some cases. For now works only for
Highcharts-based charts."
                        },
                        "zoom": {
                            "enum": [false, "x", "y", "xy"],
                            "description": "Control zoom feature of chart reports. For now works
only for Highcharts-based charts."
                        }
                    }
                },
                "loadingOverlay": {
                    "type": "boolean",
                    "description": "Enable/disable report loading overlay",
                    "default": true
                }
            }        }
    },
    "required": ["server", "resource"]
}
```

Dashboard Functions

The dashboard module exposes the following functions:

```
define(function () {

    /**
     * @param {Object} properties - Dashboard properties
     * @constructor
     */
    function Dashboard(properties){}

    //Special  getters

    /**
     * Get any result after invoking run action
     * @returns any data which supported by this bi component
    */
    Dashboard.prototype.data = function(){};

    //Actions

    /**
     * Perform main action for bi component. Callbacks will be attached to  deferred object.
     * @param {Function} callback - optional, invoked in case of successful run
     * @param {Function} errorback - optional, invoked in case of failed run
     * @param {Function} always - optional, invoked always
     * @return {Deferred} dfd
     */
    Dashboard.prototype.run = function(callback, errorback, always){};

    /**
     * Render Dashboard to container, previously specified in property.
     * Clean up all content of container before adding Dashboard's content.
     * @param {Function} callback - optional, invoked in case successful export
     * @param {Function} errorback - optional, invoked in case of failed export
     * @param {Function} always - optional, optional, invoked always
     * @return {Deferred} dfd
     */
    Dashboard.prototype.render = function(callback, errorback, always){};
```

```
    /**
     * Refresh Dashboard.
     * @param {Function} callback - optional, invoked in case of successful refresh
     * @param {Function} errorback - optional, invoked in case of failed refresh
     * @param {Function} always - optional, invoked optional, invoked always
     * @return {Deferred} dfd
     */
    Dashboard.prototype.refresh = function(callback, errorback, always){};

    /**
     * Refresh particular dashboard component.
     * @param {String} id - dashboard component id
     * @param {Function} callback - optional, invoked in case of successful refresh
     * @param {Function} errorback - optional, invoked in case of failed refresh
     * @param {Function} always - optional, invoked optional, invoked always
     * @return {Deferred} dfd
     */
    Dashboard.prototype.refresh = function(id, callback, errorback, always){};

    /**
     * Cancel Dashboard execution.
     * @param {Function} callback - optional, invoked in case of successful cancel
     * @param {Function} errorback - optional, invoked in case of failed cancel
```

```
         * @param {Function} always - optional, invoked optional, invoked always
         * @return {Deferred} dfd
         */
        Dashboard.prototype.cancel = function(callback, errorback, always){};

        /**
         * Cancel refresh process of particular dashboard component
         * @param {String} id - dashboard component id
         * @param {Function} callback - optional, invoked in case of successful cancel
         * @param {Function} errorback - optional, invoked in case of failed cancel
         * @param {Function} always - optional, invoked optional, invoked always
         * @return {Deferred} dfd
         */
        Dashboard.prototype.cancel = function(id, callback, errorback, always){};

        /**
         * Update dashboard component
         * @param {Object} component - dashboard component to update, should have id field
         * @param {Function} callback - optional, invoked in case of successful update
         * @param {Function} errorback - optional, invoked in case of failed update
         * @param {Function} always - optional, invoked optional, invoked always
         * @return {Deferred} dfd
         */
        Dashboard.prototype.updateComponent = function(component, callback, errorback, always){};

        /**
         * Update dashboard component
         * @param {String} id - dashboard component id
         * @param {Object} properties - dashboard component's properties to update
         * @param {Function} callback - optional, invoked in case of successful update
         * @param {Function} errorback - optional, invoked in case of failed update
         * @param {Function} always - optional, invoked optional, invoked always
         * @return{Deferred} dfd
         */
        Dashboard.prototype.updateComponent = function(id, properties, callback, errorback, always){};
```

```
        /**
         * Batch update of dashboard components
         * @param {Array} components - array of dashboard component objects. Each object should have
         *                             a component id.
         * @param {Function} callback - optional, invoked in case of successful update
         * @param {Function} errorback - optional, invoked in case of failed update
         * @param {Function} always - optional, invoked optional, invoked always
         * @return{Deferred} dfd
         */
        Dashboard.prototype.updateComponent = function(components, callback, errorback, always){};

        /**
         * Cancel all executions, destroy Dashboard representation if any, leave only
         * properties
         * @param {Function} callback - optional, invoked in case of successful cleanup
         * @param {Function} errorback - optional, invoked in case of failed cleanup
         * @param {Function} always - optional, invoked optional, invoked always
         * @return {Deferred} dfd
         */
        Dashboard.prototype.destroy = function(callback, errorback, always){};

        return Dashboard;
    });
```

Dashboard Structure

The Dashboard Data structure represents the rendered dashboard object manipulated by the dashboard function. Even though it's named "data," it does not contain any data in the dashboard or reports, but rather data about the dashboard. For example, the Dashboard Data structure contains information about the items in the dashboard, called dashlets.

```
{
    "title": "Dashboard Data",
    "description": "A JSON Schema describing a Dashboard Data",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "parameters": {
            "type": "array",
            "description": "Dashboard parameters",
            "items": {
                "type": "object",
                "description": "Dashboard parameter properties"
            }
        },
        "components": {
            "type": "array",
            "description": "Dashboard components",
            "items": {
                "type": "object",
                "description": "Dashboard component properties",
                "properties": {
```

```
                    "id": {
                        "type": "string",
                        "description": "Unique ID of dashboard component",
                        "readOnly": true
                    },
                    "name": {
                        "type": "string",
                        "description": "Unique name of dashboard component",
                        "readOnly": true
                    },
                    "type": {
                        "oneOf": ["reportUnit", "adhocDataView", "inputControl", "filterGroup",
                                  "webPageView", "text", "crosstab", "chart", "table", "value"],
                        "description": "Type of dashboard component",
                        "readOnly": true
                    },
                    "resource": {
                        "type": ["string", "null"],
                        "description": "Resource URI. Null if component has no resource.",
                        "readOnly": true
                    },
                    "position": {
                        "type": ["object", "null"],
                        "description": "Dashlet position on canvas in abstract units.",
                        "readOnly": true,
```

```
                        "properties": {
                            "x": {
                                "type": "integer",
                                "minimum": 0,
                                "description": "Dashlet X coordinate from top left corner of
canvas"
                            },
                            "y": {
                                "type": "integer",
                                "minimum": 0,
                                "description": "Dashlet Y coordinate from top left corner of
canvas"
                            },
                            "width": {
                                "type": "integer",
                                "minimum": 0,
                                "description": "Dashlet width"
                            },
                            "height": {
                                "type": "integer",
                                "minimum": 0,
                                "description": "Dashlet height"
                            }
                        }
                    },
                    "toolbar": {
                        "type": ["boolean", "object", "null"],
                        "description": "Whether dashlet toolbar should be shown. Works only if
                                        component is a dashlet and can have toolbar",
                        "properties": {
                            "maximize": {
                                "type": "boolean",
                                "description": "Whether maximize button should be shown in
                                                dashlet toolbar."
                            },
                            "refresh": {
                                "type": "boolean",
                                "description": "Whether refresh button should be shown in
                                                dashlet toolbar."
                            }
                        }
                    },
                    "interactive": {
                        "type": "boolean",
                        "description": "Enabled/disabled interactivity on particular component",
                        "default": true
                    },
                    "maximized": {
                        "type": ["boolean", "null"],
                        "description": "Maximized/minimized state of a component. Works only if
                                        component is a dashlet"
                    },
                    "pagination": {
                        "type": ["boolean", "null"],
                        "description": "Show/hide pagination control for visualizations. Control
will

                                        be hidden in any case if visualization has only one page"
                    }
                }
            }
        }
    }
}
```

Rendering a Dashboard

To run a dashboard on the server and render it in Visualize.js, create a dashboard object and set its properties. Like rendering a report, the resource property determines which dashboard to run, and the container property determines where it appears on your page.

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container",
    success: function() { console.log("dashboard rendered"); },
    error: function(e) { alert(e); }
});
```

The following code example shows how to define a dashboard ahead of time, then render it at a later time.

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    runImmediately: false
});

dashboard
    .run()
    .done(function() {
        this
            .container("#container")
            .render()
            .done(function() { console.log("dashboard rendered"); })
            .fail(function(e) { alert(e); });
    })
    .fail(function(e) { alert(e); });
```

By listening for the *dashboardCompleted* event, you can give information or take action when a dashboard finishes rendering. For more information, see Tracking Completion Status.

Getting the Embed Code of a Dashboard

As of JasperReports Server 7.9, the server can provide the code to embed any dashboard displayed in the Dashboard Designer. This lets you browse the repository, preview the dashboard, and get its basic embed code. You can paste this code directly in your application, and then edit it for your needs. You can also open the dashboard's embed code in JSFiddle to see the effect of your edits in real time, then copy the final code from JSFiddle.

To copy the embed code of a dashboard

1. Log into JasperReports Server and browse the repository to find the dashboard you want to embed.
2. View the dashboard so that it is displayed in the server's Dashboard Designer.
3. Click the </> icon in the menu bar to view the embed code.
4. The Dashboard Embed Code dialog shows you the Visualize.js code and a preview of the dashboard as it is currently saved. Select Copy Code to copy the entire code block to your clipboard. You can also highlight selected parts of the code and use Ctrl-C (Command-C on Mac OS), for example if you want only the main function.

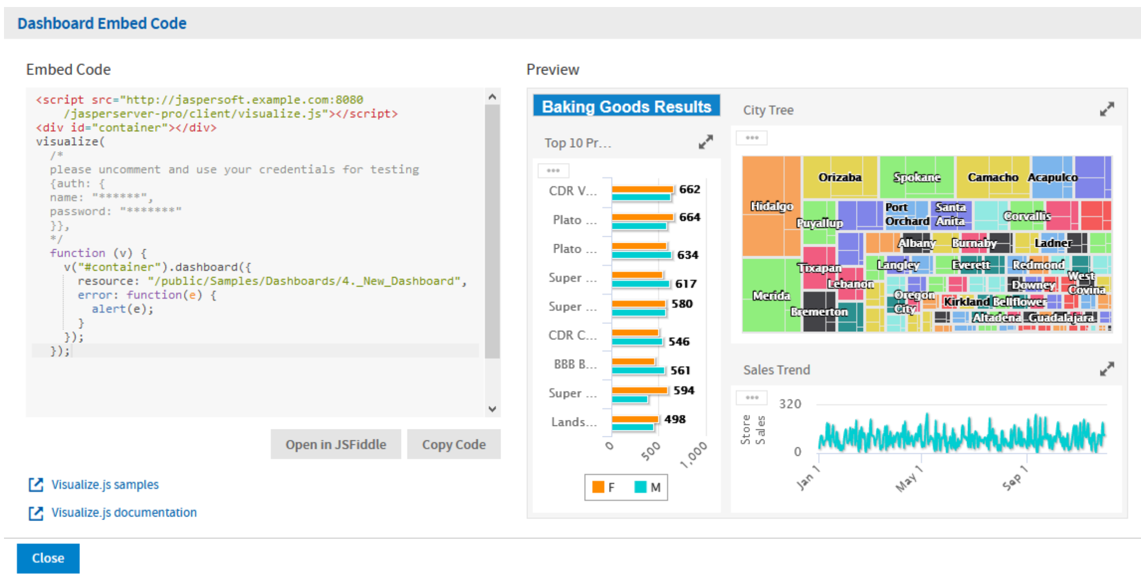

Figure 2: The Embed Code of a Dashboard

The code sample includes comments where you can enter credentials for authentication. You should also change the name of the container to match the one in your application.

5. Alternatively, select Open in JSFiddle to load the same code into a new Fiddle, an online JavaScript viewer and interactive editor. This lets you modify the JavaScript or HTML, and add CSS if desired, then see the results in real time.

If you have write permission to the dashboard, you can switch to editing mode and still get the embed code at any time. Regardless of when you get the embed code, Visualize.js always displays the latest saved version of a dashboard, as determined by the repository URL in the embed code.

When displayed through Visualize.js, the elements of a dashboard, called dashlets, include the ••• icon that allows users to switch between table, crosstab, and chart type, though not all chart types work with the data in a given dashlet. This selector changes the appearance of the dashlet in the user's current session, but the changes can't be saved. For every new session, Visualize.js displays the default appearance of the entire dashboard.

Refreshing a Dashboard

You can order the refresh or re-render of the dashboard, as well as cancel the refresh if necessary, for example if it takes too long.

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container",
    runImmediately: false
});

dashboard.run().done(function() {
    setTimeout(function() {
        var dfd = dashboard.refresh();

        // cancel refresh if it's still running after 2 seconds
        setTimeout(function() {
            if (dfd.state() === "pending") {
                dashboard.cancel();
            }
        }, 2000);
    }, 10000);
});
```

Tracking Completion Status

By listening for the dashboardCompleted event, you can give information or take action when a dashboard finishes rendering.

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.dashboard({
        resource: "/public/test_dashboard",
        container: "#container",
        events: {
            dashboardCompleted: function() {
                alert("Dashboard is ready!");
            }
        },
```

```
        error: function(error) {
            alert(error);
        },
    });
});
```

Using Dashboard Input Controls

Like reports, dashboard can have input controls, also called filters, to change the values of what is displayed.

Visualize.js dashboard input controls are rendered directly by the server. If a dashboard includes input controls, the dashboard is rendered with a small panel of corresponding UI widgets. When you embed the dashboard, it contains the interface that allows users to set input controls.

You can also set input controls programmatically; for example you might update a dashboard based on other events in your web application. Input controls are called parameters within Visualize.js dashboards. To set input controls programmatically, you must first discover the list of available parameters:

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container",
    success: function() { console.log("dashboard parameters - " + this.data().parameters); },
    error: function(e) { alert(e); }
});
```

Then you read their values, modify them, and set new values. The dashboard then renders with the new input parameter values:

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container",
    params: {
        Country: ["USA", "Mexico", "Canada"]
    },
    error: function(e) { alert(e); }
});

dashboard.params(); // returns { Country: ["USA", "Mexico", "Canada"] }
...
dashboard.params({ month: ["2"] }).run();
dashboard.params(); // returns { month: ["2"] }
```

Parameters are always sent as arrays of quoted string values, even if there is only one value, such as ["USA"]. This is also the case even for single value input such as numerical, boolean, or date/time inputs. You must also use the array syntax for single-select values as well as multi-select parameters with only one selection. No matter what the type of input, always set its value to an array of quoted strings.

The following values have special meanings:

- "" – An empty string, a valid value for text input and some selectors.
- "~NULL~" – Indicates a NULL value (absence of any value), and matches a field that has a NULL value, for example if it has never been initialized.
- "~NOTHING~" – Indicates the lack of a selection. The meaning depends on the type of parameter:
    - In multi-select parameters, this is equivalent to indicating that nothing is deselected, thus all are selected.
    - In a single-select non-mandatory parameter, this corresponds to no selection (displayed as ---).
    - In a single-select mandatory parameter, the lack of selection makes it revert to its default value.

In the following example, a button resets the parameters to their default values by sending an empty parameter set (params({})). First the HTML to define the container and the button:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<button>Reset params</button>
<br/><br/>

<div id="container"></div>
```

And then the JavaScript to perform the action:

```
function handleError(e) {
    alert(e);
}

visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
```

```
}, function (v) {
    var dashboard = v.dashboard({
        resource: "/public/Samples/Dashboards/1._Supermart_Dashboard",
        error: handleError,
        container: "#container",
        params: {
            Store_Country: ["Mexico"],
        }
    });

    $("button").click(function() {
        dashboard.params({}).run();
    });
});
```

In another example, the script initializes the parameters and the HTML displays a button when they're ready to be applied:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<br/>
<button disabled>Apply params</button>
<br/><br/>

<div id="container"></div>
```

And then the JavaScript to initialize the parameters and enable the button for the user:

```
function handleError(e) {
    alert(e);
}

visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
}, function (v) {
    var initialParams = {
        Country: ["USA", "Canada"]
    };

    var dashboard = v.dashboard({
        resource: "/public/Samples/Dashboards/3.2_Inventory_Metrics",
        container: "#container",
        error: handleError,
        params: initialParams,
        success: function() {
            $("button").prop("disabled", false);
            buildParamsInput();
        }
    });
```

```
    function buildParamsInput() {
        var params = dashboard.data().parameters;

        for (var i = params.length-1; i >= 0; i--) {
            var $el = $("<div>" + params[i].id + ": <input type='text' data-paramId='" + params
[i].id + "'/></div>");

            $("body").prepend($el);

            $el.find("input").val(dashboard.params()[params[i].id]);
        }
    }

    $("button").on("click", function() {
        var params = {};

        $("[data-paramId]").each(function() {
            params[$(this).attr("data-paramId")] = $(this).val().indexOf("[") > -1 ? JSON.parse
($(this).val()) : [$(this).val()];
        });

        $("button").prop("disabled", true);

        dashboard.params(params).run()
            .fail(handleError)
            .always(function() { $("button").prop("disabled", false); });
    });
});
```

You can create any number of user interfaces, database lookups, or your own calculations to provide the values of parameters. Your parameters could be based on 3rd party API calls that get triggered from other parts of the page or other pages in your app. When your dashboards can respond to dynamic events, they become truly embedded and much more relevant to the user.

Using the Dashboard Undo Stack

Visualize.js dashboards provide undo and redo functionality. After users interact with input controls and modify the data on the dashboard, they may want to return to their initial dashboard state. With the undo and redo events and actions, you can allow your users to navigate their data more easily in dashboards.

Visualize.js dashboards support the following actions:

- Undo – Reverts the input controls to next older set of values and updates the dashboard contents accordingly; available after at least one change to input controls.
- Redo – Reverts the input controls to next newer set of values and updates the dashboard contents accordingly; available after at least one undo action.

- Undo All – Reverts the input controls to their initial set of values and updates the dashboard contents accordingly; available when there is an older set of values that hasn't been undone.

In order to implement undo and redo actions, you must also use event listeners to know when undo and redo events become available. For example, before the user has made any changes to input controls, no actions are possible. Redo is only possible after the user performs an undo or undo-all action.

In the following example, the HTML has a table to display the input control values and buttons to perform the undo and redo actions:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<table id="params">
  <tr>
    <td>Parameter ID</td>
    <td>Set value</td>
  </tr>
</table>

<br/>
<button id="apply" disabled>Apply params</button>
<button id="undo" disabled>Undo</button>
<button id="undoAll" disabled>Undo All</button>
<button id="redo" disabled>Redo</button>
<br/>
<br/>

<div id="container"></div>
```

The CSS sets the size of the container to be large enough for a dashboard, and sets the table heading color for visibility:

```
#container {
  width: 1000px;
  height: 700px;
}

th {
  background-color: blue;
}
```

The JavaScript has listeners for the canUndo and canRedo events in order to set the visibility of the undo and redo buttons. When the buttons are enabled and the user clicks on one, the code takes the corresponding action then displays the ids and values of the input controls in the table. By combining the event listeners and actions, the page allows the user to step backward and forward through various sets of input controls in the dashboard.

```
visualize({
    auth: {...
    }
}, function(v) {
    var dashboard = v.dashboard({
        resource: "/public/Samples/Dashboards/2._Performance_Summary_Dashboard",
        container: "#container",
        error: handleError,
        success: function() {
            $("#apply").prop("disabled", false);
            buildParamsInput();
        },
        events: {
            canUndo: function(val) {
                $("#undo").prop("disabled", !val);
                $("#undoAll").prop("disabled", !val);
            },
            canRedo: function(val) {
                $("#redo").prop("disabled", !val);
            }
        }
    })

    $("#undo").on("click", function() {
        dashboard.undoParams().fail(handleError);
    });

    $("#undoAll").on("click", function() {
        dashboard.undoAllParams().fail(handleError);
    });

    $("#redo").on("click", function() {
        dashboard.redoParams().fail(handleError);
    });


    $("#apply").on("click", function() {
        var params = {};

        $("[data-paramId]").each(function() {
            params[$(this).attr("data-paramId")] = $(this).val().indexOf("[") > -1 ? JSON.parse
($(this).val()) : [$(this).val()];
        });

        $("#apply").prop("disabled", true);

        dashboard.params(params).run()
            .fail(handleError)
            .always(function() {
                $("#apply").prop("disabled", false);
                fillWithValues();
            });
    });

    function buildParamsInput() {
        var params = dashboard.data().parameters;

        for (var i = params.length - 1; i >= 0; i--) {
            var $el = $("<tr><td>" + params[i].id + "</td><td><input type='text' data-paramId='" +
params[i].id + "'/></td></tr>");
            $("#params").append($el);
            $el.find("input").val(params[i].value);
        }
    }
```

```
        function handleError(e) {
            alert(e);
        }
    });
```

Setting Dashboard Hyperlink Options

Visualize.js provides several types of hyperlinks to handle most use cases:

- Reference – The reference link indicates an external source that is identified by a normal URL. The only expression required is the hyperlink reference expression. It's possible to specify additional parameters for this hyperlink type.
- LocalAnchor – To point to a local anchor means to create a link between two locations into the same document. It can be used, for example, to link the titles of a summary to the chapters to which they refer. To define the local anchor, it is necessary to specify a hyperlink anchor expression, which will have to produce a valid anchor name. It's possible to specify additional parameters for this hyperlink type.
- LocalPage – If instead of pointing to an anchor you want to point to a specific current report page, you need to create a LocalPage link. In this case, it is necessary to specify the page number you are pointing to by means of a hyperlink page expression (the expression has to return an Integer object). It's possible to specify additional parameters for this hyperlink type.
- RemoteAnchor – If you want to point to a particular anchor that resides in an external document, you use the RemoteAnchor link. In this case, the URL of the external file pointed to will have to be specified in the Hyperlink Reference Expression field, and the name of the anchor will have to be specified in the Hyperlink Anchor Expression field. It's possible to specify additional parameters for this hyperlink type.
- RemotePage – This link allows you to point to a particular page of an external document. Similarly, in this case the URL of the external file pointed to, will have to be specified in the Hyperlink Reference Expression field, and the page number will have to be specified by means of the hyperlink page expression. Some export formats have no support for hypertext links. It's possible to specify additional parameters for this hyperlink type.

- ReportExecution – This type of hyperlink is used to implement drill-down. Page and anchor can be specified for the hyperlink type as well as additional special parameters such as _report, _anchor, _page, `_output`.
- AdHocExecution – This type of hyperlink represents an information about clicked point on chart reports generated from AdHoc Charts. It exposes names of measures and values of dimensions as parameters.
- Custom Hyperlink Type – A type of hyperlink that you can define entirely.

And there are several types of link targets:

- Self – This is the default setting. It opens the link in the current window.
- Blank – Opens the target in a new window. Used for output formats such as HTML and PDF
- Top – Opens the target in the current window but outside any frames. Used for output formats such as HTML and PDF.
- Parent – Opens the target in the parent window (if available). Used for output formats such as HTML and PDF.
- Frame name – Always opens the target in the specified frame.

The following table shows the new default action for each combination of link and target:

Type \ Targets	Self	Blank	Top	Parent
Reference (points to an external resource)	Referenced URL is opened in an iframe on top of the report.	Referenced url is opened in new tab.	Referenced url is opened in same window.	Referenced url is opened in parent frame for reports generated from AdHoc Charts.
ReportExecution (points to JasperReports report)	Referenced report is opened in same place through Viz.js (if hyperlink points to the same report, it's just gets updated with	Referenced report is opened in new tab.	Referenced report is opened in top frame (same window).	Referenced report is opened in parent frame.

Type \ Targets	Self	Blank	Top	Parent
	params/page/anchor).			
LocalAnchor (points to an anchor in current report)	Anchor is opened in same place through Viz.js.	Referenced report anchor is opened in new browser tab.	Referenced report anchor is opened in top frame (same window).	Referenced report anchor is opened in parent frame.
LocalPage (points to a page in current report)	Page is opened in same place through Viz.js.	Referenced report page is opened in new browser tab.	Referenced report page is opened in top frame (same window).	Referenced report page is opened in parent frame.
RemoteAnchor (points to an anchor in remote document)	Since remote anchor could be a link to resources like PDF or HTML, we have to open it in an iframe on top of the report.	Referenced report anchor is opened in new browser tab.	Referenced report anchor is opened in top frame (same window).	Referenced report anchor is opened in parent frame.
RemotePage (points to a page in remote document)	Since remote page could be a link to resources like PDF or HTML we have to open it in an iframe on top of the report.	Referenced report page is opened in new browser tab.	Referenced report page is opened in top frame (same window).	Referenced report page is opened in parent frame.
AdHocExecution	The hyperlink provides parameters, which are exposed to Filter Manager for additional wiring (beyond the scope of this document).			

The following code samples demonstrate the default handling of hyperlinks and the effect of defining overrides that remove the default handling. First the HTML for some buttons:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<button id="btn1">Extend default hyperlink behavior</button>
<button id="btn2">Disable hyperlink click events</button>
<div id="container"></div>
```

And now some hyperlink overrides:

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {
    var dashboard = v.dashboard({
        resource: "/public/Dashboard_with_different_hyperlinks_with_self_target",
        container: "#container",
        error: function(e) {
            alert(e);
        }
    });

     document.querySelector("#btn1").addEventListener("click", function() {
        dashboard.linkOptions({
            events: {
                click: function(ev, link, defaultHandler) {
                    alert("before default handler");
                    defaultHandler();
                    alert("after default handler");
            }}
        }).run();
    });
```

```
    document.querySelector("#btn2").addEventListener("click", function() {
        dashboard.linkOptions({
            events: {
                click: function(ev, link, defaultHandler) {
                    alert("default handler will not be called");
            }}
        }).run();
    });
});
```

However, if your dashboards are designed for custom drill-down, you can still define custom link handling so your users can access more reports and more data. Be sure to modify your code to handle all three function parameters for function(ev, link, default), as shown in this updated code sample:

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container",
```

```
    linkOptions: {
        beforeRender: function (linkToElemPairs) {
            linkToElemPairs.forEach(showCursor);
        },
        events: {
            "click": function(ev, link, default){
                if (link.type == "ReportExecution") {
                    if ("monthNumber" in link.parameters) {
                        v("#drill-down").report({
                            resource: link.parameters._report,
                            params: {
                                monthNumber: [link.parameters.monthNumber]
                            }
                        });
                    }
                }
        }}
    }
});

function showCursor(pair){
    var el = pair.element;
    el.style.cursor = "pointer";
}
```

Exporting From a Dashboard

Like a report, you can export a dashboard by invoking its export function and specifying the outputFormat and detailed property. You must wait until the dashboard's run action has completed and returns success before starting the export. The following export types and formats are supported:

- Screenshot: "pdf", "png", "docx", "pptx", "odt"
- Detailed: "pdf", "xlsx" "csv", "docx", "rtf" "odt" "ods" " Excel" "pptx"

In the following examples, the HTML page has a container for the dashboard and a button for the export, and the CSS configures a simple loading animation:

```
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="button">Export</button>
<div id="container">
    <div id="dashboard">
      <div><p class="loader">Loading...</p></div>
    </div>
</div>
```

CSS:

```
#container {
    width: 1110px;
    height: 630px;
}

/* You can replace the following with your own custom loading indicator */
@import "compass/css3";

@keyframes fadeIn {
  from { opacity: 0; }
}

.loader{
  position: relative;
  top: 50px;
  right: 250px;

  font: italic bold 1.2em/1 Bodoni, serif;
  color: #555;
  text-align: center;

  animation: fadeIn 1s infinite alternate;
}
```

The detailed property determines the export type. You can define the detailed property as follows:

- For the detailed export type - detailed: true
- For the screenshot export type - detailed: false

The default value is false. If you do not provide any value for the detailed property, it is considered false (default) and the dashboard gets exported as a screenshot.

In the first JavaScript example below, the button exports the PDF of the dashboard in the detailed export type, however the button is disabled until the dashboard has finished running and returned success. When the button is enabled and clicked, the export function is called with the "pdf" format in the detailed export type. When the export is ready, the PDF file is made available to download in the browser.

```
visualize({
  auth: { ...
  }
}, function(v) {
  var dashboard = v.dashboard({
      resource: "/public/Samples/Dashboards/3.2_Inventory_Metrics",
      container: "#container",

      success: function() {
        button.removeAttribute("disabled");
      },

      error: function(error) {
        console.log(error);
```

```
      }
    });

  // Export button handler.
  $("button").click(function() {
    dashboard.export({
      outputFormat: "pdf", detailed: true
    },

    // Export success callback.
    function(link) {
      console.log('link add', link);
      var url = link.href ? link.href : link;
      window.location.href = url;
    },

    // Export error callback.
    function(error) {
      console.log(error);
    });
  });
});
```

Visualize.js also exposes the list of available export formats. The following example uses this list to build a drop-down selector of export formats, and the chosen format is passed to the export function when the Export button is clicked.

To get the list of all supported export formats, use the following functions:

- For the detailed export type: v.dashboard.detailedExportFormats
- For the screenshot export type: v.dashboard.exportFormats

This example uses the function for the screenshot export type. This example uses the same HTML and CSS code shown above.

```
visualize({
  auth: { ...
  }
}, function(v) {
  var $select = buildControl("Export to: ", v.dashboard.exportFormats),
    $button = $("#button"),
    dashboard = v.dashboard({
      resource: "/public/Samples/Dashboards/3.2_Inventory_Metrics",
      container: "#container",

      success: function() {
        button.removeAttribute("disabled");
      },

      error: function(error) {
        console.log(error);
      }
    });
```

```
  // Export button handler.
  $button.click(function() {
    var selectedFormat = $select.val();

    dashboard.export({
      outputFormat: selectedFormat
    },

    // Export success callback.
    function(link) {
      console.log('link add', link);
      var url = link.href ? link.href : link;
      window.location.href = url;
    },

    // Export error callback.
    function(error) {
      console.log(error);
    });
  });

  // Builds drop-down selector with the list of supported export formats.
  function buildControl(name, options) {
    function buildOptions(options) {
      var template = "<option>{value}</option>";
      return options.reduce(function(memo, option) {
        return memo + template.replace("{value}", option);
      }, "")
    }

    var template = "<label>{label}</label><select>{options}</select><br>",
      content = template.replace("{label}", name).replace("{options}", buildOptions(options));

    var $control = $(content);
    $control.insertBefore($("#button"));

    return $($control[1]);
  }
});
```

Closing a Dashboard

When you want to reuse a container for other contents and free the dashboard resources, use the destroy function to close it.

```
var dashboard = v.dashboard({
    resource: "/public/test_dashboard",
    container: "#container"
});

dashboard.destroy();
```

API Reference - adhocView

The adhocView function displays an Ad Hoc view that allows users to interact with tables, crosstabs, and charts in the target container.

As of JasperReports Server 7.0, Visualize.js can render interactive Ad Hoc views in your web apps. In Visualize.js, only the table, crosstab, or chart of the Ad Hoc view is displayed, not the fields or filter panels. The server generates the user interface (UI) of an Ad Hoc view. You can also customize the appearance and behavior of the Ad Hoc view by accessing hyperlinks and setting event listeners.

This chapter contains the following sections:

- Ad Hoc View Properties
- Ad Hoc View Functions
- Ad Hoc View Data Structure
- Rendering an Ad Hoc View
- Getting the Embed Code of an Ad Hoc View
- Setting the Visualization Type
- Setting Ad Hoc View Filters
- Accessing Ad Hoc View Hyperlinks

Ad Hoc View Properties

The properties structure passed to the adhocView function determines the view to be displayed and its initial state. It is defined as follows:

```
{
  "type": "object",
  "properties": {
    "server": {
      "type": "string",
      "description": "Url to JRS instance."
```

```
    },
    "resource": {
      "type": "string",
      "description": "Report resource URI.",
      "pattern": "^/[^/~!#\\$%^|\\s`@&*()\\-+={}\\[\\]:;\"'<>,?/\\|\\\\]+(/[^/~!#\\$%|\\s`@&*()+=
{}\\[\\]:;\"'<>,?/\\|\\\\]+)*$"
    },
    "container": {
      "oneOf": [
        {
          "type": "object",
          "additionalProperties": true,
          "description": "DOM element to render report to"
        },
        {
          "type": "string",
          "description": "CSS selector for container to render report to."
        }
      ]
    },
    "autoresize": {
      "type": "boolean",
      "description": "Automatically resize Ad Hoc View on browser window resize",
      "default": true
    },
    "canvas": {
      "type": "object",
      "properties": {
        "type" :{
          "description": "Type of visualization",
          "enum": [
            "Table", "Crosstab", "Bar", "Column", "Line", "Area", "Spline",
            "AreaSpline", "StackedBar", "StackedColumn", "StackedLine", "StackedArea",
            "StackedSpline", "StackedAreaSpline", "StackedPercentBar", "StackedPercentColumn",
            "StackedPercentLine", "StackedPercentArea", "StackedPercentSpline",
            "StackedPercentAreaSpline", "Pie", "DualLevelPie", "TimeSeriesLine",
            "TimeSeriesArea", "TimeSeriesSpline", "TimeSeriesAreaSpline", "ColumnLine",
            "ColumnSpline", "StackedColumnLine", "StackedColumnSpline", "MultiAxisLine",
            "MultiAxisSpline", "MultiAxisColumn", "Scatter", "Bubble", "SpiderColumn",
            "SpiderLine", "SpiderArea","HeatMap", "TimeSeriesHeatMap", "SemiPie",
            "DualMeasureTreeMap", "TreeMap", "OneParentTreeMap"
          ]
        },
        "chart": {
          "type": "object",
          "additionalProperties": false,
          "description": "Properties of Ad Hoc View's chart",
          "properties": {}
        }
      }
    },
    "linkOptions": {
      "type": "object",
      "description": "adhocView's parameters values",
      "properties": {
        "beforeRender": {
          "type": "function",
          "description": "A function to process link - link element pairs."
        },
        "events": {
          "type": "object",
          "description": "Backbone-like events object to be applied to JR links",
          "additionalProperties" : true
```

```
        }
      }
    },
    "loadingOverlay": {
      "type": "boolean",
      "description": "Enable/disable report loading overlay",
      "default": true
    },

    "showTitle": {
      "type": "boolean",
      "description": "Option to show/hide Ad Hoc view title",
      "default": true
    },

    "params": {
      "type": "object",
      "description": "Ad Hoc View's parameters values",
      "additionalProperties": {
        "type": "array"
      }
    }
  },
  "required": ["server", "resource"]
}
```

Ad Hoc View Functions

The adhocView function exposes the following functions:

```
/**
 * @param {Object} properties - Ad Hoc View properties
 * @constructor
 */
function adhocView(properties){}

//Special  getters

/**
 * Get any result after invoking run action
 * @returns any data which supported by this bi component
*/
adhocView.prototype.data = function(){};

//Actions

/**
 * Perform main action for bi component
 * Callbacks will be attached to  deferred object.
 *
 * @param {Function} callback - optional, invoked in case of successful run
 * @param {Function} errorback - optional, invoked in case of failed run
 * @param {Function} always - optional, invoked always
 * @return {Deferred} dfd
 */
```

```
adhocView.prototype.run = function(callback, errorback, always){};

/**
 * Render ADV to container, previously specified in property.
 * Clean up all content of container before adding Ad Hoc View's content
 * @param {Function} callback - optional, invoked in case successful export
 * @param {Function} errorback - optional, invoked in case of failed export
 * @param {Function} always - optional, optional, invoked always
 * @return {Deferred} dfd
 */
adhocView.prototype.render = function(callback, errorback, always){};

/**
 * Refresh ADV
 * @param {Function} callback - optional, invoked in case of successful refresh
 * @param {Function} errorback - optional, invoked in case of failed refresh
 * @param {Function} always - optional, invoked optional, invoked always
 * @return {Deferred} dfd
 */
adhocView.prototype.refresh = function(callback, errorback, always){};

/**
 * Destroy ADV representation if any, leave only
 * properties
 * @param {Function} callback - optional, invoked in case of successful cleanup
 * @param {Function} errorback - optional, invoked in case of failed cleanup
 * @param {Function} always - optional, invoked optional, invoked always
 * @return {Deferred} dfd
 */
adhocView.prototype.destroy = function(callback, errorback, always){};

return adhocView;
```

Ad Hoc View Data Structure

The data object of an adhocView contains metadata about the Ad Hoc view:

```
{
  "title": "Ad Hoc View Data",
  "description": "A JSON Schema describing Ad Hoc View Data",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties":{
```

```
    "metadata": {
      "type": "object",
      "description": "Set of available parameters",
      "properties": {
        "availableVisualizationTypes": {
          "type": "array",
          "description": "List of available visualization types, which can be setted as
```

```
canvas.type",
          "enum": ["Crosstab", "Table", "Column", "StackedColumn",
                  "StackedPercentColumn", "Bar", "StackedBar",
                  "StackedPercentBar", "SpiderColumn", "Line",
                  "Spline", "Area", "StackedArea", "StakeckedPrecentArea",
                  "AreaSpline", "SpiderLine", "SpiderArea",
                  "Pie", "SemiPie", "HeatMap"]
        },
        "inputParameters": {
          "type": "array",
          "description": "List of available filters",
          "items": {
            "type": "object",
            "properties": {
              "id": {
                "type": "string",
                "description": "Unique id which can be used as filter name"
              },
              "type": {
                "type": "string",
                "description": "Type of filter data",
                "enum": ["boolean", "byte", "short", "integer", "bigInteger", "timestamp",
                        "float", "bouble", "bigDecimal", "string", "date", "time"]
              }
            },
            "required": ["id"]
          }
        },
        "outputParameters": {
          "type": "array",
          "description": "List of expected fields in hyperlink"
          "items": {
            "type": "object",
            "properties": {
              "id": {
                "type": "string",
                "description": "Name of field in hyperlink"
              },
              "type": {
                "type": "string",
                "description": "Type of field data",
                "enum": ["boolean", "byte", "short", "integer", "bigInteger", "timestamp",
                        "float", "bouble", "bigDecimal", "string", "date", "time"]
              }
            },
            "required": ["id", "type"]
          }
        }
      }
    },
  },
  "required": ["metadata"]
}
```

Rendering an Ad Hoc View

As with reports and dashboards, Ad Hoc views run on the server and are rendered in a container on your page. First, you create an adhocView object and then set its properties. As with other objects, the resource property determines which Ad Hoc view to run, and the container property determines where it appears on your page.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer",
    success: function() { console.log("rendered"); },
    error: function(e) { alert(e); }
});
```

Ad Hoc views are displayed in Visualize.js in the same state they are saved on the server. Only the table, crosstab, or chart of an Ad Hoc view is rendered in Visualize.js, not the design elements. Visualize.js is intended to display data, not access the underlying data structures used to create the Ad Hoc view.

However, the rendering of an Ad Hoc view includes the ••• icon in the top left corner that allows users to switch between table, crosstab, and chart type. For example, if the Ad Hoc view appears as a table, the user can switch the view to a crosstab or chart. The user may also change the type of chart, though not all chart types work with the data in a given Ad Hoc view.

If you wish to disable the chart selector, specify the following property on the container:

```
#AdHocContainer .jr-mAdhoc-visualization-launcher.jr {
  display: none;
}
```

Unlike a report that is meant to have a static layout and look the same every time it runs, an Ad Hoc view is more of a data exploration tool that users edit often and may save in an intermediate state. In order to have consistent results, you should design and save specific Ad Hoc views that are relevant to your visualize.js user, and not allow other users on the server to modify them.

Alternatively, you can use the dynamic nature of Ad Hoc views, for example to explicitly display the latest data visualization that has been designed and saved on the server. In this case, your Visualize.js app should label the container so that your users know the content of the view may change.

In the following example, the adhocView object is created and initialized first, and then it runs and renders later on demand in the specified container.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    runImmediately: false
});

ahv
    .run()
    .done(function() {
        this
            .container("#AdHocContainer")
```

```
            .render()
            .done(function() { console.log("rendered"); })
            .fail(function(e) { alert(e); });
    })
    .fail(function(e) { alert(e); });
```

Of course, you can add styles to change the size and placement of your Ad Hoc view container, as shown in the following CSS sample.

```
#AdHocContainer {
  width: 800px;
  height: 400px;
  padding-left: 50px;
  position: relative;
}
```

There are two more functions to manage your Ad Hoc view. Refresh will update the rendered view, and destroy will remove it, leaving only the adhocView object with its properties set.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer"
});

ahv.refresh();

...

ahv.destroy();
```

Getting the Embed Code of an Ad Hoc View

As of JasperReports Server 7.9, the server can provide the code to embed any Ad Hoc view displayed in the Ad Hoc Editor. This lets you browse the repository, preview an Ad Hoc view, and get its basic embed code. You can paste this code directly in your application, and then edit it for your needs. You can also open the view's embed code in JSFiddle to see the effect of your edits in real time, then copy the final code from JSFiddle.

To copy the embed code of an Ad Hoc view:

1. Log into JasperReports Server and browse the repository to find the Ad Hoc view you want to embed.
2. Open the Ad Hoc view so that it is displayed in the server's Ad Hoc Editor.

3. Click the </> icon in the menu bar to view the embed code. If this icon is not visible in the menu bar, click the icon to toggle design mode first.
4. The Ad Hoc Embed Code dialog shows you the Visualize.js code and a preview of the view as it is currently saved. Select Copy Code to copy the entire code block to your clipboard. You can also highlight selected parts of the code and use Ctrl-C (Command-C on Mac OS), for example if you want only the main function.

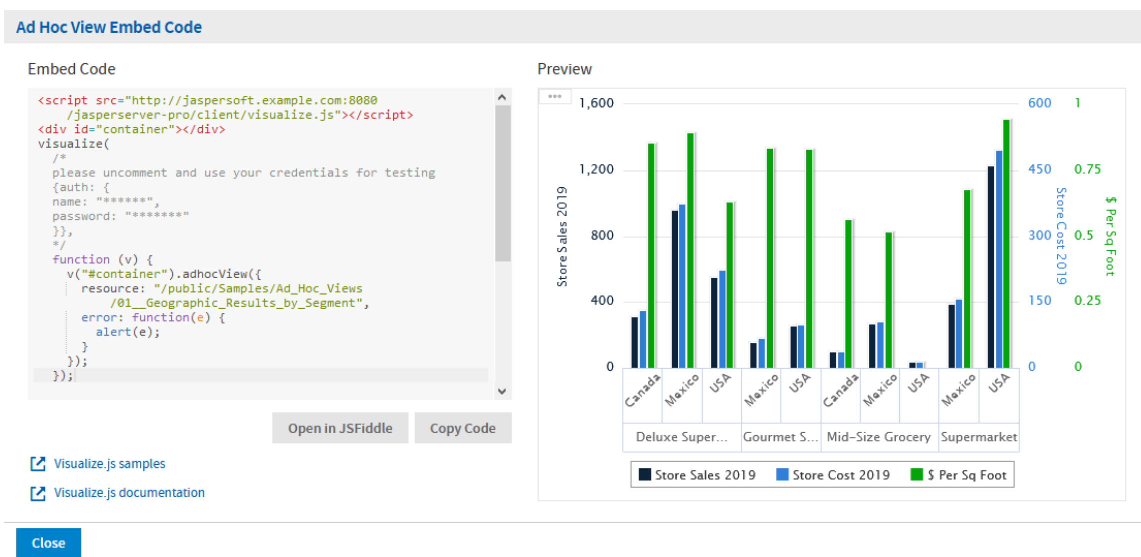

Figure 3: The Embed Code of an Ad Hoc View

The code sample includes comments where you can enter credentials for authentication. You should also change the name of the container to match the one in your application.

5. Alternatively, select Open in JSFiddle to load the same code into a new Fiddle, an online JavaScript viewer and interactive editor. This lets you modify the JavaScript or HTML, and add CSS if desired, then see the results in real time.

As shown by the preview, Visualize.js displays the table, crosstab, or chart of the Ad Hoc views without any design elements such as data selection panel, layout band, or filter panel. Regardless of any changes you make in the Ad Hoc editor, Visualize.js always displays the latest saved version of an Ad Hoc view, as determined by the repository URL in the embed code.

When displayed through Visualize.js, an Ad Hoc view includes the ••• icon in the top left corner that allows users to switch between table, crosstab, and chart type, though not all chart types work with the data in a given Ad Hoc view. This selector changes the

appearance of the Ad Hoc view in the user's current session, but the changes can't be saved. For every new session, Visualize.js displays the default appearance of the Ad Hoc view.

Setting the Visualization Type

Visualize.js allows your code to also interact programmatically with the Ad Hoc view, to customize its appearance in response to other events or inputs. The following sections give examples of how to update the appearance of the Ad Hoc view through your code.

An Ad Hoc view has a default visualization type, either table, crosstab, or chart that is determined by its creator and saved in the repository. When you run and render an Ad Hoc view, your users will see the visualization type of the view as it was last saved. Through Visualize.js, you can change the visualization type, for example so that users first see a crosstab or a specific type of chart.

The same data can usually be displayed in tables, crosstabs, and charts, but it often appears slightly differently. Different charts require different data series, so not all chart types may be compatible with the data that is currently selected in the Ad Hoc view. Before changing the visualization type, you should make sure it is compatible with your Ad Hoc view.

The following example shows how your code can set the visualization type by modifying the canvas object of your adhocView object during initialization.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer",
    canvas: {
        type: "Pie"
    },
    error: function(e) { alert(e); }
});
```

You can also let the Ad Hoc view render with the default visualization and then change to a different the visualization type at a later time:

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer",
    error: function(e) { alert(e); }
});
...
ahv.canvas({
        type: "Crosstab"
}).run();
```

The available visualization types and charts are listed by the type property of the canvas object:

```
    "canvas": {
      "type": "object",
      "properties": {
        "type" :{
          "description": "Type of visualization",
          "enum": [
            "Table", "Crosstab", "Bar", "Column", "Line", "Area", "Spline",
            "AreaSpline", "StackedBar", "StackedColumn", "StackedLine", "StackedArea",
            "StackedSpline", "StackedAreaSpline", "StackedPercentBar", "StackedPercentColumn",
"StackedPercentLine",
            "StackedPercentArea", "StackedPercentSpline", "StackedPercentAreaSpline", "Pie",
            "DualLevelPie", "TimeSeriesLine", "TimeSeriesArea", "TimeSeriesSpline",
            "TimeSeriesAreaSpline", "ColumnLine", "ColumnSpline", "StackedColumnLine",
            "StackedColumnSpline", "MultiAxisLine", "MultiAxisSpline", "MultiAxisColumn",
            "Scatter", "Bubble", "SpiderColumn", "SpiderLine", "SpiderArea","HeatMap",
"TimeSeriesHeatMap",
            "SemiPie", "DualMeasureTreeMap", "TreeMap", "OneParentTreeMap"
          ]
        },
```

Setting Ad Hoc View Filters

Another way to interact programmatically with Ad Hoc views is to get and set any filters that are defined in the view. Note that unlike a report, the set of available filters in an Ad Hoc view may change if the view is modified on the server. Once an Ad Hoc view is rendered, your code should read the current filters and values before setting them.

As with reports and dashboards, filter values are called parameters and exposed through the params object. In the following example, the Country filter is known to be saved in the Ad Hoc view, so it can be set during initialization of the view, just before it is run and rendered.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer",
    params: {
        Country: ["USA", "Mexico", "Canada"]
    },
    error: function(e) { alert(e); }
});

// read back parameter/filter values
ahv.params(); // returns { Country: ["USA", "Mexico", "Canada"] }

// if you know that filters won't change, you can set new values
ahv.params({ Country: ["Canada"] }).run();
```

You can use the metadata.inputParameters property of the data object to obtain the name and type of each current filter. The structure of the metadata.inputParameters is shown in Ad Hoc View Data Structure.

In the next example, the code requests the current filter values, processes them, and then uses the information about the current filters and values to set different values, in this case a different selection:

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    container: "#AdHocContainer",
    error: function(e) { alert(e); }
});

// read current paramter/filter names
currentFilters = ahv.data().metadata.inputParameters;

// read current parameter/filter values
currentValues = ahv.params(); // returns { State: ["California", "Oregon", "Washington"] }

// process current filters here
// now set a new filter value
ahv.params({ State: ["Oregon"] }).run();
```

Accessing Ad Hoc View Hyperlinks

Hyperlinks, or simply links, are elements of the Ad Hoc view that your code can interact with. These are generally the contents of cells in a table, for example a field name in a column header or the value in a cell. As the user interacts with the Ad Hoc View, you can capture events on these elements such as clicks and then take action using the values of the element.

The elements that you can access are called outputParameters, and their structure is defined in the Ad Hoc View Data Structure. You can use the metadata.outputParameters property of the data object to obtain the name and type of each hyperlink.

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView"
    container: "#AdHocContainer",
    success: function () {
      console.log(ahv.data().metadata.outputParameters); // Expected fields in hyperlink
    }
});

ahv.canvas({
        type: "Crosstab"
}).run();
```

Using the linkOptions properties, you can access the events on links, use the link values, and take other actions. The following example shows how to add a listener that displays link values in the console:

```
var ahv = v.adhocView({
    resource: "/public/Sample_AdHocView",
    linkOptions: {
        events: {
            click: function (el, link, defaultHandler, tableLink) {
                console.log(link); // prints the link data
                console.log(tableLink); // prints the table's link data
            },
            mouseover: function (el, link, defaultHandler, tableLink) {},
            mouseenter: function (el, link, defaultHandler, tableLink) {},
            mouseleave: function (el, link, defaultHandler, tableLink) {}
        }
    }
});
```

You can use the beforeRender property of the linkOptions to modify the appearance of elements in the Ad Hoc view. In the following example, every element is printed to the log and given the same color, but you could implement logic to highlight high or low values based on other thresholds.

```
var ahv = v.adhocView({
    resource: "/public/Sample",
    linkOptions: {
        beforeRender: function (linkToElemPairs) {
            linkToElemPairs.forEach(function (pair) {
                console.log(pair.element); // prints html element reference
                console.log(pair.link); // prints link's data related to current element

                pair.element.style.backgroundColor = "lightblue"; // set lightblue background
color to each hyperlink
            });
        }
    }
});
```

The following structure defines what properties are available for values in an Ad Hoc table, also called an Ad Hoc hyperlink:

```
{
    "title": "Adhoc Hyperlink",
    "description": "A JSON Schema describing embeddable adhoc hyperlink",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties":{
        "Measures": {
            "type": "array",
            "description": "An array of measure names",
            "items": {
```

```
                "type": "string",
                "description": "Measure name"
            }
        }
    },
    "additionalProperties": {
        "type": "string",
        "description": "Key is name of field and value is value of it"
    }
}
```

The following structure defines what properties are available for the table element of an Ad Hoc view, also called an Ad Hoc Table hyperlink. In particular, this structure gives your code access to the rows, columns, groups, and aggregation information about the Ad Hoc table:

```
{
    "title": "Extended Adhoc Hyperlink",
    "description": "An Extended JSON Schema describing embeddable adhoc hyperlink",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties":{
        "id": {
            "type": "string",
            "description": "Hyperlink id, reflected in corresponding attribute in DOM"
        },
        "linkType": {
            "type": "string",
            "description": "Hyperlink type",
            "enum": ["HEADER", "DATA", "GROUP_LABEL", "GROUP_TOTAL", "GRAND_TOTAL"]
        },
        "value": {
            "type": "string",
            "description": "Hyperlink value, which is displayed to user"
        },
        "column": {
            "type": "object",
            "description": "Map of aggregated information related to whole column",
            "properties": {
                "aggregatedFiledReferences": {
                    "type": "array",
                    "items": {
                        "type": "string",
                        "description": "Measure name"
                    }
                },
                "aggregationFormat": {
                    "type": "string",
                    "description": "Format which should be used to display aggregated values"
                },
                "aggregationFormatId" : {
```

```
                "type": "string",
                "description": "Unique key of aggregation format"
            },
            "aggregationType": {
                "type": "string",
                "description": "Type for aggregated value"
            },
            "defaultFunctionName": {
                "type": "string",
                "description": "Default aggregation function name"
            },
            "detailFieldReference": {
                "type": "string",
                "description": "Reference name to get more details of field"
            },
            "dimension": {
                "type": "string",
                "description": "Dimension name"
            },
            "field": {
                "type": "string",
                "description": "field name"
            },
            "firstLevelExpansion": {
                "type": "string",
                "description": ""
            },
            "format": {
                "type": "string",
                "description": "Format which should be used to display value"
            },
            "formatId": {
                "type": "string",
                "description": "Unique format id"
            },
            "functionName": {
                "type": "string",
                "description": "Name of aggregation function which is set by user"
            },
            "hierarchicalName": {
                "type": "string",
                "description": "Field name in schema"
            },
            "horizontalAlign": {
                "type": "string",
                "description": "Cell's horizontal alignment",
                "enum": ["Left", "Center", "Right"]
            },
            "id": {
                "type": "string",
                "description": "Reference name to get more details of field"
            },
            "includeAll": {
                "type": "boolean",
                "description": "Identify that this is total value"
            },
            "index": {
                "type": "number",
                "description": "Column index. Starts form 0"
            },
            "initialValue": {
```

```
        },
                "width": {
                    "type": "number",
                    "description": "Column width"
                }
            }
        },
        "group": {
            "type": "array",
            "description": "Grouping sequence",
            "items": {
                "type": {
                    "$ref": "#/properties/column"
                 }
            }
        },
```

```
"row": {
            "type": "object",
            "description": "Items related to whole row",
            "properties": {
                "index": {
                    "type": "number",
                    "description": "Absolute row index in dataset"
                },
                "relativeIndex": {
                    "type": "number",
                    "description": "Relative row index. Index of currently rendered data"
                },
                "cells": {
                    "type": "array",
                    "description": "Cell in row",
                    "items": {
                        "type": {
                            "$ref": "#/properties/column"
                        }
                    }
                }
            }
        }
    },
    required: ["id"]
}
```

The following example displays a lot of information about an Ad Hoc view in the console. You can use a generic script like this to examine your Ad Hoc views and determine which fields and data you can use to add interactive features.

```
visualize({
  auth: {
    name: "superuser",
    password: "superuser"
  }
}, function(v) {
  var ahv = v.adhocView({
```

```
    resource: "/public/Samples/Ad_Hoc_Views/01__Geographic_Results_by_Segment",
    container: "#container",
    success: function() {
      console.log("All data available:");
      console.log(ahv.data());

      console.log("Available filters:");
      console.log(determineFilters(ahv.data().metadata.inputParameters));

      console.log("Expected fields in data on click:");
      console.log(determineFilters(ahv.data().metadata.outputParameters));
    },
    error: function(e) {
      console.log(e);
    },
    linkOptions: {
      events: {
        click: function(ev, data, defaultHandler, extendedData) {
          console.log("Click");
          console.log(data, extendedData);
        }
      }
    }
  });
});

function determineFilters(parameters) {
  var res = [];

  for (var i = 0; i < parameters.length; i++) {
    res.push(parameters[i].id);
  }
  return res;
}
HTML:
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>
<div id="container"></div>
```

API Reference - Errors

This chapter describes common errors and explains how to handle them with Visualize.js.

This chapter contains the following sections:

- Error Properties
- Common Errors
- Catching Initialization and Authentication Errors
- Catching Search Errors
- Validating Search Properties
- Catching Report Errors
- Catching Input Control Errors

- Validating Input Controls

Error Properties

The properties structure for Generic Errors is defined as follows:

```
{
    "title": "Generic Errors",
    "description": "A JSON Schema describing Visualize Generic Errors",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "errorCode": {
           "type": "string"
        },
        "message": {
            "type": "string"
        },
        "parameters":{
            "type": "array"
        }
    },
    "required": ["errorCode", "message"]
}
```

Common Errors

The following table lists common errors, their messages, and causes.

Error	Message - Description
Page or app not responding	{no_message} - If your page or web application has stopped working without notification or errors, check that the server providing visualize.js is accessible and returning scripts.
unexpected.error	An unexpected error has occurred - In most of cases this is either a JavaScript exception or an HTTP 500 (Internal Server Error) response from server.
schema.validation.error	JSON schema validation failed: {error_message} - Validation against schema has failed. Check the validationError property in object for more details.

Error	Message - Description
unsupported. configuration.error	{unspecified_message} - This error happens only when isolateDom = true and defaultJiveUi.enabled = true. These properties are mutually exclusive.
authentication.error	Authentication error - Credentials are not valid or session has expired.
container.not.found.error	Container was not found in DOM - The specified container was not found in the DOM:error.
report.execution.failed	Report execution failed - The report failed to run on the server.
report.execution.cancelled	Report execution was canceled - Report execution was canceled.
report.export.failed	Report export failed - The report failed to export on the server.
licence.not.found	JRS missing appropriate license - The server's license was not found.
licence.expired	JRS missing appropriate license - The server's license has expired.
resource.not.found	Resource not found in Repository - Either the resource doesn't exist in the repository or the user doesn't have permissions to read it.
export.pages.out.range	Requested pages {0} out of range - The user requested pages that don't exist in the current export.
input.controls. validation.error	{server_error_message} - The wrong input control params were sent to the server.

Catching Initialization and Authentication Errors

Visualize.js is designed to have many places where you can catch and handle errors. The visualize function definition, as shown in Contents of the Visualize.js Script, is:

```
function visualize(properties, callback, errorback, always){}
```

During initialization and authentication, you can handle errors in the third parameter named errorback (an error callback). Your application would then have this structure:

```
visualize({
    auth : { ...
    }
}, function(){

    // your application logic

}, function(err){

    // handle all initialization and authentication errors here

})
```

Catching Search Errors

One way to handle search errors is to specify an error handler as the second parameter of run:

```
new ResourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
    recursive: false
})).run( usefulFunction, function(error){

    alert(error);

}))
```

Another way to handle search errors is to specify a function as the third parameter of run. This function is an always handler that runs every time when operation ends.

```
new ResourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
```

```
    recursive: false
})).run(usefulFunction, errorHandler, function(resultOrError){

    alert(resultOrError);

}))
```

Validating Search Properties

You can also validate the structure of the search properties without making an actual call to the search function:

```
var call = new ResourcesSearch({
    server:"http://localhost:8080/jasperserver-pro",
    folderUri: "/public",
    recursive: false
}));

var error = call.validate();

if (!error){
    // valid
} else {
    // invalid, read details from error
}
```

Catching Report Errors

To catch and handle errors when running reports, define the contents of the err function as shown in the following sample:

```
visualize({
    auth : { ...
    }
}, function(v){

    var report = v.report({
        error: function(err){
            // invoked once report is initialized and has run
        }
    });

    report
        .run()
        .fail(function(err){
            // handle errors here
        });

)
```

Catching Input Control Errors

Catching and handling input control errors is very similar to handling report errors. Define the contents of the err function that gets invoked in error conditions, as shown in the following sample:

```
visualize({
    auth : { ...
    }
}, function(v){

    var ic = v.inputControls({
        error: function(err){
            // invoked once input control is initialized
        }
    });

    inputControls
        .run()
        .fail(function(err){
            // handle errors here
        });

)
```

Validating Input Controls

You can also validate the structure of your input controls without making an actual call. However, the values of the input controls and their relevance to the named resource are not checked.

```
var ic = new InputControls({
    server: "http://localhost:8080/jasperserver-pro",
    resource: "/public/my_report",
    params: {
        "Country_multi_select":["Mexico"],
        "Cascading_state_multi_select":["Guerrero", "Sinaloa"]
    }
});

var error = ic.validate();

if (!error){
    // valid
} else {
    // invalid, read details from error
}
```

API Usage - Report Events

Depending on the size of your data, the report function can run for several seconds or minutes, just like reports in the JasperReports Server UI. You can listen for events that give the status of running reports and display pages sooner.

This chapter contains the following sections:

- Tracking Completion Status
- Tracking Report Container Size
- Listening for Page Totals
- Listening for the Last Page
- Customizing a Report's DOM Before Rendering

Tracking Completion Status

By listening for the reportCompleted event, you can give information or take action when a report finishes rendering.

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.report({
        // run example with a very long report
        resource: "/public/Samples/Reports/RevenueDetailReport",
        container: "#container",
        events: {
            reportCompleted: function(status) {
                alert("Report status: "+ status+ "!");
            }
        },
        error: function(error) {
            alert(error);
        },
    });
});
```

Tracking Report Container Size

By listening for the responsiveBreakpointChanged event, you can track when the container size has been passed from one interval to another.

```
visualize({
    auth: {...
    }
}, function(v) {
     let reportConfig = {
         resource: "/public/ResponsiveReport",
         container: "#container",
         reportContainerWidth: getContainerWidth(),
         events: {
               responsiveBreakpointChanged: function(error) {
                    if (error) {
                        console.log(error);
                     } else {
                         report.destroy();
                      // rerun report
                         reportConfig.reportContainerWidth = getContainerWidth();
                         report = v.report(reportConfig);
                            }
                      }
                    },
                    error: (e) => console.error(e.message || e)
            };
            let report = v.report(reportConfig);

            function getContainerWidth() {
                  return document.getElementById("container").clientWidth;
           }
 });
```

Listening for Page Totals

By listening for the changeTotalPages event, you can track the filling of the report.

```
visualize({
    auth: { ...
    }
}, function (v) {
       var report = v.report({
           resource: "/public/Samples/Reports/AllAccounts",
           container: "#container",
           error: function(error) {
               alert(error);
               },
           events: {
               changeTotalPages: function(totalPages) {
                   alert("Total Pages:" + totalPages);
               }
           }
       });
});
```

Listening for the Last Page

Listening for the pageFinal event, lets you know when the last page of a running report has been generated.

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.report({
        // run example with a very long report
        resource: "/public/Samples/Reports/RevenueDetailReport",
        container: "#container",
        events: {
            pageFinal: function(el) {
                console.log(el);
                alert("Final page is rendered!");
            },
            reportCompleted: function(status) {
                alert("Report status: "+ status+ "!");
            }
        },
        error: function(error) {
            alert(error);
        },
    });
});
```

Customizing a Report's DOM Before Rendering

By listening for the beforeRender event, you can access the Document Object Model (DOM) of the report to view or modify it before it is displayed. In the example the listener finds span elements and adds a color style and an attribute my-attr="test" to each one.

```
visualize({
    auth: { ...
    }
}, function (v) {
    // enable report chooser
    $(':disabled').prop('disabled', false);

    //render report from provided resource
    startReport();

    $("#selected_resource").change(startReport);

    function startReport () {
        // clean container
        $("#container").html("");
        // render report from another resource
        v("#container").report({
```

```
            resource: $("#selected_resource").val(),
            events:{
                beforeRender: function(el){
                    // find all spans
                    $(el).find(".jrPage td span")
                         .each(function(i, e){
                               // make them red
                               $(e).css("color","red")
                                   .attr("data-my-attr", "test");
                          });
                    console.log($(el).find(".jrPage").html());
                }
            }
        });
    };
});
```

The HTML page that displays the report uses a static list of reports in a drop-down selector, but otherwise needs only a container element. This is similar to the basic report example in Rendering a Report, except that the JavaScript above will change the report before it's displayed.

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<select id="selected_resource" disabled="true" name="report">
    <option value="/public/Samples/Reports/1._Geographic_Results_by_Segment_Report">Geographic
Results by Segment</option>
    <option value="/public/Samples/Reports/2_Sales_Mix_by_Demographic_Report">Sales Mix by
Demographic</option>
    <option value="/public/Samples/Reports/3_Store_Segment_Performance_Report">Store Segment
Performance</option>
    <option value="/public/Samples/Reports/04._Product_Results_by_Store_Type_Report">Product
Results by Store Type</option>
</select>
<!-- Provide a container to render your visualization -->
<div id="container"></div>
```

API Usage - Hyperlinks

Both reports and dashboards include hyperlinks (URLs) that link to websites or other reports. Visualize.js gives you access to the links so that you can customize them or open them differently. For links generated in the report, you can customize both the appearance and the container where they are displayed.

This chapter contains the following sections:

- Structure of Hyperlinks
- Customizing Links

- Drill-Down in Separate Containers
- Accessing Data in Links

Structure of Hyperlinks

The following JSON schema describes all the parameters on links, although not all are present in all cases.

```
"jrLink": {
    "title": "JR Hyperlink",
    "description": "A JSON Schema describing JR hyperlink",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties":{
        "id": {
            "type": "string",
            "description": "Hyperlink id, reflected in corresponding attribute in DOM. Is not used
for AdHocExecution hyperlink type."
        },
        "type": {
            "type": "string",
            "description": "Hyperlink type. Default types are LocalPage, LocalAnchor, RemotePage,
RemoteAnchor, Reference, ReportExecution, AdHocExecution. Custom hyperlink types are possible"
        },
        "target": {
            "type": "string",
            "description": "Hyperlink target. Default targets are Self, Blank, Top, Parent. Custom
hyperlink targets are possible"
        },
        "tooltip": {
            "type": "string",
            "description": "Hyperlink tooltip"
        },
        "href": {
            "type" : "string",
            "description": "Hyperlink reference. Is an empty string for LocalPage, LocalAnchor and
ReportExecution hyperlink types"
        },
        "parameters": {
            "type": "object",
            "description": "Hyperlink parameters. Any additional parameters for hyperlink"
        },
        "resource": {
            "type": "string",
            "description": "Repository resource URI of resource mentioned in hyperlink. For
LocalPage and LocalAnchor points to current report, for ReportExecution - to _report parameter"
        },
        "pages": {
            "type": ["integer", "string"],
            "description": "Page to which hyperlink points to. Is actual for LocalPage, RemotePage
and ReportExecution hyperlink types"
        },
        "anchor": {
            "type": "string",
            "description": "Anchor to which hyperlink points to. Is actual for LocalAnchor,
RemoteAnchor and ReportExecution hyperlink types"
        }
    },
```

```
    "required": ["type", "id"]
}
```

Customizing Links

You can customize the appearance of link elements in a generated report in two ways:

- The linkOptions exposes the beforeRender event to which you can add a listener with access to the links in the document as element pairs.
- The normal click event lets you add a listener that can access to a link when it's clicked.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
},function (v) {
    v("#container1").report({
        resource: "/AdditionalResourcesForTesting/Drill_Reports_with_Controls/main_report",
        linkOptions: {
            beforeRender: function (linkToElemPairs) {
                linkToElemPairs.forEach(function (pair) {
                    var el = pair.element;
                    el.style.backgroundColor = "red";
                });
            },
            events: {
                "click": function(ev, link){
                    if (confirm("Change color of link id " + link.id + " to green?")){
                        ev.currentTarget.style.backgroundColor = "green";
                        ev.target.style.color = "#FF0";
                    }
                }
            }
        },
        error: function (err) {
            alert(err.message);
        }
    });
});
```

Drill-Down in Separate Containers

By using the method of listing for clicks on hyperlinks, you can write a visualize.js script that sets the destination of drill-down report links to another container. This way, you can create display layouts or overlays for viewing drill-down links embedded in your reports. This sample code also changes the cursor for the embedded links, so they are more visible to users.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    v("#main").report({
        resource: "/MyReports/Drill_Reports_with_Controls/main_report",
        linkOptions: {
            beforeRender: function (linkToElemPairs) {
                linkToElemPairs.forEach(showCursor);
            },
            events: {
                "click": function(ev, link){
                   if (link.type == "ReportExecution"){
                        v("#drill-down").report({
                            resource: link.parameters._report,
                            params: {
                                city: [link.parameters.city],
                                country: link.parameters.country,
                                state: link.parameters.state
                            },
                        });
                    }
                     console.log(link);
                }
            }
        },
        error: function (err) {
            alert(err.message);
        }
    });

    function showCursor(pair){
        var el = pair.element;
        if (typeof(el) != "undefined") {
            el.style.cursor = "pointer";
        }
    }

});
```

Associated HTML:

```
<script src="http://underscorejs.org/underscore.js"></script>
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<!-- Provide a container for the main report and one for the drill-down -->
<div>
    <div style="width:830px;" id="main"></div>
    <div style="width:500px;" id="drill-down"></div>
</div>
```

Associated CSS:

```
#main{
   float: left;
}

#drill-down{
    float: left;
}
```

Accessing Data in Links

In this example, we access the hyperlinks through the data.links structure after the report has successfully rendered. From this structure, we can read the tooltips that were set in the JRXML of the report. The script uses the information in the tooltips of all links in the report to create a drop-down selector of city name options.

By using link tooltips, your JRXML can create reports that pass runtime information to the display logic in your JavaScripts.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    var $select = $("#selectCity"),
        report = v.report({
            resource: "/MyReports/Drill_Reports_with_Controls/main_report",
            container: "#main",
            success: refreshSelect,
            error: showError
    });
```

```
    function refreshSelect(data){
        console.log(data);
        var options = data.links.reduce(function(memo, link){
                console.log(link);
                return memo + ""+link.tooltip+"";
            },"");
        $select.html(options);
    }

    $("#previousPage").click(function() {
        var currentPage = report.pages() || 1;
        goToPage(--currentPage);
    });

    $("#nextPage").click(function() {
        var currentPage = report.pages() || 1;
        goToPage(++currentPage);
    });

    function goToPage(numder){
        report
            .pages(numder)
            .run()
                .done(refreshSelect)
                .fail(showError);
    }

    function showError(err){
        alert(err.message);
    }

});
```

By using link tooltips, your JRXML can create reports that pass runtime information to the display logic in your JavaScripts.

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<select id="selectCity"></select>
<button id="previousPage">Previous Page</button>
<button id="nextPage">Next Page</button>
<!-- Provide a container for the main report -->
<div>
    <div style="width:20px;"></div>
    <div style="width:500px;" id="main"></div>
</div>
```

Associated CSS:

```
#main{
   float: left;
}
```

API Usage - Interactive Reports

Most reports rendered in the JasperReports Server native interface have interactive abilities such as column sorting provided by a feature called JIVE: Jaspersoft Interactive Viewer and Editor. The JIVE UI is the interface of the report viewer in JasperReports Server, and the same JIVE UI is replicated on reports generated in clients using Visualize.js.

Not only does the JIVE UI allow users to sort and filter regular reports, it also provides many opportunities for you to further customize the appearance and behavior of your reports through Visualize.js .

This chapter contains the following sections:

- Interacting With JIVE UI Components
- Using Floating Headers
- Changing the Chart Type
- Changing the Chart Properties
- Undo and Redo Actions
- Sorting Table Columns
- Filtering Table Columns
- Formatting Table Columns
- Conditional Formatting on Table Columns
- Sorting Crosstab Columns
- Sorting Crosstab Rows
- Implementing Search in Reports
- Providing Bookmarks in Reports
- Disabling the JIVE UI

Interacting With JIVE UI Components

The visualize.report interface exposes the updateComponent function that gives your script access to the JIVE UI. Using the updateComponent function, you can programmatically interact with the JIVE UI to do such things as set the sort order on a specified column, add a filter, and change the chart type. In addition, the undoAll function acts as a reset.

For the API reference of the visualize.report interface, see Report Functions.

First, your script must enable the default JIVE UI to make its components available after running a report:

```
var report = v.report({
    resource: "/public/SampleReport",
    defaultJiveUi : {
        enabled: true
    }
});
...
var components = report.data().components;
```

The components that can be modified are columns and charts. These components of the JIVE UI have an ID, but it may change from execution to execution. To refer to these components, create your report in JRXML and use the net.sf.jasperreports.components.name property to name them. In the case of a column, this property should be set on the column definition in the table model. In Jaspersoft Studio, you can select the column in the Outline View, then go to Properties > Advanced, and under Misc > Properties you can define custom properties.

Then you can reference the component by this name, for example a column named sales, and use the updateComponent function to modify it.

```
report.updateComponent("sales", {
    sort : {
        order : "asc"
    }
});
```

Or:

```
report.updateComponent({
    name: "sales",
    sort : {
        order : "asc"
    }
});
```

We can also get an object that represents the named component of the JIVE UI:

```
var salesColumn = report
                    .data()
                    .components
                    .filter(function(c){ return c.name === "sales"})
                    .pop();
```

The following example shows how to create buttons whose click events modify the report through the JIVE UI. This example assumes you have a report whose components already have names, in this case, columns named my_accounts and my_dept, and a chart named revenue:

```
visualize({
    auth: { ...
    }
}, function (v) {
```

```
    //render report from provided resource
    var report = v.report({
        resource: "/public/SampleReport",
        container: "#container",
        success: printComponentsNames,
        error: handleError
    });

    $("#resetAll").on("click", function(){
        report.undoAll();
    });

    $("#changeAccounts").on("click", function () {
        report.updateComponent("my_accounts", {
            sort: {
                order: "asc"
            },
            filter: {
                operator: "greater_or_equal",
                value: 15000
            }
        }).fail(handleError);
    });

    $("#changeDept").on("click", function () {
        report.updateComponent("my_dept", {
            sort: {
                order: "desc"
            }
        }).fail(handleError);
    });

    $("#changeChart").on("click", function () {
        report.updateComponent("revenue", {
             chartType: "Pie"
        }).fail(handleError);
    });

    //show error
    function handleError(err){
        alert(err.message);
    }

    function printComponentsNames(data){
        data.components.forEach(function(c){
            console.log("Component Name: " + c.name, "Component Label: "+ c.label);
        });
    }

});
```

The associated HTML has buttons that will invoke the JavaScript actions on the JIVE UI:

```
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<button id="resetAll">Reset All</button>
<button id="changeAccounts">View Top Accounts</button>
<button id="changeDept">Sort Departments</button>
<button id="changeChart">Show Pie Chart</button>
<!-- Provide a container for the report -->
<div id="container"></div>
```

Using Floating Headers

One feature of the JIVE UI for tables and crosstabs is the floating header. When you turn on floating headers, the header rows of a table or crosstab float at the top of the container when you scroll down. The report container must allow scrolling for this to take effect. This means that CSS property overflow with values like scroll or auto must be specifically set for the report container.

To turn on floating headers for your interactive reports, set the following parameters when you enable the JIVE UI:

```
var report = v.report({
    resource: "/public/SampleReport",
    defaultJiveUi : {
        floatingTableHeadersEnabled: true,
        floatingCrosstabHeadersEnabled: true
    }
});
```

Changing the Chart Type

If you have the name of a chart component, you can easily set a new chart type and redraw the chart.

```
var mySalesChart = report
                    .data()
                    .components
                    .filter(function(c){ return c.name === "salesChart"})
                    .pop();

mySalesChart.chartType = "Bar";

report
    .updateComponent(mySalesChart)
    .done(function(){
```

```
        alert("Chart type changed!");
    })
    .fail(function(err){
        alert(err.message);
    });
```

Or:

```
report
    .updateComponent("salesChart", {
          chartType: "Bar"
    })
    .done(function(){
        alert("Chart type changed!");
    })
    .fail(function(err){
        alert(err.message);
    });
```

The following example creates a drop-down menu that lets users change the chart type. You could also set the chart type according to other states in your client.

This code also relies on the report.chart.types interface described in Discovering Available Charts and Formats.

```
visualize({
    auth: { ...
    }
}, function (v) {

    //persisted chart name
    var chartName = "geo_by_seg",
        $select = buildControl("Chart types: ", v.report.chart.types),
        report = v.report({
            resource: "/public/Reports/1._Geographic_Results_by_Segment_Report",
            container: "#container",
            success: selectDefaultChartType
        });

    $select.on("change", function () {
        report.updateComponent(chartName, {
                chartType: $(this).val()
            })
            .done(function (component) {
                chartComponent = component;
            })
            .fail(function (error) {
                alert(error);
            });
    });

    function selectDefaultChartType(data) {
        var component = data.components
                            .filter(function (c) {
                                   return c.name === chartName;
                            })
```

```
                        .pop();
        if (component) {
            $select.find("option[value='" + component.chartType + "']")
                   .attr("selected", "selected");
        }
    }
```

```
    function buildControl(name, options) {

        function buildOptions(options) {
            var template = "<option>{value}</option>";
            return options.reduce(function (memo, option) {
                return memo + template.replace("{value}", option);
            }, "")
        }

        console.log(options);

        if (!options.length) {
            console.log(options);
        }

        var template = "<label>{label}</label><select>{options}</select><br>",
            content = template.replace("{label}", name)
                .replace("{options}", buildOptions(options));

        var $control = $(content);
        $control.insertBefore($("#container"));
        return $control;
    }

});
```

As shown in the following HTML, the control for the chart type is created dynamically by the JavaScript:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<!-- Provide a container for the report -->
<div id="container"></div>
```

Changing the Chart Properties

Those chart components that are based on Highcharts have a lot of interactivity such as built-in zooming and animation. The built-in zooming lets users select data, for example columns in a chart, but it can also interfere with touch interfaces. With Visualize.js, you have full control over these features and you can choose to allow your users access to them or not. For example, animation can be slow on mobile devices, so you could turn off both zooming and animation. Alternatively, if your users have a range of mobile devices,

tablets, and desktop computers, then you could give users the choice of turning on or off these properties themselves.

The following example creates buttons to toggle several chart properties and demonstrates how to control them programmatically. First the HTML to create the buttons:

```
<script src="http://localhost:8080/jasperserver-pro/client/visualize.js"></script>

<button id="disableAnimation">disable animation</button>
<button id="enableAnimation">enable animation</button>
<button id="resetAnimation">reset animation to initial state</button>

<button id="disableZoom">disable zoom</button>
<button id="zoomX">set zoom to 'x' type</button>
<button id="zoomY">set zoom to 'y' type</button>
<button id="zoomXY">set zoom to 'xy' type</button>
<button id="resetZoom">reset zoom to initial state</button>

<div id="container"></div>
```

Here are the API calls to set the various chart properties:

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    var report = v.report({
        resource: "/public/Samples/Reports/01._Geographic_Results_by_Segment_Report",
        container: "#container",
        error: function(e) {
            alert(e);
        }
    });
```

```
    function changeChartProperty(prop, value) {
        var chartProps = report.chart();

        if (typeof value === "undefined") {
            delete chartProps[prop];
        } else {
            chartProps[prop] = value;
        }

        report.chart(chartProps).run().fail(function(e) { alert(e); });
    }

    $("#disableAnimation").on("click", function() {
        changeChartProperty("animation", false);
    });
```

```
    $("#enableAnimation").on("click", function() {
        changeChartProperty("animation", true);
    });

    $("#resetAnimation").on("click", function() {
        changeChartProperty("animation");
    });

    $("#disableZoom").on("click", function() {
        changeChartProperty("zoom", false);
    });

    $("#zoomX").on("click", function() {
        changeChartProperty("zoom", "x");
    });

    $("#zoomY").on("click", function() {
        changeChartProperty("zoom", "y");
    });

    $("#zoomXY").on("click", function() {
        changeChartProperty("zoom", "xy");
    });

    $("#resetZoom").on("click", function() {
        changeChartProperty("zoom");
    });
});
```

Undo and Redo Actions

As in JasperReports Server, the JIVE UI supports undo and redo actions that you can access programmatically with Visualize.js. As in many applications, undo and redo actions act like a stack, and the canUndo and canRedo events notify your page you are at either end of the stack.

```
visualize({
    auth: { ...
    }
}, function (v) {

    var chartComponent,
        report = v.report({
            resource: "/public/Samples/Reports/1._Geographic_Results_by_Segment_Report",
            container: "#container",
            events: {
                canUndo: function (canUndo) {
                    if (canUndo) {
                        $("#undo, #undoAll").removeAttr("disabled");
                    } else {
                        $("#undo, #undoAll").attr("disabled", "disabled");
                    }
                },
                canRedo: function (canRedo) {
                    if (canRedo) {
```

```
                        $("#redo").removeAttr("disabled");
                    } else {
                        $("#redo").attr("disabled", "disabled");
                    }
                }
```

```
            },
            success: function (data) {
                chartComponent = data.components.pop();
                $("option[value='" + chartComponent.chartType + "']").attr("selected",
"selected");
            }
        });

    var chartTypeSelect = buildChartTypeSelect(report);

    chartTypeSelect.on("change", function () {
        report.updateComponent(chartComponent.id, {
            chartType: $(this).val()
        })
            .done(function (component) {
            chartComponent = component;
        })
            .fail(function (error) {
            console.log(error);
            alert(error);
        });
    });

    $("#undo").on("click", function () {
        report.undo().fail(function (err) {
            alert(err);
        });
    });

    $("#redo").on("click", function () {
        report.redo().fail(function (err) {
            alert(err);
        });
    });

    $("#undoAll").on("click", function () {
        report.undoAll().fail(function (err) {
            alert(err);
        });
    });
});
```

```
function buildChartTypeSelect(report) {

    var chartTypes = report.schema("chart").properties.chartType.enum,
        chartTypeSelect = $("#chartType");

    $.each(chartTypes, function (index, type) {
        chartTypeSelect.append("" + type + "");
    });

   return chartTypeSelect;
}
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<select id="chartType"></select>
<button id="undo" disabled="disabled">Undo</button>
<button id="redo" disabled="disabled">Redo</button>
<button id="undoAll" disabled="disabled">Undo All</button>
<!-- Provide a container for the report -->
<div id="container"></div>
```

Sorting Table Columns

This code example shows how to set the three possible sorting orders on a column in the JIVE UI: ascending, descending, and no sorting.

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.report({
        resource: "/public/Samples/Reports/5g.AccountsReport",
        container: "#container",
        error: showError
    });

    $("#sortAsc").on("click", function () {
        report.updateComponent("name", {
            sort: {
                order: "asc"
            }
        })
        .fail(showError);
    });

    $("#sortDesc").on("click", function () {
        report.updateComponent("name", {
            sort: {
                order: "desc"
            }
        })
        .fail(showError);
    });

    $("#sortNone").on("click", function () {
        report.updateComponent("name", {
            sort: {}
        }).fail(showError);
    });

    function showError(err) {
        alert(err);
    }
});
```

Associated HTML:

```
<script src="https://code.jquery.com/jquery-3.3.1.min.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="sortAsc">Sort NAME column ASCENDING</button>
<button id="sortDesc">Sort NAME column DESCENDING</button>
<button id="sortNone">Reset NAME column</button>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Filtering Table Columns

This code example shows how to define filters on columns of various data types (dates, strings, numeric) in the JIVE UI. It also shows several filter operator such as equal, greater, between, contain (for string matching), and before (for times and dates).

```
visualize({
    auth: { ...
    }
}, function (v) {
    var report = v.report({
             resource:"/public/viz/report_with_different_column_types",
            container: "#container",
            error: function(err) {
                alert(err);
            }
        });

    $("#setTimestampRange").on("click", function() {
        report.updateComponent("column_timestamp", {
            filter: {
                operator: "between",
                value: [$("#betweenDates1").val(), $("#betweenDates2").val()]
            }
        }).fail(handleError);
    });

    $("#resetTimestampFilter").on("click", function() {
        report.updateComponent("column_timestamp", {
            filter: {}
        }).fail(handleError);
    });

    $("#setBooleanTrue").on("click", function() {
        report.updateComponent("column_boolean", {
```

```
            filter: {
                operator: "equal",
                value: true
            }
        }).fail(handleError);
    });

    $("#resetBoolean").on("click", function() {
        report.updateComponent("column_boolean", {
            filter: {}
        }).fail(handleError);
    });

    $("#setStringContains").on("click", function() {
        report.updateComponent("column_string", {
            filter: {
                operator: "contain",
                value: $("#stringContains").val()
            }
        }).fail(handleError);
    });

    $("#resetString").on("click", function() {
        report.updateComponent("column_string", {
            filter: {}
        }).fail(handleError);
    });

    $("#setNumericGreater").on("click", function() {
        report.updateComponent("column_double", {
            filter: {
                operator: "greater",
                value: parseFloat($("#numericGreater").val(), 10)
            }
        }).fail(handleError);
    });

    $("#resetNumeric").on("click", function() {
        report.updateComponent("column_double", {
            filter: {}
        }).fail(handleError);
    });

    $("#setTimeBefore").on("click", function() {
        report.updateComponent("column_time", {
            filter: {
                operator: "before",
                value: $("#timeBefore").val()
            }
        }).fail(handleError);
    });

    $("#resetTime").on("click", function() {
        report.updateComponent("column_time", {
            filter: {}
        }).fail(handleError);
    });
});

function handleError(err) {
     console.log(err);
     alert(err);
}
```

Associated HTML:

```
<script src="http://code.jquery.com/jquery-2.1.0.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>


<input type="text" value="2014-04-10T00:00:00" id="betweenDates1"/> -
<input type="text" id="betweenDates2" value="2014-04-24T00:00:00"/>
<button id="setTimestampRange">Set timestamp range</button>
<button id="resetTimestampFilter">Reset timestamp filter</button>
<br/><br/>
<button id="setBooleanTrue">Filter boolean column to true</button>
<button id="resetBoolean">Reset boolean filter</button>
<br/><br/>
<input type="text" value="hou" id="stringContains"/>
<button id="setStringContains">Set string column contains</button>
<button id="resetString">Reset string filter</button>
<br/><br/>
<input type="text" value="40.99" id="numericGreater"/>
<button id="setNumericGreater">Set numeric column greater than</button>
<button id="resetNumeric">Reset numeric filter</button>
<br/><br/>
<input type="text" value="13:15:43" id="timeBefore"/>
<button id="setTimeBefore">Set time column before than</button>
<button id="resetTime">Reset time filter</button>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Formatting Table Columns

The JIVE UI allows you to format columns by setting the alignment, color, font, size, and background of text in both headings and cells. You can also set the numeric format of cells, such as the precision, negative indicator, and currency. Note that the initial appearance of any numbers also depends on the locale set either by default on JasperReports Server, or specified in your script request, as described in Requesting the Visualize.js Script.

```
visualize({
    auth: { ...
    }
}, function (v) {
    var columns,
    report = v.report({
        resource: "/public/viz/report_with_different_column_types",
        container: "#container",
        events: {
```

```
                reportCompleted: function (status, error) {
                    if (status === "ready") {
                        columns = _.filter(report.data().components, function (component) {
                            return component.componentType == "tableColumn";
                        });

                        var column4 = columns[4];

                        $("#label").val(column4.label);

                        $("#headingFormatAlign").val(column4.headingFormat.align);
                        $("#headingFormatBgColor").val(column4.headingFormat.backgroundColor);
                        $("#headingFormatFontSize").val(column4.headingFormat.font.size);
                        $("#headingFormatFontColor").val(column4.headingFormat.font.color);
                        $("#headingFormatFontName").val(column4.headingFormat.font.name);
                        if (column4.headingFormat.font.bold) {
                            $("#headingFormatFontBold").attr("checked", "checked");
                        } else {
                            $("#headingFormatFontBold").removeAttr("checked");
                        }
                        if (column4.headingFormat.font.italic) {
                            $("#headingFormatFontItalic").attr("checked", "checked");
                        } else {
                            $("#headingFormatFontItalic").removeAttr("checked");
                        }
                        if (column4.headingFormat.font.underline) {
                            $("#headingFormatFontUnderline").attr("checked", "checked");
                        } else {
                            $("#headingFormatFontUnderline").removeAttr("checked");
                        }

                        $("#detailsRowFormatAlign").val(column4.detailsRowFormat.align);
                        $("#detailsRowFormatBgColor").val(column4.detailsRowFormat.backgroundColor);
                        $("#detailsRowFormatFontSize").val(column4.detailsRowFormat.font.size);
                        $("#detailsRowFormatFontColor").val(column4.detailsRowFormat.font.color);
                        $("#detailsRowFormatFontName").val(column4.detailsRowFormat.font.name);
                        if (column4.detailsRowFormat.font.bold) {
                            $("#detailsRowFormatFontBold").attr("checked", "checked");
                        } else {
                            $("#detailsRowFormatFontBold").removeAttr("checked");
                        }
                        if (column4.detailsRowFormat.font.italic) {
                            $("#detailsRowFormatFontItalic").attr("checked", "checked");
                        } else {
                            $("#detailsRowFormatFontItalic").removeAttr("checked");
                        }
                        if (column4.detailsRowFormat.font.underline) {
                            $("#detailsRowFormatFontUnderline").attr("checked", "checked");
                        } else {
                            $("#detailsRowFormatFontUnderline").removeAttr("checked");
                        }

                        $("#detailsRowFormatPatternNegativeFormat").val(
                            column4.detailsRowFormat.pattern.negativeFormat);
                        $("#detailsRowFormatPatternPrecision").val(
                            column4.detailsRowFormat.pattern.precision);
                        $("#detailsRowFormatPatternCurrency").val(
                            column4.detailsRowFormat.pattern.currency || "");

                        if (column4.detailsRowFormat.pattern.percentage) {
```

```
                    $("#detailsRowFormatPatternPercentage").attr("checked", "checked");
                } else {
                    $("#detailsRowFormatPatternPercentage").removeAttr("checked");
                }

                if (column4.detailsRowFormat.pattern.grouping) {
                    $("#detailsRowFormatPatternGrouping").attr("checked", "checked");
                } else {
                    $("#detailsRowFormatPatternGrouping").removeAttr("checked");
                }
            }
        }
    },
    error: function (err) {
        alert(err);
    }
});

$("#changeHeadingFormat").on("click", function () {
    report.updateComponent(columns[4].id, {
        headingFormat: {
            align: $("#headingFormatAlign").val(),
            backgroundColor: $("#headingFormatBgColor").val(),
            font: {
                size: parseFloat($("#headingFormatFontSize").val()),
                color: $("#headingFormatFontColor").val(),
                underline: $("#headingFormatFontUnderline").is(":checked"),
                bold: $("#headingFormatFontBold").is(":checked"),
                italic: $("#headingFormatFontItalic").is(":checked"),
                name: $("#headingFormatFontName").val()
            }
        }
    }).fail(function (e) {
        alert(e);
    });
});

$("#changeDetailsRowFormat").on("click", function () {
    report.updateComponent(columns[4].id, {
        detailsRowFormat: {
            align: $("#detailsRowFormatAlign").val(),
            backgroundColor: $("#detailsRowFormatBgColor").val(),
            font: {
                size: parseFloat($("#detailsRowFormatFontSize").val()),
                color: $("#detailsRowFormatFontColor").val(),
                underline: $("#detailsRowFormatFontUnderline").is(":checked"),
                bold: $("#detailsRowFormatFontBold").is(":checked"),
                italic: $("#detailsRowFormatFontItalic").is(":checked"),
                name: $("#detailsRowFormatFontName").val()
            },
            pattern: {
                negativeFormat: $("#detailsRowFormatPatternNegativeFormat").val(),
                currency: $("#detailsRowFormatPatternCurrency").val() || null,
                precision: parseInt($("#detailsRowFormatPatternPrecision").val(), 10),
                percentage: $("#detailsRowFormatPatternPercentage").is(":checked"),
                grouping: $("#detailsRowFormatPatternGrouping").is(":checked")
            }
        }
    }).fail(function (e) {
        alert(e);
    });
});

$("#changeLabel").on("click", function () {
    report.updateComponent(columns[4].id, {
```

```
            label: $("#label").val()
        }).fail(function (e) {
            alert(e);
        });
    });
});
```

The associated HTML has static controls for selecting all the formatting options that the script above can modify in the report.

```
<script src="http://code.jquery.com/jquery-2.1.0.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<div style="float: left;">
    <h3>Heading format for 5th column</h3>
    Align:  <select id="headingFormatAlign">
                <option value="left">left</option>
                <option value="center">center</option>
                <option value="right">right</option></select>
    <br/>
    Background color: <input type="text" id="headingFormatBgColor" value=""/>
    <br/>
    Font size: <input type="text" id="headingFormatFontSize" value=""/>
    <br/>
    Font color: <input type="text" id="headingFormatFontColor" value=""/>
    <br/>
    Font name: <input type="text" id="headingFormatFontName" value=""/>
    <br/>
    Bold: <input type="checkbox" id="headingFormatFontBold" value="true"/>
    <br/>
    Italic: <input type="checkbox" id="headingFormatFontItalic" value="true"/>
    <br/>
    Underline: <input type="checkbox" id="headingFormatFontUnderline" value="true"/>
    <br/><br/>
    <button id="changeHeadingFormat">Change heading format</button>
</div>
<div style="float: left;">
    <h3>Details row format for 5th column</h3>
    Align: <select id="detailsRowFormatAlign">
        <option value="left">left</option>
        <option value="center">center</option>
        <option value="right">right</option></select>
    <br/>
    Background color: <input type="text" id="detailsRowFormatBgColor" value=""/>
    <br/>
    Font size: <input type="text" id="detailsRowFormatFontSize" value=""/>
    <br/>
    Font color: <input type="text" id="detailsRowFormatFontColor" value=""/>
    <br/>
    Font name: <input type="text" id="detailsRowFormatFontName" value=""/>
    <br/>
    Bold: <input type="checkbox" id="detailsRowFormatFontBold" value="true"/>
    <br/>
    Italic: <input type="checkbox" id="detailsRowFormatFontItalic" value="true"/>
    <br/>
    Underline: <input type="checkbox" id="detailsRowFormatFontUnderline" value="true"/>
    <br/><br/>
```

```
    <b>Number pattern:</b>
    <br/>
    Negative format: <input type="text" id="detailsRowFormatPatternNegativeFormat"/>
    <br/>
    Precision: <input type="text" id="detailsRowFormatPatternPrecision"/>
    <br/>
    Currency: <select id="detailsRowFormatPatternCurrency">
                    <option value="">----</option>
                    <option value="USD">USD</option>
                    <option value="EUR">EUR</option>
                    <option value="GBP">GBP</option>
                    <option value="YEN">YEN</option>
                    <option value="LOCALE_SPECIFIC">LOCALE_SPECIFIC</option>
            </select>
    <br/>
    Thousands grouping: <input type="checkbox" id="detailsRowFormatPatternGrouping" value="true"/>
    <br/>
    Percentage: <input type="checkbox" id="detailsRowFormatPatternPercentage" value="true"/>
    <br/><br/>
    <button id="changeDetailsRowFormat">Change details row format</button>
</div>
<div style="float: left;">
    <h3>Change label of 5th column</h3>
    <br/>
    Label <input type="text" id="label"/>
    <br/>
    <button id="changeLabel">Change label</button>
</div>
<div style="clear: both;"></div>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Conditional Formatting on Table Columns

The JIVE UI also supports conditional formatting so that you can change the appearance of a cell's contents based on its value. This example highlights cells in a given column that have a certain value by changing their text color and the cell background color. Note that the column name must be known ahead of time, for example by looking at your JRXML.

```
visualize({
    auth: { ...
    }
}, function (v) {
    // column name from JRXML (field name by default)
    var salesColumnName = "sales_fact_ALL.sales_fact_ALL__store_sales_2013",
        report = v.report({
            resource: "/public/Samples/Reports/04._Product_Results_by_Store_Type_Report",
            container: "#container",
            error: showError
        });

    $("#changeConditions").on("click", function() {
```

```
            report.updateComponent(salesColumnName, {
                conditions: [
                    {
                        operator: "greater",
                        value: 10,
                        backgroundColor: null,
                        font: {
                            color: "FF0000",
                            bold: true,
                            underline: true,
                            italic: true
                        }
                    },
                    {
                        operator: "between",
                        value: [5, 9],
                        backgroundColor: "00FF00",
                        font: {
                            color: "0000FF"
                        }
                    }
                ]
            })
            .then(printConditions)
            .fail(showError);
    });

    function printConditions(component){
        console.log("Conditions: "+ component.conditions);
    }

    function showError(err) {
           alert(err);
    }
});
```

This example has a single button that allows the user to apply the conditional formatting when the report is loaded:

```
<script src="http://code.jquery.com/jquery-2.1.0.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="changeConditions">Change conditions for numeric column</button>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Sorting Crosstab Columns

Crosstabs are more complex and do not have as many formatting options. This example shows how to sort the values in a given column of a crosstab (the rows are rearranged). Note that the code is slightly different than Sorting Table Columns.

```
visualize({
    auth: {
        name: "superuser",
        password: "superuser"
    }
}, function (v) {
    var column2,
    report = v.report({
        resource: "/public/MyReports/crosstabReport",
        container: "#container",
        events: {
            reportCompleted: function (status, error) {
                if (status === "ready") {
                    var columns = _.filter(report.data().components, function (component) {
                        return component.componentType == "crosstabDataColumn";
                    });

                    column2 = columns[1];
                    console.log(columns);
                }
            }
        },
        error: function (err) {
            alert(err);
        }
    });

    $("#sortAsc").on("click", function () {
        report.updateComponent(column2.id, {
            sort: {
                order: "asc"
            }
        }).fail(function (e) {
            alert(e);
        });
    });

    $("#sortDesc").on("click", function () {
        report.updateComponent(column2.id, {
            sort: {
                order: "desc"
            }
        }).fail(function (e) {
            alert(e);
        });
    });

    $("#sortNone").on("click", function () {
        report.updateComponent(column2.id, {
            sort: {}
        }).fail(function (e) {
            alert(e);
        });
    });
});
```

The associated HTML has the buttons to trigger the sorting:

```
<script src="http://code.jquery.com/jquery-2.1.0.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
```

```
<button id="sortAsc">Sort 2nd column ascending</button>
<button id="sortDesc">Sort 2nd column descending</button>
<button id="sortNone">Do not sort on 2nd column</button>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Sorting Crosstab Rows

This example shows how to sort the values in a given row of a crosstab (the columns are rearranged).

```
visualize({
    auth: { ...
    }
}, function (v) {
    var row,
    report = v.report({
        resource: "/public/MyReports/crosstabReport",
        container: "#container",
        events: {
            reportCompleted: function (status, error) {
                if (status === "ready") {
                    row = _.filter(report.data().components, function (component) {
                        return component.componentType == "crosstabRowGroup";
                    })[0];
                }
            }
        },
        error: function (err) {
            alert(err);
        }
    });

    $("#sortAsc").on("click", function () {
        report.updateComponent(row.id, {
            sort: {
                order: "asc"
            }
        }).fail(function (e) {
            alert(e);
        });
    });

    $("#sortDesc").on("click", function () {
        report.updateComponent(row.id, {
            sort: {
                order: "desc"
            }
        }).fail(function (e) {
            alert(e);
        });
    });

    $("#sortNone").on("click", function () {
        report.updateComponent(row.id, {
```

```
            sort: {}
        }).fail(function (e) {
            alert(e);
        });
    });
});
```

The associated HTML has the buttons to trigger the sorting:

```
<script src="http://code.jquery.com/jquery-2.1.0.js"></script>
<script src="http://underscorejs.org/underscore-min.js"></script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>

<button id="sortAsc">Sort rows ascending</button>
<button id="sortDesc">Sort rows descending</button>
<button id="sortNone">Do not sort rows</button>

<!-- Provide a container for the report -->
<div id="container"></div>
```

Implementing Search in Reports

The JIVE UI supports a search capability within the report. The following example relies on a page with a simple search input.

```
<input id="search-query" type="input" />
<button id="search-button">Search</button>
<!--Provide container to render your visualization-->
<div id="container"></div>
```

Then you can use the search function to return a list of matches in the report. In this example, the search button triggers the function and passes the search term. It uses the console to display the results, but you can use them to locate the search term in a paginated report.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "organization_1"
    }
}, function (v) {

    //render report from provided resource
```

```
    var report = v.report({
        resource: "/public/Samples/Reports/AllAccounts",
        error: handleError,
        container: "#container"
    });

    $("#search-button").click(function(){
        report
        .search($("#search-query").val())
            .done(function(results){
                !results.length && console.log("The search did not return any results!");
                for (var i = 0; i < results.length; i++) {
                    console.log("found " + results[i].hitCount + " results on page: #" +
                                results[i].page);
                }
            })
            .fail(handleError);
    });


    //show error
    function handleError(err){
        alert(err.message);
    }

});
```

The search function supports several arguments to refine the search:

```
    $("#search-button").click(function(){
        report
        .search({
            text: $("#search-query").val(),
            caseSensitive: true,
            wholeWordsOnly: true
        })
           ...
```

Providing Bookmarks in Reports

The JIVE UI also supports bookmarks that are embedded within the report. You must create your report with bookmarks, but then Visualize.js can make them available on your page. The following example has a container for the bookmarks and one for the report:

```
<div>
    <h4>Bookmarks</h4>
    <div id="bookmarksContainer"></div>
</div>
<!--Provide container to render your visualization-->
<div id="container"></div>
```

Then you need a function to read the bookmarks in the report and place them in the container. A handler then responds to clicks on the bookmarks.

```
visualize({
    auth: {
        name: "jasperadmin",
        password: "jasperadmin",
        organization: "Organization_1"
    }
}, function (v) {

    //render report from provided resource
    var report = v.report({
        // resource: "/public/Samples/Reports/AllAccounts",
        resource: "/reports/interactive/TableReport",
        error: handleError,
        container: "#container",
        events: {
            bookmarksReady: handleBookmarks
        }
    });

    //show error
    function handleError(err){
        alert(err.message);
    }

    $("#bookmarksContainer").on("click", ".jr_bookmark", function(evt) {
        report.pages({
            anchor: $(this).data("anchor")
        }).run();
    });

    // handle bookmarks
    function handleBookmarks(bookmarks, container) {
        var li, ul = $("<ul/>");
        !container && $("#bookmarksContainer").empty();
        container = container || $("#bookmarksContainer");

        $.each(bookmarks, function(i, bookmark) {
            li = $("<li><span class='jr_bookmark' title='Anchor: " + bookmark.anchor + ", page: "
+ bookmark.page + "' data-anchor='" + bookmark.anchor + "' data-page='" + bookmark.page + "'>" +
bookmark.anchor + "</span></li>");
            bookmark.bookmarks && handleBookmarks(bookmark.bookmarks, li);
            ul.append(li);
        });

        container.append(ul);
    }
});
```

Disabling the JIVE UI

The JIVE UI is enabled by default on all reports that support it. When the JIVE UI is disabled, the report is static and neither users nor your script can interact with the report elements. You can disable it in your visualize.report call as shown in the following example:

```
visualize({
    auth: { ...
    }
}, function (v) {
    v.report({
        resource: "/public/Samples/Reports/RevenueDetailReport",
        container: "#reportContainer",
        defaultJiveUi: { enabled: false },
        error: function (err) {
            alert(err.message);
        }
    });
});
```

Associated HTML:

```
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<p>JIVE UI is disabled on this Visualize.js report:</p>
<div id="reportContainer">Loading...</div>
```

Visualize.js Tools

This chapter provides two extended code examples that you can use to test various parts of your own Visualize.js scripts.

This chapter contains the following sections:

- Checking the Scope in Visualize.js
- CSS Diagnostic Tool

Checking the Scope in Visualize.js

This example reads and displays the properties in the scope after visualize.report finishes rendering a report (success).

```
visualize({
    auth: { ...
    }
}, function (v) {

    createReport();
```

```
    $("#selected_resource").change(function () {
        //clean container
        $("#container").html("");
        createReport();
    });

     //enable report chooser
    $(':disabled').prop('disabled', false);

     function createReport(){
        //render report from another resource
        v("#container").report({
            resource: $("#selected_resource").val(),
            success: function(){
                 setTimeout(function () {
                    console.log("------------------Scope Check Results--------------------------
");
                    console.log(scopeChecker.compareProperties(propertiesNames));
                    console.log("----------------------------------------------------------------
");
                }, 5000);
            },
            error:handleError
```

```
        });
    }
    //show error
    function handleError(err){
        alert(err.message);
    }
});
```

The ScopeChecker is another JavaScript used in this example. It can either be a separate .js file or included in your HTML file as shown in this example:

```
<!-- JavaScript for ScopeChecker -->
<script>
    function ScopeChecker(scope) {
        this.scope = scope;
    }

    ScopeChecker.prototype.getPropertiesCount = function() {
        return this.getPropertiesNames().length;
    };

    ScopeChecker.prototype.getPropertiesNames = function() {
        return Object.keys(this.scope);
    };

    ScopeChecker.prototype.compareProperties = function(scope1PropertiesNames,
scope2PropertiesNames) {
        if (!scope1PropertiesNames) {
            throw "Properties for scope 1 not specified";
        }
        if (!scope2PropertiesNames) {
            scope2PropertiesNames = this.getPropertiesNames();
        }
```

```
        var comparisonResult = {
            added: [],
            removed: [],
            madeUndefined: [],
            pollution: []
        };

        var i, j;
        for (i = 0; i < scope1PropertiesNames.length; i++) {
            comparisonResult.removed.push(scope1PropertiesNames[i]);
            for (j = 0; j < scope2PropertiesNames.length; j++) {
                if (scope1PropertiesNames[i] === scope2PropertiesNames[j]) {
                    comparisonResult.removed.pop();
                    break;
        }   }   }

        for (i = 0; i < scope2PropertiesNames.length; i++) {
            comparisonResult.added.push(scope2PropertiesNames[i]);
            for (j = 0; j < scope1PropertiesNames.length; j++) {
                if (scope2PropertiesNames[i] === scope1PropertiesNames[j]) {
                    comparisonResult.added.pop();

                    break;
        }   }   }
```

```
        for (i = 0; i < comparisonResult.added.length; i++) {
            if (this.scope[comparisonResult.added[i]] === undefined) {
                comparisonResult.madeUndefined.push(comparisonResult.added[i]);
            } else {
                comparisonResult.pollution.push(comparisonResult.added[i]);
        }   }

        return comparisonResult;
    };


    var propertiesNames = [];
    var scopeChecker = new ScopeChecker(window);
    propertiesNames = scopeChecker.getPropertiesNames();
</script>
<!-- Provide the URL to visualize.js -->
<script src="http://bi.example.com:8080/jasperserver-pro/client/visualize.js"></script>
<select id="selected_resource" name="report">
    <option value="/public/Samples/Reports/1._Geographic_Results_by_Segment_Report"
        >Geographic Results by Segment</option>
    <option value="/public/Samples/Reports/2_Sales_Mix_by_Demographic_Report"
        >Sales Mix by Demographic</option>
    <option value="/public/Samples/Reports/3_Store_Segment_Performance_Report"
        >Store Segment Performance</option>
    <option value="/public/Samples/Reports/04._Product_Results_by_Store_Type_Report"
        >Product Results by Store Type</option>
</select>
<!-- Provide a container to render your visualization -->
<div id="container"></div>
```

CSS Diagnostic Tool

The CSS diagnostic tool lets you load various CSS libraries and see how they interact or interfere with the CSS that Visualize.js uses to render reports. It lets you choose your

JasperReports Server from a static list, so that you may try different themes on different servers. After you load a report using visualize.report, you can choose to load a variety of popular CSS libraries and see if they affect your report. The list of reports to choose from is also a static list, as shown in the HTML code below.

The key feature of this tool is the ability to set the isolateDOM property on the visualize.report function call. This property modifies the CSS of the report so it doesn't conflict with other CSS libraries. The downside is that you can't use the defaultJiveUi property in conjunction with isolateDOM, and the tool enforces this by clearing the former if you select the latter.

Save the Javascript, HTML, and CSS for the CSS Diagnostic Tool to your environment and edit the files to use your server instances, reports, and Visualize.js code.

```
// ************ SETTINGS **********
var serverUrls = [
    "http://test.example.com:8080/jasperserver-pro",
    "http://cust.example.com:8080/jasperserver-pro",
    "http://localhost:8080/jasperserver-pro",
    "http://bi.example.com:8080/jasperserver-pro"
];
urlUsed = 3; // default used (one css loads from this server before visualize)
var reportsList = {};
// *******************************
function setupLoaderV() {
    var html,
        radioTpl = '<input id="#id" type="radio" value="#value" name="confServer" /><label
for="#id" title="#title" >#label</label><br/>';
    for (var i = 0, l = serverUrls.length; i < l; i++) {
        html = radioTpl.replace(/#id/g, "serverUrl_" + i)
            .replace(/#value/g, i)
            .replace(/#title/g, serverUrls[i])
            .replace("#label", serverUrls[i].split("/")[2]);

        $("#serverUrlsDiv").append(html);
        $("#serverUrl_" + i).prop("checked", i === urlUsed);
    }
}
function onLoad() {
    setupLoaderV();
    $( "#buttons_ui button:first" ).button({
      icons: {
        primary: "ui-icon-locked"
      },
      text: false
    }).next().button({
      icons: {
        primary: "ui-icon-locked"
      }
    }).next().button({
      icons: {
        primary: "ui-icon-gear",
        secondary: "ui-icon-triangle-1-s"
      }
    }).next().button({
      icons: {
        primary: "ui-icon-gear",
        secondary: "ui-icon-triangle-1-s"
```

```
        },
        text: false
      });
```

```
      var availableTags = [
        "ActionScript",
        "AppleScript",
        "Asp",
        "BASIC",
        "C",
        "C++",
        "Clojure",
        "COBOL",
        "ColdFusion",
        "Erlang",
        "Fortran",
        "Groovy",
        "Haskell",
        "Java",
        "JavaScript",
        "Lisp",
        "Perl",
        "PHP",
        "Python",
        "Ruby",
        "Scala",
        "Scheme"
      ];
```

```
      $( "#tags" ).autocomplete({
        source: availableTags
      });
      $("#datepicker-user").datepicker();
      $("#datepicker-user2").datepicker();
      $( "#tabs" ).tabs();

      $("#loadV").click(loadV);
      fillSheetList();
      loadCSS();
      $(window).on("keypress", function(e){
          var char = e.charCode - 49;
          if (char < 1 && char > 9) return;
          var input = $("#sheetList > li > input")[char];
          if (!input) return;
          $(input).trigger( "click" )
      });
      $("#isolateDOM").change(function() {
          $("#defaultJiveUi").attr({
              "disabled": $(this).is(':checked') ? "disabled" : null,
              "checked": false
          });
      });
  }

  function loadCSS() {

      var CSSlibs = [
          { disable: true, href: serverUrls[urlUsed] + "/themes/reset.css" },
```

```
        { disable: true, href: "//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/bootstrap.min.css"
},
        { disable: true, href:
"//cdnjs.cloudflare.com/ajax/libs/normalize/3.0.1/normalize.min.css" },
        { disable: true, href: "//cdnjs.cloudflare.com/ajax/libs/meyer-reset/2.0/reset.css" },
        { disable: true, href: "http://yui.yahooapis.com/3.16.0/build/cssreset/cssreset-min.css"
},
        { disable: true, href: "http://tantek.com/log/2004/undohtml.css" },
        { disable: true, href: "//cdnjs.cloudflare.com/ajax/libs/sanitize.css/2.0.0/sanitize.css"
},
        { disable: false, href:
"http://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/themes/vader/jquery-ui.css" }
    ];
    var head = $("head"),
        link;

    for (var i = 0, l = CSSlibs.length; i < l; i++) {
        link = $("<link rel='stylesheet' type='text/css' href='" + CSSlibs[i].href + "' />");
        head.append(link);
        if (CSSlibs[i].disable) {
            link.on("load", (function(link){
                return function() {
                    $(link)[0].disabled = true;
                    fillSheetList();
                }
            })(link));
        }
        link.on("error", (function(link){
            return function() {
                $(link)[0].href += "-LOAD_ERROR"
                fillSheetList();
            }
        })(link));
    }
}
```

```
function loadV() {
    var locale = $('#confLocale').val() || "en";
    urlUsed = $('input[name="confServer"]:checked', '#containerLoadV').val();
    var useOptimize = $('#confOptimized').is(":checked");
    $.getScript(serverUrls[urlUsed] + "/client/visualize.js?_opt=" + useOptimize.toString(),
function () {
        visualize({
            auth: {
                name: "superuser",
                password: "superuser",
                locale: locale
            }
        }, function (v) {
            fillSheetList();
            $("#loadV").remove();
            $("#loadReports").show();

            $("#addReport").on("click", function(){
                var uri = $(),
                    defaultJiveUi,
                    isolateDOM
                createReport(
                    v,
                    $("#selected_resource").val(),
                    $("#defaultJiveUi").is(':checked'),
                    $("#isolateDOM").is(':checked')
                );
```

```
            });
        });
    });
    $("#loadV").html("Loading...").attr("disabled", "disabled");
       $("#containerLoadV").addClass("disabled").children("input").attr("disabled", "disabled");
}
```

```
function createReport(v, uri, defaultJiveUi, isolateDOM) {
    var reportIndex = (+new Date() + "").substr(-5);
    console.log(reportIndex)
    reportsList[reportIndex] = "";
    fillReportsList();
    $("#reportContainer").append("<div id='vis_" + reportIndex + "'></div>");
    $("#vis_" + reportIndex).addClass("qwe");
    v.report({
        server: serverUrls[urlUsed],
        resource: uri,
        container: "#vis_" + reportIndex,
        error: function (err) {
            alert(err.message);
        },
        defaultJiveUi: { enable: defaultJiveUi },
        isolateDOM: isolateDOM || false,
        success: function () {
            fillSheetList();
            reportsList[reportIndex] = uri;
            fillReportsList();
            //processIC(v, uri);
        }
    });

}
```

```
function processIC (v, reportUri) {
    var inputControls = v.inputControls({
        resource: reportUri,
        success: function(data) {
            console.log(data);
        }
    });
}
```

```
function fillReportsList() {
    $("#reportsList").html("");
    for (var reportIndex in reportsList) {
        if (!reportsList.hasOwnProperty(reportIndex)) continue;
        var uri = reportsList[reportIndex];
        //if (uri)
        var li = $("<li>" + (/[^\/][\w\.]+$/g.exec(reportsList[reportIndex]) || "Loading...") + "
(<a href='#'>remove</a>)</li>");
        $("#reportsList").append(li);
        li.children("a").click((function (reportIndex) {
            return function (e) {
                e.preventDefault();
                $("#vis_" + reportIndex).remove();
                delete reportsList[reportIndex];
                $(e.target).parent().remove();
            };
        })(reportIndex));
```

```
    }

}
```

```
function fillSheetList() {
    var sheets = $("link");
    var checkboxLI = '<li>#index: <input type="checkbox" id="#id" checked="checked"><label
for="#id" title="#title">#label</label></li>',
        sheetPath = '',
        sheetPathSplitted = '',
        html = "";

    $("#sheetList").html("");

    for (var i = 0; i < sheets.length; i++) {

        if (sheets[i].href === null) continue;
        sheetPath = sheets[i].href;

        sheetPathSplitted = sheetPath.split("/");
        html = checkboxLI.replace(/#id/g, "sheetItem_" + i)
            .replace(/#index/g, i+1)
            .replace("#title", sheetPath)
            .replace("#label", sheets[i].label || sheetPathSplitted[sheetPathSplitted.length -
1]);

        $("#sheetList").append(html);
        $("#sheetItem_" + i).change(function (e) {
            var id = this.id.split("_")[1];
            $("link")[id].disabled = !$(this).is(':checked');
        });
        $("#sheetItem_" + i)[0].checked = !$("link")[i].disabled;
    }
}
```

The HTML for the CSS diagnostic tool contains a static list of reports to load. Add your own reports to this list.

```
<script type="text/javascript">
    window.addEventListener("load", onLoad);
</script>
<div style="width:327px;float: left;border-right:1px solid #333;margin-right:2px">
     <h4>Settings</h4>

    <div>
        <div id="containerLoadV">
            <div id="serverUrlsDiv"></div>
            <label for="confLocale">Locale: </label>
            <input id="confLocale" value="en" />
            <!-- options -->
            <br/>
            <input id="confOptimized" type="checkbox" />
            <label for="confOptimized" >- use optimized javascript </label>
            <br/>
            <button id="loadV">Load visualize</button>
            <br/>
```

```
        </div>
        <div id="loadReports" style="display:none;">
            <div>Add report:</div>
            <input id="defaultJiveUi" type="checkbox" checked="checked" />
            <label for="defaultJiveUi" >- default JIVE UI </label>
            <br/>
            <input id="isolateDOM" type="checkbox" />
            <label for="isolateDOM" >- isolate DOM </label>
            <br/>

            <select id="selected_resource" name="report" style="width:195px">
                <option value="">-</option>
                <option value="/public/Samples/Reports/1._Geographic_Results_by_Segment_Report"
                    >Geographic Results by Segment</option>
                <option value="/public/Samples/Reports/2_Sales_Mix_by_Demographic_Report"
                    >Sales Mix by Demographic</option>
                <option value="/public/Samples/Reports/3_Store_Segment_Performance_Report"
                    >Store Segment Performance</option>
                <option value="/public/Samples/Reports/04._Product_Results_by_Store_Type_Report"
                    >Product Results by Store Type</option>
            </select>
            <button id="addReport">Add</button>
        </div>
        <p>Loaded reports list:</p>
        <ul id="reportsList" style="overflow-wrap: break-word;display:inline;"></ul>
        <div style="border-top:1px solid #333;width:100%"></div>
        <h4>Stylesheets list:</h4>
        <em>Use 1-9 keys to enable\disable css libs.</em>
```

```
        <ul id="sheetList" style="display:inline;"></ul>
    </div>

    <div style="border-top:1px solid #333;width:100%"></div>

    <div style="width:320px;">
         <h4>User components</h4>

        <!-- detepicker -->
        <div style="height: 240px;">
            <div id="datepicker-user"></div>
        </div>
        <input id="datepicker-user2"/>
        <!-- /datepicker -->
        <!-- autocomplete -->
        <div class="ui-widget">
            <label for="tags">Autocomplete</label>
            <input id="tags"/>
        </div>
        <!-- autocomplete -->
```

```
        <!-- tabs -->
        <div id="tabs">
          <ul>
            <li><a href="#tabs-1">Nunc</a></li>
            <li><a href="#tabs-2">Pr dor</a></li>
            <li><a href="#tabs-3">A laia</a></li>
          </ul>
          <div id="tabs-1">
            <p>Proin elit arcu Aliquam sodales tortor vitae ipsum. Aliquam nulla. Duis aliquam
```

```
molestie erat. Ut et mauris vel pede varius sollicitudin. Sed ut dolor nec orci tincidunt
interdum. Phasellus ipsum. Nunc tristique tempus lectus.</p>
          </div>
          <div id="tabs-2">
            <p>Morbi tincidunt, tellus pellentesque pretium posuere, felis lorem euismod felis, eu
ornare leo nisi vel felis. Mauris consectetur tortor et purus.</p>
          </div>
          <div id="tabs-3">
            <p>Ut sagittis. Donec nisi lectus, feugiat porttitor, tempor ac, tempor vitae, pede.
Aenean vehicula velit eu tellus interdum rutrum. Maecenas commodo. Pellentesque nec elit. Fusce in
lacus. Vivamus a libero vitae lectus hendrerit hendrerit.</p>
          </div>
        </div>
        <!-- /tabs -->
```

```
        <!-- buttons -->
        <div id="buttons_ui">
            <button>Button with icon only</button>
            <button>Button with icon on the left</button>
            <button>Button with two icons</button>
            <button>Button with two icons and no text</button>
        </div>
        <!-- /buttons -->
    </div>

</div>
<div style="margin-left:330px;height:100%">
    <div id="reportContainer"></div>
</div>
```

Associated CSS:

```
.qwe {
    height: 100%;
}
#reportContainer {
    height: 100%;
}
body, html {
    height: 100%;
}
#containerLoadV.disabled {
    color: #666;
}

/**
  * Break something: modify the CSS here to something visibly wrong.
  * Add more elements, classes, or IDs to see if they affect Visualize.js content.
  */

table {
    font-size: 25px;

}
```

Jaspersoft Documentation and Support Services

For information about this product, you can read the documentation, contact Support, and join Jaspersoft Community.

How to Access Jaspersoft Documentation

Documentation for Jaspersoft products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the JasperReports® Server Product Documentation page.

How to Access Related Third-Party Documentation

When working with JasperReports® Server, you may find it useful to read the documentation of the following third-party products:

How to Contact Support for Jaspersoft Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join Jaspersoft Community

Jaspersoft Community is the official channel for Jaspersoft customers, partners, and employee subject matter experts to share and access their collective experience. Jaspersoft Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from Jaspersoft products. In addition, users can submit and vote on feature requests from within the Jaspersoft Ideas Portal. For a free registration, go to Jaspersoft Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

Jaspersoft, JasperReports, Visualize.js, and TIBCO are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.tibco.com/patents.

Copyright © 2005-2024. Cloud Software Group, Inc. All Rights Reserved.