Oracle® AutoVue

API Guide

Release 21.0.1

E84706-01

February 2017

ORACLE®

Oracle AutoVue API Guide, Release 21.0.1

E84706-01

Contents

Preface vii

Part I Java API Guide

1 Introduction – Java API

2 AutoVue API Packages

2.1 VueBean Package 2-1
2.1.1 Event Package 2-2
2.1.1.1 VueEvent 2-3
2.1.1.2 VueModelEvent 2-3
2.1.1.3 VueEventBroadcaster 2-3
2.1.1.4 VueFileListener 2-4
2.1.1.5 VueMarkupListener 2-4
2.1.1.6 VueViewListener 2-4
2.1.1.7 VueStateListener 2-4
2.1.1.8 VueModelListener 2-4
2.1.2 MarkupBean Package 2-4
2.1.2.1 Markup 2-5
2.1.2.2 MarkupLayer 2-5
2.1.2.3 MarkupEntity 2-5
2.2 Server Control 2-6
2.3 VueAction Package 2-7
2.3.1 AbstractVueAction 2-7
2.3.2 VueAction 2-7
2.3.2.1 Create an action that performs a single function 2-7
2.3.2.2 Create an action that performs multiple functions 2-8

3 Sample Cases

3.1 Building an AutoVue API Application 3-1
3.2 Custom VueAction 3-4
3.2.1 Action that Performs a Single Function 3-5
3.2.2 Action that Performs Multiple Functions 3-6
3.3 Directly Invoking VueActions 3-9
3.4 Markups 3-9

3.4.1 Entering Markup Mode ... 3-9
3.4.2 Checking Whether Markup Mode is Enabled ... 3-9
3.4.3 Exiting Markup Mode ... 3-10
3.4.4 Adding an Entity to an Active Markup/Layer ... 3-10
3.4.5 Enumerating Entities ... 3-10
3.4.6 Getting Entity Specification of a Given Entity ... 3-10
3.4.7 Changing Specification of an Existing Entity Programmatically ... 3-10
3.4.8 Adding a Text Box Entity ... 3-11
3.4.9 Open Existing Markup ... 3-11
3.4.10 Saving Markups to a DMS/PLM ... 3-12
3.4.11 Adding a Markup Listener to Your Application ... 3-13
3.5 Converting Files ... 3-13
3.5.1 Making a Call to a Convert Method ... 3-13
3.5.2 Converting to JPEG (Custom Conversion) ... 3-14
3.5.3 Converting to PDF ... 3-14
3.6 Printing a File to 11x17 Paper ... 3-15
3.7 Monitoring Event Notifications ... 3-15
3.8 Retrieving the Dimension and Units of a File ... 3-16

4 FAQs

4.1 MarkupBean ... 4-1
4.2 Printing ... 4-2
4.3 Upgrading ... 4-2
4.4 General ... 4-2

Part II JavaScript API

5 Introduction – JavaScript API

6 Architecture

7 AutoVue Client Launch

7.1 AutoVue Client Launch from Java Web Start ... 7-1
7.1.1 Include AutoVue JavaScript API ... 7-1
7.1.2 Instantiate an AutoVue JavaScript Object ... 7-1
7.1.3 Start AutoVue Client ... 7-3

8 AutoVue Advanced Scripting

8.1 Advanced Scripting ... 8-1
8.2 Applet API vs. New API ... 8-11

Part III ABV Guide

9 Introduction – ABV Guide

10 Hotspots

10.1 Creating a Visual Dashboard 10-1
10.2 Creating a Visual Action 10-2
10.3 Hotspot Features 10-2
10.3.1 Tooltips 10-2
10.3.2 Triggering Actions 10-3
10.4 3D Hotspots 10-3
10.4.1 Defining a 3D Hotspot 10-4
10.5 Text Hotspots in 2D and EDA Documents 10-4
10.5.1 Defining a Text Hotspot 10-4
10.6 Regional Hotspots 10-5
10.6.1 Defining Page-Specific Regional Hotspots 10-6
10.6.2 Defining Coordinates of a Box/Polygon 10-6
10.6.3 Defining a Box Hotspot 10-6
10.6.4 Defining a Polygon Hotspot 10-6
10.6.5 Invoking performHotspot() 10-7
10.7 Web CGM Hotspots 10-7

11 AutoVue Hotspot API

11.1 Hotspot INI Options 11-1
11.1.1 PDF Text Hotspot 11-1
11.1.2 PDF Text Hotspot INI Options 11-2
11.2 Define Hotspots 11-2
11.2.1 Hotspot Definition Types 11-2
11.2.2 Hotspot Definition Parameters 11-2
11.2.2.1 Common Definition Parameters 11-2
11.2.2.2 Text Definition Parameters 11-3
11.2.2.3 3D Definition Parameters 11-3
11.2.2.4 Regional Definition Parameters 11-4
11.2.3 Perform an Action on a Hotspot 11-4
11.2.3.1 Hotspot Actions 11-5
11.3 AutoVue API for ABV Integration 11-5
11.4 Interactions with Hotspots from JavaScript 11-5

12 Hotspot Samples

12.1 Adding a Hotspot 12-1
12.2 3D Hotspot 12-3
12.3 Box Hotspot 12-3
12.4 Polygon Hotspot 12-4
12.5 Text Hotspot 12-4
12.6 Text Hotspot with Visual Actions and Visual Dashboard 12-5
12.7 3D Hotspot with Visual Actions and Visual Dashboard 12-6

13 VueAction Sample

13.1 Running the VueAction Sample 13-2
13.2 Customizing the VueAction Sample 13-2

14 ABV Design and Security Recommendations

A Feedback

A.1 General AutoVue Information A-1
A.2 Oracle Customer Support A-1
A.3 My Oracle Support AutoVue Community A-1
A.4 Sales Inquiries A-1

Preface

This document has two parts:

- The first part of this document - AutoVue API Developer's Guide provides detailed technical information on the AutoVue API concepts introduced in Oracle AutoVue Integration Guide.
- The second part covers information about AutoVue JavaScript API allowing integration of AutoVue application into Web context.
- The third part of this guide discusses Augmented Business Visualization(ABV) solution that connects portion of documents to business data found in enterprise applications.

  For the most up-to-date version of this document, go to the AutoVue Documentation Web site on the Oracle Technology Network at http://www.oracle.com/technetwork/documentation/autovue-091442.html.

Audience

The first part of this document is intended for Oracle partners and third-party developers (such as integrators) who want to implement their own integration with AutoVue. Note that these developers are expected to have a good understanding of JAVA programming. The instructions in the first part of the guide serves as a good starting point for developers and professional services to become more familiar with the AutoVue API.

The second part of this document is intended for system integrators and developers looking to create links between objects in AutoVue's data model and objects in an external system.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program Web site at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents on OTN:

- *Oracle AutoVue Integration Guide*
- *VueBean and jVueApp Javadocs*
- *Oracle AutoVue Installation and Configuration Guide*
- *Oracle AutoVue Planning Guide*
- *Oracle AutoVue Integration SDK Overview*
- *Oracle AutoVue Web Services Guide*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

Java API Guide

This part includes information on the AutoVue API, its fundamental packages and classes, as well as sample code for building your own integration.

Part I contains the following chapters:

- Introduction – Java API
- AutoVue API Packages
- Sample Cases
- FAQs

1

Introduction – Java API

The AutoVue Application Programming Interface (API) is a Java-based toolset that provides tools to modify the functionality of Oracle's AutoVue client, and allows you to create your own customized Java applications based on AutoVue API components.

This document presents the technical application of the AutoVue Java API and its packages and classes. Additionally, basic and advanced applications of the AutoVue Java API are provided along with their source code.

Note: For a more general introduction to the AutoVue API, refer to the "AutoVue API Solution" section of the *Oracle AutoVue Integration Guide*. For detailed information on the packages and classes included in the AutoVue API, refer to the *VueBean and jVueApp Javadocs*.

2

AutoVue API Packages

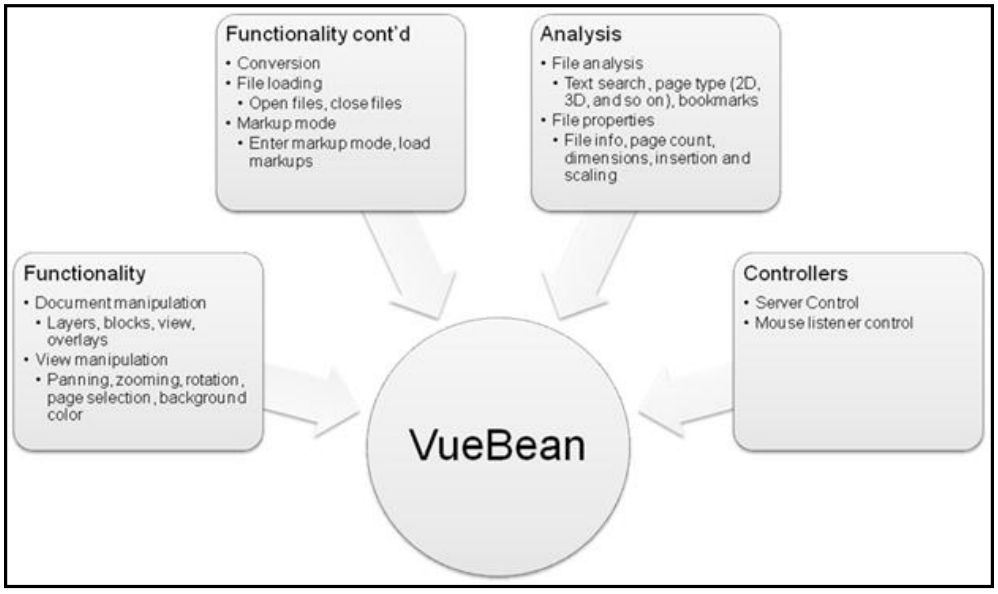
The chapter provides an overview of common classes and interfaces that are used to create a solution based on the AutoVue API.

Note: For more information on classes/packages, refer to the *VueBean Javadocs* located in the <AutoVue Installation>\docs directory.

2.1 VueBean Package

The VueBean component is central to the AutoVue client architecture. An application can embed the VueBean component and use its API to provide comprehensive support for file viewing, markup, real-time collaboration, and so on. The Figure 2–1 provides a graphical overview of how the VueBean can be used when developing your own application.

Figure 2–1 VueBean overview



Note: It is possible to have multiple instances of the VueBean class. For example, when AutoVue is in Compare mode there are three instances of the VueBean class.

A typical VueBean usage scenario is as follows:

1. Create a VueBean Object.
2. Create a server control or use the default one obtained from the VueBean.
3. Use the server control to connect to the server and open a session on it.
4. View a file by invoking the VueBean.setFile(DocID) method.

The following file types are supported by the VueBean:

- Vector files (2D and 3D)
- Raster files
- Document files (MS Word, and so on)
- Spreadsheet files
- Archive files

The file type can be queried through the VueBean.getFileType() method and file information can be retrieved through the VueBean.getFileInfo() method.

You may have to convert a file to another file type. To do so, use the VueBean.convert() method.

In its various modes, such as viewing and markup, the VueBean manages the representation of a file including the management of overlays, layers, and external references to other files or resources upon which the file depends. Use the VueBean.getResourceInfoState() method to query for resources that are attached to a file.

To search for a particular string in the file use the VueBean.search(PAN_CtlSearchInfo) method. The following is an example of how to build the PAN_CtlSearchInfo object.

```
// Construct the search object with arguments (Search String, Search Multiple
// Occurrences, Search Downwards, Wrapped Search, Match Case, Whole Word),
// in this example we search for the word "line".
PAN_CtlSearchInfo searchInfo = new PAN_CtlSearchInfo("line", true, true,
                                                     true, false, true);
```

Note: Since the VueBean is only a client-side component, the connection to the AutoVue server must be established before any operation can be performed on the VueBean. Refer to Server Control for more information.

2.1.1 Event Package

`com.cimmetry.vuebean.event`

For VueBean-specific events, the event delegation model of the VueBean is slightly different from the standard Java one. Listeners such as VueViewListener, VueFileListener, VueMarkupListener, or VueStateListener should register to the VueBean's VueEventBroadcaster object instead of the VueBean itself.

`For example: vueBean.getVueEventBroadcaster().addFileListener(listener).`

This package provides interfaces and classes for VueBean event broadcasting. Every VueBean object has an event broadcaster. Depending on the operation type, the broadcaster notifies listeners using an instance of VueEvent or VueModelEvent. The following types of events are supported:

- File events
- View events
- Markup events
- State events
- Model events

Every event type has a corresponding event listener interface which is registered to the broadcaster. Objects that are responsible for handling of events should implement one or more of the listener interfaces.

The following code sample defines and registers an event handler:

```
import com.cimmetry.vuebean.*;
import com.cimmetry.vuebean.event.*;
.
.
.
final VueBean vueBean = getVueBean();// Get the valid active VueBean
if (vueBean != null) {
  VueFileListener eventHandler = new VueFileListener() {
    public void onFileEvent(VueEvent ev) {
      switch (ev.getType()) {
      case VueEvent.ONSETFILE:
          System.out.println("Set file: " + vueBean.getFile());
          break;
      case VueEvent.ONSETPAGE:
          System.out.println("Set page: " + vueBean.getPage());
          break;
      }
    }
  };
  vueBean.getVueEventBroadcaster().addFileListener(eventHandler);
}
.
.
.
```

2.1.1.1 VueEvent

`com.cimmetry.vuebean.event.VueEvent`

VueEvent object encapsulates information for all notifications sent by VueBean and is generated for the VueFileListener, VueViewListener, VueMarkupListener and VueStateListener interfaces. The event type is used to differentiate between a view event, file event, markup event or state event.

2.1.1.2 VueModelEvent

`com.cimmetry.vuebean.event.VueModelEvent`

The VueModelEvent class handles all notifications for model-related events such as entity attributes, 3D transformation, and so on. It is generated for objects implementing VueModelListener interface.

2.1.1.3 VueEventBroadcaster

`com.cimmetry.vuebean.event.VueEventBroadcaster`

VueEventBroadcaster is used to manage the event delegation model for the VueBean. Each listener has to register to a VueEventBroadcaster to be notified of events in the VueBean. By design, each VueBean owns its own VueEventBroadcaster. However, you may find it useful to use only one VueEventBroadcaster for all beans by using the VueBean.setVueEventBroadcaster method.

2.1.1.4 VueFileListener

`com.cimmetry.vuebean.event.VueFileListener`

Objects implementing this interface listen for file event notifications (such as setting file, setting page, and so on).

2.1.1.5 VueMarkupListener

`com.cimmetry.vuebean.event.VueMarkupListener`

Objects implementing this interface listen for markup event notifications (such as entering or exiting markup mode).

2.1.1.6 VueViewListener

`com.cimmetry.vuebean.event.VueViewListener`

Objects implementing this interface listen for view event notifications (such as zoom, begin and end paint, and so on).

2.1.1.7 VueStateListener

`com.cimmetry.vuebean.event.VueStateListener`

Objects implementing this interface listen for state event notifications (such as server error, file error, and so on).

2.1.1.8 VueModelListener

`com.cimmetry.vuebean.event.VueModelListener`

Objects implementing this interface listen for model event notifications (such as model attribute, selection, transformation changes, and so on).

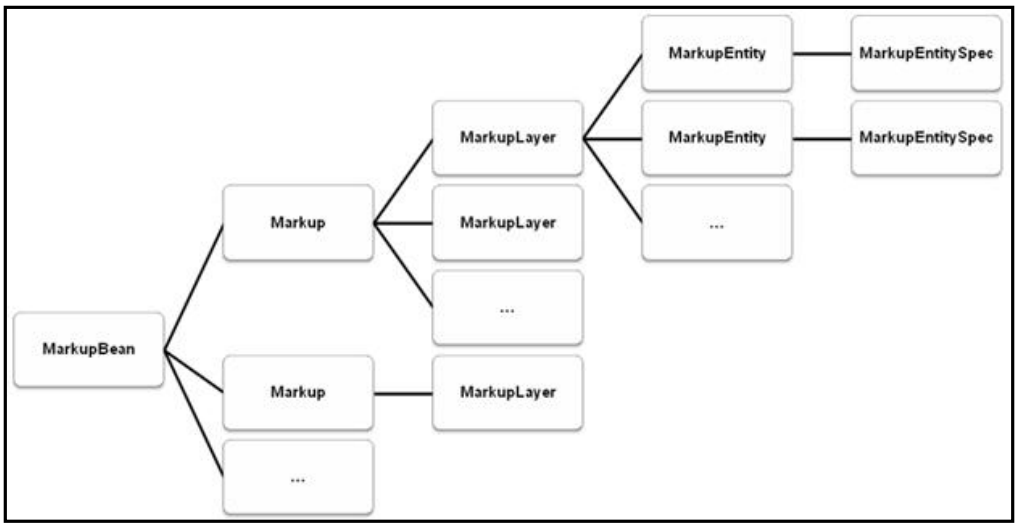
2.1.2 MarkupBean Package

`com.cimmetry.markupbean`

The top-level class for the com.cimmetry.markupbean package is the MarkupBean class. MarkupBean represents the Markup functionality in the VueBean API. Each VueBean instance can contain only one MarkupBean instance, represented by a private member variable. Through the MarkupBean class, you can add/modify/remove Markup Files, Markup Layers, and Markup Entities, as well as open and save Markup Files.

The following diagram displays how the architecture of a Markup is structured into four separate levels: Section 2.1.2.1, "Markup," Section 2.1.2.2, "MarkupLayer," Section 2.1.2.3, "MarkupEntity," and Section 2.1.2.3.1, "MarkupEntitySpec."

Figure 2–2 Markup architecture



2.1.2.1 Markup

`com.cimmetry.markupbean.Markup`

This interface represents an individual Markup file. The key functionalities are as follows:

- Get/set information regarding the Markup files, such as:
  - Name
  - Visibility
  - Whether Markup is modified
  - Whether Markup is read-only
- Get information regarding the base file
- Get the layers in the Markup

2.1.2.2 MarkupLayer

`com.cimmetry.markupbean.MarkupLayer`

This interface represents an individual Markup layer. The key functionalities are as follows:

- Get/set information regarding the specific layer, such as:
  - Name
  - Color
  - Visibility
  - Line type and width
- Get the entities in the Markup layer

2.1.2.3 MarkupEntity

`com.cimmetry.markupbean.MarkupEntity`

This interface represents an individual Markup entity. The key functionalities are as follows:

- Get/set information regarding the specific Markup entity, such as:
  - Name
  - Author
  - Date modified
  - Color
  - Line type and width
  - Tooltip text
  - Visibility
  - Selection state
- Get children entities of the specific entity
- Perform actions when user double-clicks on entity

2.1.2.3.1 MarkupEntitySpec

```
com.cimmetry.markupbean.MarkupEntitySpec
```

This class represents an entity's specification. Each entity has its own specification class that is derived from this class that defines the attributes specific to that entity's context.

For example, the specification for a rectangle entity includes attributes for the XY coordinates of all four corners, while the specification for a text entity includes attributes for the contained text as well as its alignment.

2.2 Server Control

```
com.cimmetry.vueconnection.ServerControl
```

The ServerControl class handles the server connection object and the user session. Prior to using the VueBean, you must first set its ServerControl properties, connect to the server via the connect() method, and then open a session via the sessionOpen() method.

For example:

```
import com.cimmetry.vuebean.*;
import com.cimmetry.vueconnection.ServerControl;
…
VueBean bean = new VueBean();
ServerControl control = bean.getServerControl();
try {
    control.setHost(<SERVER URL>);
    control.connect();
    control.setUser("scarlati");
    control.sessionOpen();
} catch (Exception e) {
    System.out.println("Failed to connect to AutoVue Server.");
}
…
```

Note: Set the server URL to the VueServlet URL.

For example, http://<HostName>:5098/servlet/VueServlet

2.3 VueAction Package

`com.cimmetry.vueaction`

This package provides a hierarchy of classes implementing the AutoVue action API. It can be used to add graphical user interface (GUI) elements to different contexts (such as menu bar, toolbar, status bar, and so on). For example, when a menu option is selected in the GUI, a VueAction is triggered.

To add a new action to the AutoVue client, create a new action class by extending VueAction.

Use the methods in this package to:

- Specify resources for an action. For example, menu item text, an icon, tooltip text, and so on.
- Specify which resource bundle (a properties file with resource mappings) to search in for the action's resources.
- Specify sub-actions (for example, Zoom In, Zoom Out, Zoom Previous, and so on) for the action if it can perform more than one function.
- Receive a message signifying that the action should be performed. If the action has sub-actions, the sub-action to perform is specified.
- Specify the display properties of the action or its sub-actions that appear in the GUI in the menu bars, toolbars, and popup menus. For example, specify whether the action can be selected (behaves as a checkbox) and/or whether it is enabled.
- Specify groups of sub-actions (if the action includes sub-actions) in which selection is exclusive (that is, in which only one sub-action can be selected at a time).

2.3.1 AbstractVueAction

`com.cimmetry.vueaction.AbstractVueAction`

The abstract class AbstractVueAction is the super class of all action classes. All actions performed on the session must be derived from this class or a descendent of this class.

2.3.2 VueAction

`com.cimmetry.vueaction.VueAction`

VueAction is an abstract class that extends VueActionMultiMenu. It provides a simple yet powerful interface for creating actions.

To create a new action class, you must extend this class. There are two ways to do this depending on whether your action performs a single function or multiple functions. The following sections describe both scenarios.

2.3.2.1 Create an action that performs a single function

1. Make sure your class extends VueAction.

2. In the constructor of your class, call the appropriate super constructor.

> **Note:** Since your action performs only one function, the super constructor takes the two String arguments: resource key and resource bundle. The resource bundle identifies the set of text files (one for each locale your action supports) containing the resources identified by the resource key for your action.

3. Implement a perform() method to override the one in VueAction.

> **Note:** This method is called when your action has been fired. In this method, enter your action's code.

4. Implement event handlers onFileEvent and onViewEvent to ensure that your action is enabled or disabled when appropriate. For example, if no base file has been loaded yet, your action will be disabled. However, once a file has been reloaded, your action must be enabled.
5. Create one or more resource files (one resource file per language your action supports) containing the resource keys and their values needed by your action. Together with any icon files used by your action, these files are referred to as a *resource bundle*. For an example of a resource file, refer to VueFrame_en.properties.
6. Create a copy of AutoVue's .gui file and insert the name of your new action in the appropriate location.

To view an example of implementing an action that performs a single function, refer to Action that Performs a Single Function.

2.3.2.2 Create an action that performs multiple functions

1. Make sure your class extends VueAction.
2. In the constructor of your class, call the appropriate super constructor.

> **Note:** Since your action performs multiple functions, the super constructor takes one String argument: the resource bundle name. The resource bundle name identifies the set of text files (one for each language your action supports) containing the resources for your action.

3. After you call the super constructor, call defineSubAction() to define each sub-action your action can perform.

> **Note:** In each case, specify the name by which you want to refer to the sub-action and its resource key. The resource key identifies where to find the resources for your action (for example, menu item text, icon, tooltip text and so on) in your resource bundle. Optionally, you can call defineExclusiveGroup() to define a subset of your sub-actions that form an exclusive group. That is, sub-actions that are selectable where only one can be selected at a time.

4. Implement a performSubAction(String) method to override the one in VueAction.

Note: This method is called when your action's sub-action has been fired. The method is passed the name of the sub-action fired, so that you will know which one to perform. In this method, enter your sub-action's code.

5. Implement event handlers onFileEvent and onViewEvent to ensure that your sub-actions are enabled or disabled when appropriate. For example, if no base file has been loaded, your sub-action will be disabled. However, once a file has been reloaded, your sub-actions must be enabled.
6. Create one or more resource files (one resource file per language that your action supports) containing the keys and values needed by your action.

Note: Together with any icon files used by your action, these files are referred to as a *resource bundle*.

7. Create a copy of AutoVue's .gui file and insert the name of your new action in the appropriate location. You must also specify the appropriate sub-actions.

To view an example of implementing an action that performs multiple functions, refer to Action that Performs Multiple Functions.

Note: When deploying VueAction jar on the web, you have to properly sign the jar. Refer to how to Configure and Run the AutoVue VueActionSample (Doc ID 1677471.1)

3

Sample Cases

This chapter provides information on typical use cases you may come upon when creating an AutoVue API application or adding enhanced functionality to the AutoVue client. Refer to the VueBean and jVueApp Javadocs for more information.

Important: When executing a task in sequence you must make sure the previous task is completed before starting a new one. For example, when opening a file, the process should listen for the file event VueEvent.ONPAGELOADED to be notified. In the event of a file error, the state even VueEvent.ONFILEERROR is notified. When loading markups, listen and wait for the markup event MarkupEvent.ONMARKUPLOADED to be notified.

Note: Throughout this document, m_vueBean is used as a valid active VueBean object and m_JVue as a valid JVueApp object. This is done assuming that the methods or segments of code that use objects have access to a class which owns them.

3.1 Building an AutoVue API Application

A good starting point with the AutoVue API is to create an application that opens and displays a file.

This section provides detailed steps for creating a file open application using the AutoVue API.

1. Import required packages.

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import com.cimmetry.util.Messages;

import com.cimmetry.core.*;
import com.cimmetry.vuebean.*;
import com.cimmetry.vueconnection.ServerControl;
```

2. Create a Java class, ApplicationSample, that can be run as a stand-alone application, and declare all external parameters and internal data members.

```
public class ApplicationSample {
  private String m_host = "http://<HostName>:5098/servlet/VueServlet";
```

```
  private String m_user = "guest";
  private String m_fileName = null;
  private String m_verbose = null;
  private String m_format = "AUTO";
  // Internal data members
  private VueBean m_vueBean = null;
  private ServerControl m_control = null;
  private static JFrame m_frame = null;
  private JMenu m_fileMenu = null;
}
```

3. Create stand-alone application support.

```
public static void main(final String args[]) {
  ApplicationSample app = new ApplicationSample();
  app.init(args);
}
public void init(final String[] args) {
  switch (args.length) {
    case 4:
      m_verbose = args[3];
    case 3:
      m_fileName = args[2];
    case 2:
      m_user = args[1];
    case 1:
      m_host = args[0];
    default:
      break;
  }
  init();
}
```

4. Initialize the application.

```
public void init() {
  // Setup verbosity
  if (m_verbose != null && m_verbose.length() > 0) {
    Messages.setVerbosity(m_verbose);
  }
…
```

> **Note:** The init() method continues until step 13.

5. Establish a connection with the server.

```
m_control = new ServerControl();
try {
  m_control.setHost(m_host);
  m_control.connect();
} catch (Exception e) {
  System.out.println("Unable to connect to:"+m_host);
  e.printStackTrace();
  return;
}
```

6. Open the session.

```
try {
  m_control.setUser(m_user);
```

```
  m_control.sessionOpen();
} catch (Exception e) {
  System.out.println("Unable to open session for " + m_user);
  e.printStackTrace();
  return;
}
```

7. Initialize the frame.

```
m_frame = new JFrame("VueBean Sample");
m_frame.setBounds(100, 100, 640, 480);
m_frame.addWindowListener( new WindowAdapter() {
  public void windowClosing( WindowEvent e) {
    destroy();
  }
});
```

8. Set the menus and actions.

```
setMenuBar();
```

9. Create the bean.

```
m_vueBean = new VueBean(m_format);
m_vueBean.setServerControl(m_control);
m_vueBean.setBackground(Color.lightGray);
```

10. Add the VueBean to the frame.

```
m_frame.getContentPane().add(m_vueBean);
```

11. Display the frame.

```
m_frame.setVisible(true);
```

12. Show the file.

```
updateFile();
}// Closing bracket for init() method
```

> **Note:** This step marks the end of the init() method.

13. Close the session.

```
public void destroy() {
  try {
      m_control.sessionClose();
  } catch (Exception ex) {
      ex.printStackTrace();
  }
  m_frame.setVisible(false);
  m_frame.dispose();
  System.exit(0);
}
```

14. Get the attached VueBean.

```
public VueBean getVueBean() {
  return m_vueBean;
}
```

15. Get the attached frame.

```
public JFrame getFrame() {
  return m_frame;
}
```

16. Get the file menu.

```
protected JMenu getFileMenu() {
  return m_fileMenu;
}
```

17. Get the frame. The following method sets the client's menu bar to File Open, Print, and Exit.

```
public void setMenuBar() {
  m_fileMenu = new JMenu("File");
  JMenuItem menuItem;
  // File open menu item
  menuItem = m_fileMenu.add(new JMenuItem("Open"));
  menuItem.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    showFile();
  }
  });
  // set the client's menu bar
  JMenuBar menu_bar = new JMenuBar();
  m_frame.setJMenuBar(menu_bar);
  menu_bar.add(m_fileMenu);
}
```

18. Load the file.

```
public void updateFile() {
  // Set the vuebean's file
  if (m_fileName != null && !m_fileName.equals("")) {
    m_vueBean.setFile(new DocID(m_fileName));
    m_vueBean.setBackground(Color.lightGray);
  }
}
```

19. Display the client-side (upload) File Open dialog and set the selected file in the bean.

```
public void showFile() {
  FileDialog openDlg = new FileDialog(m_frame, "File Open", FileDialog.LOAD);
  openDlg.setVisible(true);
  m_fileName = "upload://"+openDlg.getDirectory() + openDlg.getFile();
  openDlg.dispose();
  updateFile();
}
}//end of class
```

> **Note:** End of class ApplicationSample. In order to run the application properly, an AutoVue server needs to be running on either a local or remote host that is specified through command line arguments. Refer to step 3 for the definition of each argument.

3.2 Custom VueAction

This section presents examples implementing a custom VueAction class.

3.2.1 Action that Performs a Single Function

The following example shows how to implement a custom action for AutoVue that performs a single function. That is, when a user double-clicks on a hotspot, a dialog appears and lists all components of a drawing that are represented by the hotspot.

For information on AutoVue's hotspot capabilities, refer to the ABV Guide.

> **Note:** The following are segments of the source code of the VueAction example to illustrate the essential steps of creating a custom action, it may not compile if you just copy and paste the code here. For the complete source code, refer to PartListAction.java.

1. Import all required packages.

```
import java.awt.*;
import java.util.Vector;
import com.cimmetry.vuebean.*;
import java.awt.event.*;
import com.cimmetry.vuebean.event.*;
import com.cimmetry.vueframe.*;
import com.cimmetry.vueframe.hotspot.*;
import com.cimmetry.core.*;
import com.cimmetry.dialogs.VueBasicDialog;
import com.cimmetry.vueaction.VueAction;
import com.cimmetry.gui.*;
```

2. Make your class extend VueAction.

```
public class PartListAction extends VueAction { …}
```

3. In the constructor of your class, call the appropriate super constructor. Since this action only performs a single function, a call to the super-constructor of VueAction takes this action's resource key as well as its resource bundle name.

```
public PartListAction() {
      super("LIST_PARTS",RESOURCE_BUNDLE_NAME);
      setViewListener(true);
}
```

> **Note:** The resource bundle name here is the common part of resource bundle files for different languages. The actual name of a resource bundle file should include the language suffix and file extension. For example, PartListAction_en.properties is the resource bundle file for English.

4. Implement a perform method for this action.

```
public void perform() {
      PartInfo[] parts = new PartInfo[m_cart.size()];
      m_cart.copyInto(parts);
      PartListDialog dialog = new PartListDialog(getFrame(), parts);
      dialog.show();
}
```

5. Implement the event handlers onFileEvent and onViewEvent to notify when a file has changed and to update the user-interface.

```
public void onFileEvent(VueEvent e) {
```

```
      switch (e.getEvent()) {
         case VueEvent.ONPAGELOADED:
         setEnabledByCurrentState();
         break;
      }
}
public void onViewEvent(VueEvent e) {
      switch(e.getEvent()) {
         case VueEvent.ONLINKCLICKED:
         Object[] params = (Object[]) e.getParameter();
          MouseEvent me = (MouseEvent) params[0];
          if (me.getClickCount() == 2) {
             Object link = params[1];
             if (link instanceof HotSpot) {
                  HotSpot hotspot = (HotSpot) link;
                  PartInfo part = getPartInfo(hotspot);
                  m_cart.addElement(part);
             }
         }
          break;
          default:
          super.onViewEvent(e);
          break;
     }
}
```

6. The dialog that lists all components of a drawing extends VueBasicDialog. You must implement your own constructor that calls the super-constructor and over-rides buildDialog() and buttonAction(int).

```
public static class PartListDialog
extends
      VueBasicDialog
implements
      ActionListener (…)
protected void buildDialog() {

  super.buildDialog();
…
}
protected void buttonAction(int index){…}
```

7. You must define a model for the table that represents the displayed product parts list.

```
public static class PartListModel implements CTableModel { …}
```

8. Close the PartListDialog() method.
9. Get a PartInfo associated with a given hotspot.

```
private PartInfo getPartInfo (HotSpot hotspot) {
            return new PartInfo(hotspot.getDefinitionKey(),
            hotspot.getHotSpotKey(),
            hotspot.getProperty(HotSpot.PROPERTY_DESCRIPTION));
)
```

3.2.2 Action that Performs Multiple Functions

The following example shows how to implement a custom action for AutoVue that performs multiple tasks. The custom action consists of several related sub-actions that access information about parts of a model. One sub-action permits the user to order a

part, another permits the user to display part information, and another sub-action displays a list of all the model's parts.

> **Note:** The following are segments of the source code of the VueAction example to illustrate the essential steps of creating a custom action, it may not compile if you just copy and paste the code here. For the complete source code, refer to *PartCatalogueAction.java*.

1. Make your class extend VueAction.

```
public class PartCatalogueAction extends VueAction {
  private static final String RESOURCE_BUNDLE_NAME ="/PartCatalogueAction";

  // Names of the sub-actions used in *.gui file
  private static final String ORDER_SUBACTION = "Order";
  private static final String LIST_PARTS_SUBACTION = "ListParts";
  private static final String SHOW_INFO_SUBACTION = "ShowInfo";
  …
}
```

2. In the constructor of your class, call the appropriate super constructor.

```
public PartCatalogueAction() {
        super(RESOURCE_BUNDLE_NAME);
        defineSubAction(SHOW_INFO_SUBACTION,"SHOW_PART_INFO");
```

> **Note:** The resource bundle name used here is the common part of resource bundle files for different languages. The actual name of a resource bundle file should include the language suffix and file extension. For example, PartCatalogueAction_en.properties is the resource bundle file for English.

3. Call defineSubAction to define each sub-action your action can perform.

```
defineSubAction(ORDER_SUBACTION,"ORDER_PART");
defineSubAction(LIST_PARTS_SUBACTION,"LIST_PARTS");
defineSubAction(SHOW_INFO_SUBACTION,"SHOW_PART_INFO");
}
```

4. Implement a performSubAction(String) method to override the one in VueAction.

```
public void performSubAction(String subActionName) {
        if (subActionName.equals(ORDER_SUBACTION)) {
          //Code for performing the "Order" subaction
            …
        } else if (subActionName.equals(LIST_PARTS_SUBACTION)) {
            //Code for performing the "List Parts" subaction
            …
          }
…
}
```

5. Implement the event handlers onFileEvent and onViewEvent to ensure that your sub-actions are enabled or disabled when appropriate.

```
public void onFileEvent(VueEvent e) {
  switch (e.getEvent()) {
        case VueEvent.ONSETFILE:
            //Code for handling ONSETFILE event
            …
        case VueEvent.ONPAGELOADED:
```

```
            //Code for handling ONPAGELOADED event
            setEnabledByCurrentState();
            …
            break;
        }
    }
public void onViewEvent(VueEvent e) {
  switch(e.getEvent()) {
        case VueEvent.ONVIEWCHANGED:
            //Code for handling ONVIEWCHANGED event
            setEnabledByCurrentState();
            …
            break;
        case VueEvent.ONOPTIONSCHANGED:
            //Code for handling ONOPTIONSCHANGED event
            …
            break;
        }
    }
```

6. Create one or more resource files, one per language your action supports, containing the keys and values needed by your action. For example:

```
…
FILE_MARKUP_NEW_MARKUP=&New Markup, 32_new_markup_red.png, New Markup
FILE_MARKUP_OPEN=&Open..., 57_markup_red.png, Open Markup(s)
FILE_MARKUP_SMALL=     &Markup, 57_markup_red_small.png, Markup
FILE_MRU=Recent Files
FILE_NOTFOUND=File not found.
FILE_NOTSUPPORTED=This file format is not supported by your server.
FILE_NOTUPLOADED=Failed to upload file.
FILE_OPEN=&Open...\\tCTrL+O, 59_open.png, Open File
FILE_OPEN_SERVER=Open from &Server..., , Open a file from the server
…
```

Similarly, in our resource bundle file for English language PartCatalogueAction_en.properties, it should contain the resource keys for the PartCatalogueAction shown in the following:

```
…
ORDER_PART = &Order Part, order_part.png, Order a part
LIST_PARTS = &List Parts, list_parts.png, List product parts
SHOW_INFO_SUBACTION = &Show Part Info, show_info.png, Show part information
…
```

> **Note:** Each resource key has three entries separated by a comma ",". The first entry (for example, &Order Part) is the text displayed on the GUI item (such as a menu item or toolbar button) and the ampersand "&" defines a shortcut key. The second entry (for example, order_part.png) is the file of the icon displayed on its GUI item. The third entry is the tooltip text for the GUI item. The second and third entries are optional. You should get the icon files ready if needed and add them to the JAR file for your custom action.

7. Make a copy of AutoVue's default.gui file located in the <AutoVue Installation Root>\bin directory, and insert the name of your new action in the appropriate locations of your GUI file. Note that for an action that performs multiple tasks, you must also specify the appropriate sub-actions.

Note: For information on how PartCatalogueAction sub-actions are inserted into a menu bar, tool bar, and custom pop-up menu, refer to default.gui and the custom.gui file located in the <AutoVue Installation Root>\examples\VueActionSample\ directory.

8. To allow the custom action to take effect, you may need to create a JAR file with your custom VueAction classes and all resource files they depend on. For example, for the resource bundle files for different languages and icon files, if any, place your JAR file under AutoVue's bin directory or its web root directory and include your JAR file in the classpath of the stand-alone AutoVue (JVueApp) application.
9. You must specify the name of the modified GUI file through Applet or Command line parameters. For more information, refer to the "Customizing the GUI" section of the *Oracle AutoVue Installation and Configuration Guide*.

3.3 Directly Invoking VueActions

You can develop your own customized user interface in an HTML page that incorporates AutoVue functionality. To do so, you must call invokeAction() of the AutoVue JavaScript Object from the HTML page (see JavaScript API for more details). This call to the action can be done purely through JavaScript. For a list of actions/subactions, refer to the default.gui file located in <AutoVue Install Root>\bin directory.

Example 3–1 invokeAction()

```
invokeAction(VueActionFileOpen) //Displays the File Open dialog
```

3.4 Markups

The following sections describes some ways to execute common Markup actions.

Note: Various MarkupBean functionalities (and various functionalities throughout the AutoVue API) require the use of the *Property* class. This class is used to define various property hierarchies for other classes in the API.

3.4.1 Entering Markup Mode

```
VueBean.setMarkupModeEnabled(true)
```

Checks whether the MarkupBean member is null, and if so:

- Instantiates a new MarkupBean object
- Gets the markup settings from the user's profile
- Sets the markup-specific mouse listeners
- Points the VueBean's MarkupBean member to the new instance
- Broadcasts VueEvent.ONENTERMARKUPMODE

3.4.2 Checking Whether Markup Mode is Enabled

```
VueBean.isMarkupModeEnabled()
```

Checks whether the MarkupBean member is enabled.

3.4.3 Exiting Markup Mode

```
VueBean.setMarkupModeEnabled(false)
```

Checks whether the MarkupBean member is null, and if not:

- Sets the MarkupBean member to null
- Removes markup-specific mouse listeners
- Saves markup settings into the user's profile
- Broadcasts VueEvent.ONEXITMARKUPMODE

3.4.4 Adding an Entity to an Active Markup/Layer

```
MarkupBean.setMarkupEntityClass(<class name of desired markup entity>)
MarkupBean.setActionMode(MarkupBean.ACTION_MODE_ADD)
```

Adds a new markup entity to the active layer in an active Markup (based on the class name provided) through user input from the GUI. To programmatically add a markup entity, you must call: MarkupBean.addMarkupEntity(MarkupEntitySpec spec)

3.4.5 Enumerating Entities

```
MarkupLayer.getEntities()
```
or
```
MarkupBean.getMarkupEntities(MarkupLayer layer)
```

Returns an array of MarkupEntity objects in a markup layer.

3.4.6 Getting Entity Specification of a Given Entity

```
MarkupBean.getMarkupEntityFullSpec(MarkupEntity ent)
```

You must pass in the specific entity for MarkupBean to return its specification.

3.4.7 Changing Specification of an Existing Entity Programmatically

```
MarkupBean.exchangeMarkupEntity(MarkupEntity a, MarkupEntity b)
```

Allows you to dynamically change the properties of an existing entity. That is, it replaces markup entity *a* with markup entity *b*. Some properties can be directly changed via the following set methods of MarkupEntitySpec inherited from the MarkupGraphicSpec parent class:

- setColor
- setFillColor
- setFilled
- setFilltype
- setFont
- setLineType

- setLineWidth

For other properties, such as the entity position, entity size, entity text content, and so on, there are no set methods directly on the specification. As a result, you must do the following:

1. Create a new specification instance (with the new properties).
2. Create a new entity instance (with the new specification).
3. Use exchangeMarkupEntity to replace the existing entity.
4. Make a call to MarkupBean.repaint().

3.4.8 Adding a Text Box Entity

The following code shows how to add a text box entity programmatically.

```
import com.cimmetry.markupbean.*;
import com.cimmetry.gui.*;
.
.
.
public void addTextBox(String text){

  m_vueBean.setMarkupModeEnabled(true);

  CTextPane textPane = GUIFactory.createTextPane();
  textPane.setText(text);
  byte[] textRTF = textPane.getRTF();
  PAN_CtlRange rect = new PAN_CtlRange(m_vueBean.getViewExtents());
  rect.scale(0.2);
  TextBoxSpec spec = new
  TextBoxSpec(m_vueBean.getMarkupBean().getMarkupEntitySpec(),
          rect.min, textRTF, rect.max,TextBoxSpec.MRK_ALIGN_BOTTOMCENTER);
  m_vueBean.getMarkupBean().setMarkupEntityClass(spec.getEntityClassName());
  m_vueBean.getMarkupBean().addMarkupEntity(spec);
}
```

3.4.9 Open Existing Markup

```
MarkupBean.readMarkup(InputStream is)
```

InputStream can be relative to the client (for example, a locally-saved Markup), relative to the AutoVue server (for example, managed by AutoVue's markups.map file) or from a DMS/PLM/ERP.

To read a Markup from the AutoVue server, you first must get the InputStream by reading the Markup *Property* from the VueBean, and then choose a child property (that represents a Markup file) you want to read into the stream. The following code illustrates how to create a markup, save it, and then read it into the MarkupBean.

```
import com.cimmetry.markupbean.*;
.
.
Property[] name = {new Property(Property.PROP_DOC_NAME, <your Markup name>)};
Property prop = new Property(Property.PROP_MARKUP, name);
ByteArrayOutputStream os = new ByteArrayOutputStream();
m_markupBean.writeMarkup(os);
m_vueBean.writeMarkup(prop, os);
Property masterMarkup = m_vueBean.getMarkupProperty();
Property[] listMarkups = masterMarkup.getChildrenWithName(Property.PROP_MARKUP);
```

```
Property aMarkup = listMarkup[0];
InputStream is = m_vueBean.readMarkup(aMarkup);
m_markupBean.readMarkup(is);
…
```

3.4.10 Saving Markups to a DMS/PLM

Note: This example is not applicable if you are building an ISDK-based application.

The following example uses the same concept as saving a Markup back to the AutoVue server; you must set the appropriate *Property* and build the OutputStream. In order to build the Markup property, you need to first read the CSI_Markups property so that you can retrieve the values that the user sets in the Markup Save dialog.

```
private void saveMarkupToDMS() {
  // Gets the master markup property for the current file, that is,
  // the property containing the GUI and the markup list
  Property propMaster = m_vueBean.getMarkupProperty();

  // If none, the an output appears stating "Could not get master markup property"
  if ( propMaster == null ) {
    System.out.println("Could not get master markup property!");
    return;
  } else {
       // Get the GUI child property under master markup property
      Property[] listGuiProp =propMaster.getChildrenWithName(Property.PROP_GUI);
      if (listGuiProp == null || listGuiProp.length != 1) {
         System.out.println("No valid GUI property!");
         return;
      }
         Property propGui = listGuiProp[0];
      // Get the user field (Edit) child property under GUI property
      Property[] listEditProp =propGui.getChildrenWithName(Property.PROP_GUI_
         EDIT);
      if (listEditProp == null || listEditProp.length != 1) {
        System.out.println("No valid GUI edit property!");
        return;
      }
      Property propGuiEdit = listEditProp[0];
     // Get the list of user fields from save dialog all children items under GUI
     // edit property
     Property [] itemsEdit = propGuiEdit.getChildren();
    // ToDo: Use the list of edit items (GUI element) to construct a
    // save dialog to get user input for properties under PROP_GUI_EDIT.
    // Assume the input for attribute "CSI_DocName" we got from the dialog
    // is "myMarkup" and the input for attribute "CSI_MarkupType" is
    // "Normal", now the following code using the inputs to construct
    // the markup property contains these two attributes. In reality
    // there can be more than two attributes.
    Property [] listProp = {
        new Property("CSI_DocName", "myMarkup"),
        new Property("CSI_MarkupType", "Normal")
    };
    // Create a Markup property with the specified name & type properties
    Property propMarkup = new Property(Property.PROP_MARKUP, listProp);
    // Save the Markup
    try {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        m_vueBean.getMarkupBean().writeMarkup(os);
        m_vueBean.writeMarkup(propMarkup, os);
```

```
    } catch (MarkupIOException e) {
        System.out.println("Markup IO Exception!");
    }
  }
}
```

3.4.11 Adding a Markup Listener to Your Application

```
MarkupBean.getMarkupBroadcaster().addMarkupEventListener(MarkupEventListener mel);
```

A Markup listener listens for Markup events related to creating/saving/deleting Markups, Markup entities, Markup file information, fonts, Markup status, and so on. Note that you must implement the com.cimmetry.MarkupBean.event.MarkupEventListener interface (thereby implementing the onMarkupEvent method).

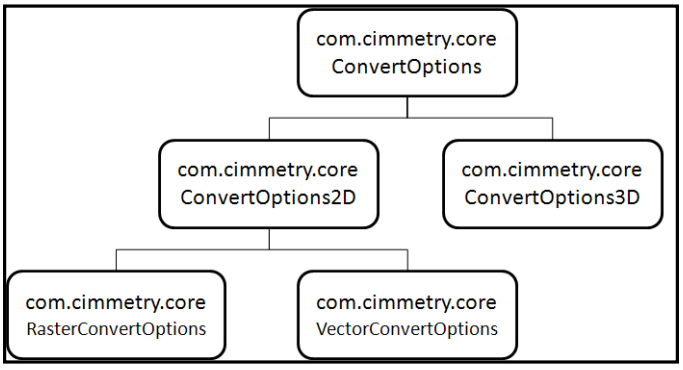
3.5 Converting Files

The following sections discuss how to execute common Conversion actions such as making a call to convert, converting an image to a JPEG using a custom conversion, and converting a vector file to a PDF. In some cases, there are additional methods to achieve the same functionality. Refer to the *VueBean Javadocs* for more information.

Note: Conversion of 3D files or pages containing 3D data is no longer supported.

The class hierarchy for conversion is as follows:

Figure 3–1 Conversion class hierarchy



Note: The classes represent the format which you are converting a file to. For example, if you are converting to a vector format, you should define a VectorConvertOptions and pass it into the conversion method.

3.5.1 Making a Call to a Convert Method

```
com.cimmetry.vuebean.VueBean.convert(ConvertOptions opts)
```

or

```
com.cimmetry.jvue.JVueApp.convertFile
```

Once the convert options are defined, you must call one of the methods to convert.

> **Note:** When making a call from the VueBean you must call VueBean.convert. When making a call from the JVueApp layer, you must call JVueApp.convertFile.

3.5.2 Converting to JPEG (Custom Conversion)

To convert an image to a JPEG, you must use the encode() method that Java provides as part of the com.sun.image.coded.jpeg.JPEGImageEncoder interface. This method encodes buffers of the image data in JPEG data streams. To use this interface, you must provide the image data in raster format or a BufferedImage. The following example illustrates how to use this interface with the AutoVue API:

```
import java.io.*;
import java.awt.*;
import java.awt.image.*
import com.cimmetry.core.*;
import com.sun.image.codec.jpeg.*;
…

double scaling=0.5; BufferedImage bi = new BufferedImage(
  (int)( m_vueBean.getWidth()*scaling),(int)(m_vueBean.getHeight()*scaling),
  BufferedImage.TYPE_INT_RGB);

//Create or get Graphics and RenderOptions object here
Graphics2D g = bi.createGraphics();
RenderOptions optsRender = new RenderOptions();
//TODO: Initialize the Graphics object and RenderOptions object properly such
//as setting the source and destination.
try {
   m_vueBean.renderOntoGraphics(g,optsRender);
   FileOutputStream out = new FileOutputStream("c:\\temp\\my.jpeg");
   JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
   JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(bi);
   //TODO: Use the JPEGEncodeParam Interface to set the encoder parameters.
   encoder.encode(bi, param);
   out.flush();
   out.close();
} catch (Exception e) {
   System.out.println("Exception while converting to JPEG ");
   return;
}
…
```

3.5.3 Converting to PDF

To convert a vector file to a PDF you must perform the following steps:

- Create new VectorConvertOptions() object
- Set all appropriate convert options
- Call VueBean.convert and pass in the convert options

The following convertToPDF() method converts a vector file to a PDF.

```
public void convertToPDF() {
```

```
      VectorConvertOptions opts = new VectorConvertOptions();

      opts.setStepsPerInch(1);
      PAN_CtlFileInfo fi = m_vueBean.getFileInfo();
      PAN_CtlRange ps = m_vueBean.getPageSizeEx();

      if (fi.getType() == fi.PAN_DocumentFile) {
        ps = fi.getPageSize();
       }
      opts.setInputRange(ps);
      opts.setArea(ConvertOptions2D.AREA_EXTENTS);
      opts.setScaleFactor(100);
      opts.setScaleType(ConvertOptions2D.TYPE_SCALE);
      opts.setUnits(Constants.UNITS_INCH);
      opts.setPages(ConvertOptions2D.PAGES_ALL);
      opts.setFromPage(1);
      opts.setToPage(fi.getPagesNumber());
      opts.setFormat("PCVC_PDF");
      opts.setSubFormatID(0);
      opts.setFileName("c:\\output.pdf");

      //Uploads all currently loaded markups to the AutoVue server
      Property[] p = m_ vueBean.uploadMarkups();

      opts.setProperties(p);
      m_ vueBean.convert(opts);
}
```

3.6 Printing a File to 11x17 Paper

The following code prints a file to 11x17 paper size using the com.cimmerty.common.PrintProperties and com.cimmetry.common.PrintOptions classes.

```
import com.cimmetry.common.PrintProperties;
import com.cimmetry.common.PrintOptions;
public void printFile() {
  PrintProperties paramPrintProperties = new PrintProperties();
  PrintOptions po = new PrintOptions();
  po.setPrinter("AutoVue Document Converter");
  po.setPaperSize(po.PAPER_11X17);
  paramPrintProperties.setOptions(po);
  // The second parameter will enable the bypass of the Windows dialog
  m_ JVue.printFile(paramPrintProperties, true);
}
```

3.7 Monitoring Event Notifications

`com.cimmetry.vuebean.event`

If you have a requirement to programmatically execute specific file actions (such as rotation, zooming, and so on) as soon as a file has finished loading, you must monitor for the appropriate event notifications. If you do not check for file load completion, you might call a file action too early which may lead to errors.

The VueBean includes a set of notifications known as VueEvents. You can set up a listener to catch VueEvents, and catch the specific events that represent the completion of a file loading. In order to catch file loading completion, you must use a file listener, with the VueFileListener interface.

The steps are as follows:

1. Implement your own VueFileListener.
2. In the onFileEvent method, check for occurrence of the Vue.Event.ONPAGELOADED event.
3. Implement your code to be executed when the Vue.Event.ONPAGELOADED event is detected.
4. Add your file listener to the VueBean.
5. Add this to your application.

3.8 Retrieving the Dimension and Units of a File

The following sample code shows how to get the dimensions and units of a file.

```
PAN_CtlDimensions pctlDim = m_vueBean.getFileInfo().getDimensions();
double height = pctlDim.getHeight();
double width = pctlDim.getWidth();
double depth = pctlDim.getDepth();
int units = m_vueBean.getFileInfo().getInsertion().units;
```

4

FAQs

The following sections provide frequently asked questions regarding the AutoVue API.

4.1 MarkupBean

Q: How do you determine the layer that a given entity is in?

A: Get the entity's specification and then get the layer from the specification.

Q: Do I have to implement the entire text editing dialog for the Text/Leader/Note entity?

A: No. The text editing dialog is inherent to these entities.

Q: An entity specification is tied to a given entity. Why was it decided to have an entity specification tied to the MarkupBean?

A: The entity specification on the MarkupBean was designed to be a reference to the most recent specification settings. When you create a new Markup entity, it defaults much of its specification attributes to the current specification in the MarkupBean. To retrieve the most recent specification settings, you can call MarkupBean.getMarkupEntitySpec().

Note: The other two methods MarkupBean.getMarkupEntitySpec(MarkupEntity ent) and MarkupBean.getMarkupEntityFullSpec(MarkupEntity ent) are for when you need to get the specification of a specific entity.

Q: What is the difference between MarkupGraphicSpec and MarkupEntitySpec? Why are the specs such as ArcSpec subclass not derived directly from MarkupGraphicSpec?

A: The MarkupGraphicSpec is a top-level specification that manages visual attributes such as color, fill type, and so on. The MarkupEntitySpec is a top-level specification that has access to the overall structure such as the MarkupBean, Markups, layers, pages, and so on. Since MarkupEntitySpec extends MarkupGraphicSpec, and this is the base class for all markup entities, the ArcSpec subclass is derived from MarkupEntitySpec.

Q: Can you work with MarkupBean independent of VueBean?

A: In theory it is possible to instantiate and work with MarkupBean without having a VueBean. However, there are not many use cases or practical reasons where this would be valuable.

Q: Are the Markup tree and Markup toolbars from the AutoVue client accessible if I am building a custom application from VueBean/MarkupBean?

A: No. The UI such as toolbars and Markup tree are part of the "JVueApp" class. If you build your solution using the JVueApp class you can use or customize this UI. However, if you build your solution directly from VueBean you need to implement your own UI.

Q: Is it possible to add AutoVue markup capabilities to a third-party application?

A: Yes. There are two primary ways to add markup entities using MarkupBean:

- With user input, using MarkupBean.setActionMode(MarkupBean.ACTION_MODE_ADD)
- Programmatically, using MarkupBean.addMarkupEntity(MarkupEntitySpec spec)

4.2 Printing

Q: What is the purpose of com.cimmetry.core.PrintInfo class?

A: It is used to pass information between the client and server.

4.3 Upgrading

Q: Will my custom code still work when I perform an AutoVue upgrade?

A: Yes. You must recompile your custom code against the latest release and update the path to the new jvue.jar.

4.4 General

Q: Can I perform file type-dependent operations?

A: Yes. You can do so by using the getFileInfo() method. The PAN_CtlFileInfo object that is returned can be queried to determine file format (such as vector, raster, spreadsheet, document, archive, or a database file).

Q: Can I delete server-side Markups when using the VueBean API?

A: No. It is not currently possible to programmatically delete server-managed Markups (referenced in the markups.map file on the server) using the VueBean API.

Part II

JavaScript API

This part covers information about AutoVue JavaScript API allowing integration of AutoVue application into Web context, in a simple way.

Part II contains the following chapters:

- Introduction – JavaScript API
- Architecture
- AutoVue Client Launch
- AutoVue Advanced Scripting

Note: This part applies only to the Oracle AutoVue Client/Server Deployment product.

5

Introduction – JavaScript API

The AutoVue JavaScript API is wrapped into a JavaScript Object allowing launching an AutoVue client from a WEB context. It supports a scripting API that the browser can use to interact with AutoVue, and allows you to write your own customized HTML client interfacing with AutoVue.

This document presents the technical application of AutoVue JavaScript API and its scripting commands. Additionally, basic applications of the AutoVue JavaScript API are provided along with their source code.

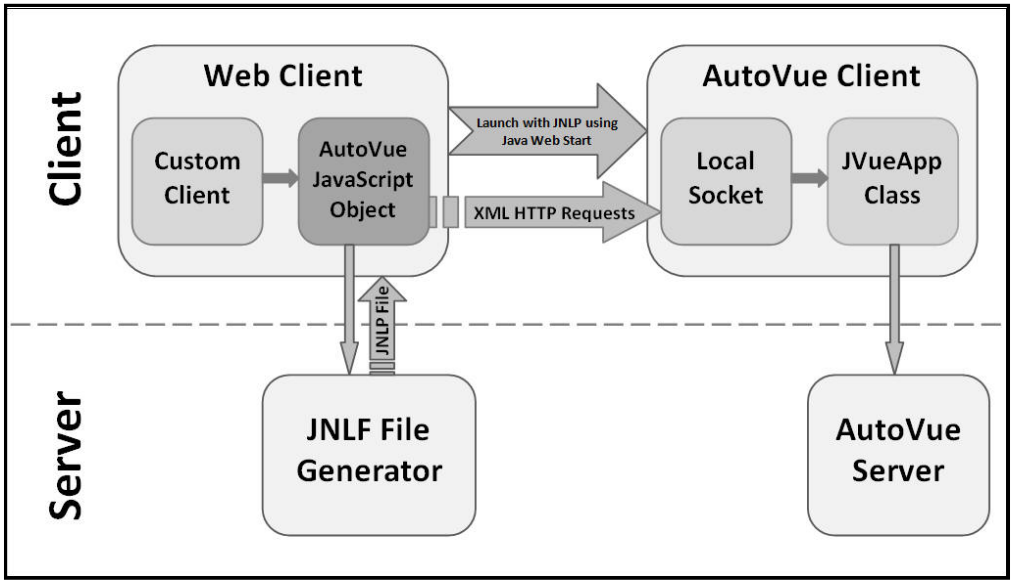
6

Architecture

AutoVue Client is a Java application that can be started through *Java Web Start* framework. This framework requires a JNLP file to start the application. An integration solution requires that the server generate a JNLP file to be used by Java Web Start framework to launch AutoVue client. The servlet **– VueJNLPServlet,** provided with AutoVue is designed to generate the required JNLP file.

Note: For more information about the required *VueJNLPServlet*, JNLP file specifications and its generation, refer to the "Deploying JNLP Components" section of the *Oracle AutoVue Client/Server Deployment Installation and Configuration Guide*.

Figure 6–1 Architecture



AutoVue client supports a scripting API and starts a socket listening to XML HTTP requests invoking this API. These requests are wrapped into a JavaScript Object named **AutoVue** and implemented in the file *autovue.js*. This object is designed to simplify the integration of AutoVue into a WEB context and provides a JavaScript method for each scripting API supported by AutoVue.

Note: If you want to send XML HTTP requests using your own approach or scripting language, then look at the methods **initScriptService** and **processScriptMethod** of the file *autovue.js*. These methods show you how to build these XML HTTP requests.

A typical usage scenario of AutoVue JavaScript API is as follows:

1. Include the source of AutoVue JavaScript API into an HTML page.
2. Instantiate an AutoVue JavaScript Object.
3. Invoke the **start** API of the AutoVue JavaScript Object to launch AutoVue client.
4. Invoke the public methods of the AutoVue JavaScript Object to interact with AutoVue.

7

AutoVue Client Launch

This chapter provides the procedure of how to start AutoVue Client using the new Java Web Start technology.

7.1 AutoVue Client Launch from Java Web Start

In order to launch AutoVue Client using the Java Web Start technology, do the following:

1. Include AutoVue JavaScript API
2. Instantiate an AutoVue JavaScript Object
3. Start AutoVue Client

7.1.1 Include AutoVue JavaScript API

The first step would be to include the source of AutoVue JavaScript API into an HTML page as shown in Example 7–1:

Example 7–1 Code to include AutoVue JavaScript API

```
<script type="text/javascript" src="autovue.js"></script>
```

7.1.2 Instantiate an AutoVue JavaScript Object

Then, you must instantiate an AutoVue Object into a JavaScript block within your HTML code as shown in Example 7–2:

Example 7–2 Code to instantiate AutoVue JavaScript Object

```
<script>
   var myAvApp = new AutoVue(JNLP_HOST, CODEBASE_HOST, CLIENT_PORTS,
                             INIT_PARAMS,   ENCRYPT_COOKIES, VERBOSITY,
                             STARTUP_DELAY)
</script>
```

The parameters required by AutoVue JavaScript Object Constructor are described in Table 7–1.

Table 7–1 Parameters required by AutoVue JavaScript Object Constructor

Parameter	Description	Default Value
JNLP_HOST	Specifies the URL on your Web/application server, to a host returning the JNLP File required by Java Web Start to run AutoVue client.	No default. This parameter is required

Table 7–1 (Cont.) Parameters required by AutoVue JavaScript Object Constructor

Parameter	Description	Default Value
CODEBASE_HOST	Specifies the location URL of AutoVue client files (**jvue.jar**, **jogl.jar**, **gluegen-rt.jar** and **jsonrpc.jar**) on your Web/application server.	No default. This parameter is required
CLIENT_PORTS	Specify a list of *localhost* ports for communication between the browser and AutoVue client. The expected format is a vector of port values or port intervals. **Example:[2345, [7500, 7510], [8500, 8510], 8888]**	No default. This parameter is required
INIT_PARAMS	Specify the client parameters to pass at the start-up of AutoVue client. The expected format is a *JSON* format of an object where the parameter/value fields are the names/values pairs of AutoVue client parameters. **Example:{'JVUESERVER': 'http://AutoVueServer:ServletPort /servlet/VueServlet','VERBOSE':'debug'}** **Note:** For a complete list of the client parameters, refer to the table H-1 in "AutoVue Client parameters" section of the *Oracle AutoVue Installation and Configuration Guide*.	`null`
ENCRYPT_ COOKIES	Toggle On/Off the encryption of the cookies passed from the browser to AutoVue Client, by the JNLP file generator, on Server side (typically; *VueJNLPServlet*). When the parameter is set to "true", then you must provide encryption key-pair using the JavaScript method **setEncryptionKeyPair**, otherwise; for security reasons, the cookies won't be sent: **myAvApp.setEncryptionKeyPair(public_key, private_key)** The public and private key values above are expected to be encoded using base64 and serialized into HEX format. The servlet **VueKeyPairServlet** provided with AutoVue produces them into this format. If you decide to use it then you must include its URL in the HTML page instantiating AutoVue object; as illustrated in the sample *av_ jnlp.html*. **<script type="text/javascript" src="graphics/VueKeyPairServlet"></script>** **Note:** For more information about the cookies encryption and *VueKeyPariServlet*, refer to "Deploying JNLP Components" section of the *Oracle AutoVue Installation and Configuration Guide*.	`true`
VERBOSITY	Specify how the browser should output error messages related to its connection with AutoVue client. The expected value should be one of the following: **0:** No output **1:** Output connection errors on the browser console **2:** Output connection errors as alerts to the user **3:** Output connection errors on the browser console and also as alerts	`1`

Table 7–1 (Cont.) Parameters required by AutoVue JavaScript Object Constructor

Parameter	Description	Default Value
STARTUP_DELAY	The start-up process can take some time to complete since the java classes (jars) have to be downloaded to the client machine and the browser may prompt the user before starting any download. At the same time, AutoVue JavaScript Object tries to establish communication with AutoVue client to detect when it is ready to handle scripting calls. This parameter specifies the required delay before assuming a start-up failure of AutoVue client.	`30`

7.1.3 Start AutoVue Client

In order to start an AutoVue client, you need to invoke the start API of AutoVue JavaScript Object as shown in Example 7–3.

Example 7–3 Code to start AutoVue Client

```
<script>
    myAvApp.start(onInit, onFail, user_data)
</script>
```

This API performs the following actions:

1. Connects to the JNLP file generator used to start AutoVue through Java Web Start, given by its URL in the argument *JNLP_HOST* of AutoVue JavaScript Object Constructor.
2. Sends the client parameters available or required at this stage (Ticket, Ports and Cookies). Then, it establishes the communication with AutoVue sending the rest of initialization parameters to complete the initialization stage.

The method takes the following optional parameters:

- **user_data:** Custom object that will be sent within the arguments of the *onFail* callback.
- **onInit:** JavaScript Callback method invoked when the custom client connects to AutoVue and the scripting API is ready for use.
- **onFail:** JavaScript Callback method invoked when the custom client fails to connect to AutoVue. It must follow the prototype above:

  ```
  function onFail(xmlhttp_request, error_msg, user_data)
  ```

  where:

  - *xmlhttp_request* is the last XML HTTP request object used to communicate with AutoVue.
  - *error_msg* is a text string describing the error preventing the connection to AutoVue.
  - *user_data* Custom object sent among the argument of this **start** API.

8

AutoVue Advanced Scripting

This chapter discusses the public scripting API that has been provided within the interface of the AutoVue JavaScript Object to integrate with the AutoVue client in dynamic Web pages.

8.1 Advanced Scripting

When integrating the AutoVue client in dynamic Web pages, a public scripting API is provided within the interface of the AutoVue JavaScript Object. Most of the API methods (except **start**, **connect**, **setEventListener** and **closeAutoVue**) do take a parameter called **frameID** which is set by default to *null*. This parameter is provided by the caller to identify the AutoVue frame that he wants to invoke the API on. When this parameter is null, the API will be invoked on the main frame. However, if it has another value, it will be invoked on the frame that was created initially with the given *frameID*. If such frame was never created before (or closed in between), then it will be created dynamically and the API invoked on it. For example, if the caller wants to create a secondary empty frame with a *frameID* "foo", then he can do it making the following call:

```
myAvApp.setFile(null, 'foo');
```

Afterwards, the caller can invoke any API on this frame using the identifier "foo" for the *frameID* parameter:

Table 8–1 Methods of AutoVue Advanced Scripting

Method	Description
connect(onInit, onFail, user_data)	Connects to AutoVue Client. It assumes that AutoVue start-up process was already launched and waiting for the custom client connection to complete the initialization stage. The method establishes the communication with AutoVue sending the rest of initialization parameters to complete the initialization stage. It is called for instance by the "start" method above after launching AutoVue start-up process through Java Web Start. To be authenticated, the browser must use the same AutoVue JavaScript Object to connect to AutoVue as the one used to start it, otherwise; the connection will be rejected by AutoVue to prevent illicit communication cross independent AutoVue sessions. The method takes the same optional parameters as the start method.
setFile(file, frameID)	Open a file in AutoVue. It takes the following parameters: ■ **file** *URL* of the file to load. If it is null or an empty string then no action will be performed. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
setPage(page, frameID)	Switch the document to a given page. It takes the following parameters: ■ **page** Index of the page to set (Number, 1-based). ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
setGUI(guiFile, frameID)	Customize AutoVue User Interface by providing a UI configuration file. It takes the following parameters: ■ **guiFile** UI configuration file identifier AutoVue needs to run with. The GUI files must be deployed on the server side (Profiles folder). ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
openMarkup(markup, frameID)	Open a Markup. It takes the following parameters: ■ **markup** Optional, semicolon (;) separated key-value list (name1=value1; name2= value2; ...) holding the markup attributes. If not provided, AutoVue will simply start a new empty markup. **Example:** `'CSI_DocID=mmMarkupID;CSI_DocName=mmMarkupName'` ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
saveModifiedMarkups (mayCancel, frameID)	In the event that the user has modified markups since the last save, it will prompt the user whether he wants to save the markups or not. It takes the following parameters: ■ **mayCancel** Specify whether the user can cancel the operation or not. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
setCompareFile(file, frameID)	Compare a given file with the current one. It takes the following parameters: ■ **file** *URL* of the file to compare with. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
import3DFile(file, transform, frameID)	Import a 3D file in the current 3D model (DMU). It takes the following parameters: ■ **file** *URL* of the file to compare with. ■ **transform** Specify a 4x4 transformation matrix (HMatrix). The API expects a four-sized JavaScript array containing the rows of the matrix. So each entry of this array is itself expected to be four-sized array of floats. **Example:** `[[1,-1,0,0],[1,1,0,0.1],[0,0,1.414,0],[0,0,0,1]]` ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
addOverlay(file, frameID)	Overlay a file onto the current one (2D). It takes the following parameters: ■ **file** *URL* of the file to overlay. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
crossProbe(file, frameID)	Load an *EDA* file in cross-probe mode to cross-probe with the current *EDA* file. It takes the following parameters: ■ **file** *URL* of the file to cross-probe. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
invokeAction(actionClassStr, frameID)	Invoke a *VueAction*. For example, the *VueActionOptionsConfiguration* will trigger AutoVue Configuration dialog. It takes the following parameters: ■ **actionClassStr** VueAction string name. The names of the supported VueAction are listed in the **AutoVue Viewing and Configuration Guide**. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
printFile(printOptions, useDefaultPrinter, frameID)	Print the current file. It takes the following parameters: ■ **printOptions** Printing options used during printing. The API expects a JavaScript object wrapping the following fields in the following hierarchy: ■ **printer:** Name of the printer to output to ■ **forceToBlack:** Whether to apply "Force to Black" rendering in the print out (Default: *false*) ■ **pages:** Sub-object holding information about the pages to print. It has the following attributes: * **choice:** 0: All, 1: Current, 2: Range * **from:** First page to print. Used only with when *printOptions.pages.choice* = *2* (Range) * **to:** Last page to print. Used only with when *printOptions.pages.choice* = *2* (Range) ■ **scale** Sub-object holding information about the paper to use. It holds the attributes below: * **value:** Scaling type, value could be "FIT", a string "<percentage>%" indicates a scale or a string "<factor>" indicates scaling to a factor (Default: "FIT") * **units:** Scaling Units (1: *in*, 2: *mm*) ■ **paper** Sub-object holding info about the paper to use. It has the following attributes: * **choice**: See AutoVue Documentation about supported paper sizes * **orientation:** 0: Portrait, 1: Landscape, 2: Auto ■ **margins** Sub-object holding the print margins to set. It has the following attributes: * **top:***float* * **bottom:***float* * **left:***float* * **right:***float* * **units:** Units in which the margin values above are given (1: *in*, 2: *mm*)

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description

	■ **printoptions** (continued)
	■ **headers** Sub-object holding additional information about the headers to add to the output. It holds the attributes below:
	* **lh:** Left header text
	* **ch:** Center header text
	* **rh:** Right header text
	* **lf:** Left footer text
	* **cf:** Center footer text
	* **rf:** Right footer text
	The options have been used in the following example:
	`{'printer':'PrimoPDF',`
	`'forceToBlack':true,`
	`'pages':{'choice':2, 'from':2, 'to':3},`
	`'scale':{'value':'75%'},`
	`'paper':{'choice':1, 'orientation':0},`
	`'margins':{ 'top':.25, 'bottom':.21,`
	`'left':.25, 'right':.25,`
	`'units':1},`
	`'headers':{ 'lh':'Left-Header',`
	`'ch':'Center-Header',`
	`'rh':'Right-Header',`
	`'lf':'Left-Footer',`
	`'cf':'Center-Footer',`
	`'rf':'Right-Footer' } }`
	■ **useDefaultPrinter**
	Specify whether to apply directly the given Print-Options as they are or allow user to change them. If this parameter is false, AutoVue will popup the print configuration dialog allowing user to modify the printer and printing parameters.
	■ **frameID**
	User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
	Note: AutoVue will initiate the print options from INI and override the ones given by the API.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
batchPrint(fileList, printOptions, openAllMarkups, useDefaultPrinter, frameID)	Print a list of files. It takes the following parameters: ■ **fileList** List of URLs of the files to print provided into a JavaScript array. ■ **printOptions** Printing options to apply during the printing operation (same structure defined in *printFile* API). ■ **openAllMarkups** This takes a Boolean value, and determines whether to include all associated markups during the printing operation. ■ **useDefaultPrinter** Specify whether to apply directly the given Print-Options as they are or allow user to change them. If this parameter is false, AutoVue will popup the print configuration dialog allowing user to modify the printer and printing parameters. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
collaborationInit (session, frameID)	Initiates a collaboration session. It takes the following parameters: ■ **session** Property string describing collaboration session in the following format: `CSI_ClbSessionID=987654321;CSI_ClbDMS=dmsIndex;CSI_ClbSessionData=123456789;CSI_ClbSessionSubject=Subject;CSI_ClbSessionType=public|private;CSI_ClbUsers=user1,user2,x;` ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
collaborationJoin (session, frameID)	Joins a collaboration session. It takes the following parameters: ■ **session** Property string describing collaboration session in the following format: `CSI_ClbSessionID=987654321;CSI_ClbDMS=dmsIndex;CSI_ClbSessionData=123456789;` ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
convertFile(convertOptions, frameID)	Convert the current file to a given format. It takes the following parameters: ■ **convertOptions** Conversion options to use in the conversion operation. The API expects a JavaScript object wrapping the following fields in the following hierarchy: ■ **file:** Mandatory. Sub-object holding information about the conversion file. It has the following attributes: * **format:** '*PCRS_BMP*' or '*PCRS_TIF*' or '*PCVC_PDF*' * **subFormat:** Format flavour: Specific to *Tif* (Currently ignored for the others) *PCRS_TIF* => 0: Uncompressed, 1: PackBits, 2: Fax III, 3: Fax IV) * **filePath:** Path of the destination file ■ **pages:** Sub-object holding information about the pages to convert. Used in multi-page formats (*PCRS_TIF*, *PCVC_PDF*). It has the following attributes: * **choice:** 0 – All, 1– Range, 2 – Current * **from:** First page to convert. Used only when *printOptions.pages.choice = 1* (Range) * **to:** Last page to convert. Used only when *printOptions.pages.choice = 1* (Range) ■ **Output:** Sub-object holding information about conversion output settings, with the following attributes: * **colorDepth:** Color depth * **fgColor:** Foreground color (in windows *RGB*) * **stepsPerInch:** Steps per inch value. For rasters this will contain *DPI* value **Example:** `{ 'file':{ 'format': 'PCVC_PDF', 'subFormat':0, 'filePath':'C:/temp/converted.pdf'}, 'pages':{'choice':1, 'from':2, 'to':3}, 'output':{ 'colorDepth':-1, 'fgColor':0, 'stepsPerInch':1016 } }` ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
setHotSpotHandler(definitionType, definitionKey, definition, frameID, caller)	Sets the hotspot handler for the given hotspot definition. This method should be called before the file session. It will initialize the hotspots in the file of AutoVue based on external application data. It takes the following parameters: ■ **definitionType** Hotspot definition type (Native WebCGM, Text Search, Attribute search...). **Note:** See Hotspot Definition Types for more information about the definition types supported for Hotspots. ■ **definitionKey** Hotspot definition key string, used to refer to this definition later. ■ **definition** Semicolon (";") separated key-value string specifying hotspot definition parameters: `param1=value1;param2=value2,…;paramN=valueN` **Note:** See Hotspot Definition Parameters for more information about the supported parameters and values of this argument. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame. ■ **caller** Parent object of the callback function mentioned in the definition (**E.g.:** window).
performHotSpot (definitionKey, hotspotKey, action, params, frameID)	Performs a hotspot action on the given hotspot. This method should only be called during the file session when the hotspots have been already initialized. It takes the following parameters: ■ **definitionKey** Hotspot definition key string provided at the creation. ■ **hotspotKey** Hotspot property key string identifying the hotspot instance, to interpret based on the definition key. ■ **action** Action to perform on the hotspot. The supported actions are: '`highlight`', '`zoomTo`', '`zoomNext`' and '`zoomPrev`'. ■ **params** Semicolon (";") separated key-value string specifying hotspot action parameters: `param1=value1;param2=value2,Ã¢Â Â¦;paramN=valueN` **Note:** See Hotspot Actions for more information about the arguments supported for each hot spot action. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.

Table 8–1 (Cont.) Methods of AutoVue Advanced Scripting

Method	Description
getInfo(info, callback, frameID, caller)	Get the requested information asynchronously. It takes the following parameters: ■ **info** Identifies the requested info. Currently only 'Custom Properties' is supported. ■ **callback** JavaScript Callback to invoke when the requested information is ready. ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame. ■ **caller** Parent object of the callback parameter (**E.g.:** window).
setEventListener(listener, filter, caller)	Enable event notifications by registering a listener to AutoVue frame events. It takes the following parameters: ■ **listener** Name of the JavaScript Callback to invoke when an event is fired and caught by AutoVue Frame. The callback should have the following signature: `function onEvent(type, event, frameID)` where: ■ *type* is the type of the event (Model Event, or Markup Event) ■ *event* us a JavaScript Object wrapping the event information ■ *frameID* is the frame ID used initially to generate the frame issuing this event If the listener is set to null, then event notifications are disabled (Default behaviour). ■ **filter** Hotspot definition key string, used to refer to this definition later. Specifies the type of events that caller wants to receive. Here are the types supported: ■ `AutoVue.EVENTFILTER_FILE`: File Events (Page switch, Loading progress, etc) ■ `AutoVue.EVENTFILTER_MARKUP`: Markup Events (Enter/Exit Markup mode) ■ **caller** Parent object of the listener callback (**E.g.:** window).
closeDocument(frameID)	Close the current document. It takes the following parameters: ■ **frameID** User identifier of the secondary frame on which this API method will be invoked. The frame will be created dynamically if it does not exist. If this parameter is null, then the method will be invoked on the main frame.
closeAutoVue()	Close AutoVue Client exiting AutoVue JNLP process.

8.2 Applet API vs. New API

The JavaScript API provided by the Java Web Start client was based on the API that was supported by the AutoVue applet. For developers who used the applet's JavaScript capabilities, note the following differences:

- The functions **printFile**, **batchPrint** and **convertFile** were ported with trimmed options. The function **invokeAction** was ported but without the second parameter referring to sub-action. Additional options and parameters may be included in the future based on customer needs. However, the Java classes options that were earlier sent through the applet are now initialized from the INI options on AutoVue side before applying the options sent through the new API.
- The following functions were removed and merged to other functions available in the new *API*:
  - **setFileInNewWindow**: Call *setFile* and set its second argument (*bNewWindow*) to *true*. By default this argument is set to *false*.
  - **setMarkupMode**: Call *openMarkup* without arguments. Notice that even though the applet function *setMarkupMode* expects a *boolean* argument, it ignores it and always enables the markup mode.
- The following functions are not ported to the new *API*:
  - setFileThreaded
  - isPrinting
  - saveActiveMarkup
  - importMarkup
  - exportMarkup
  - getActiveVueBean
  - getClass
  - createMobilePack
  - syncMobilePack
  - getDMSInfo
  - collaborationJoin
  - collaborationEnd
  - waitForLastMethod
  - SetStatusListener

Part III

ABV Guide

The Augmented Business Visualization (ABV) is a visualization framework which provides rich and actionable visual decision making environments by connecting portions of documents to business data found in enterprise applications.

Part II contains the following chapters:

- Introduction – ABV Guide
- Hotspots
- AutoVue Hotspot API
- Hotspot Samples
- VueAction Sample
- ABV Design and Security Recommendations

> **Note:** The content in the ABV Guide is applicable to the Client/Server Deployment of AutoVue only.

9

Introduction – ABV Guide

The Augmented Business Visualization (ABV) is a visualization framework which provides rich and actionable visual decision making environments by connecting portions of documents to business data found in enterprise applications. ABV's hotspot capabilities allow you to create links between objects in AutoVue's data model and objects in an external system. With this hotspot feature, an ABV solution can be built that integrates AutoVue tightly into other applications. By clicking an area of a document in AutoVue, a visual action is triggered and/or information displays in other applications. With visual dashboards, you can expose data from enterprise systems visually by changing the hotspot color.

This document provides the technical details of the ABV architecture, ABV sample code, and guidelines on how to create visual dashboards and visual actions using ABV's hotspot capability.

Note: For a general overview of ABV and its features, refer to the *Oracle AutoVue Integration Guide*.

10 Hotspots

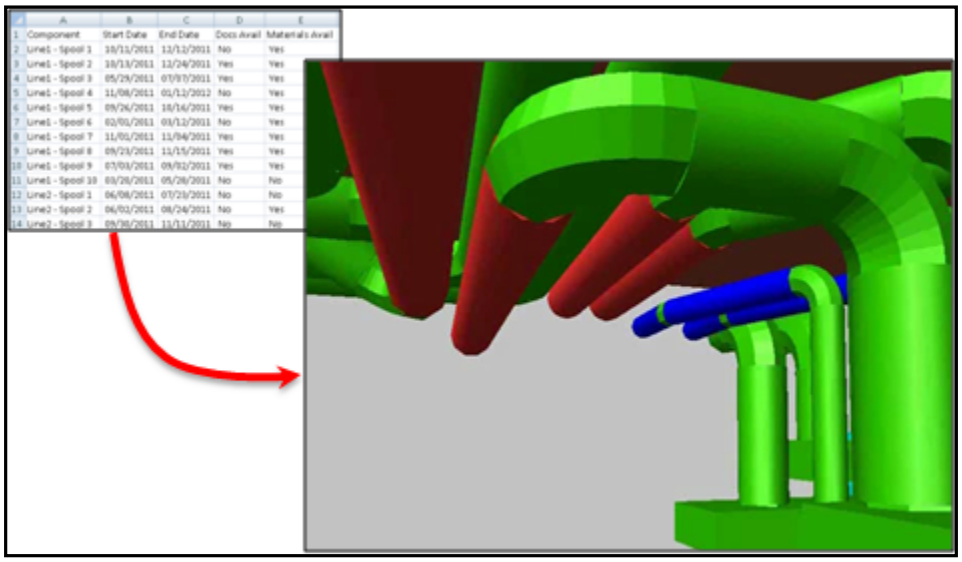
AutoVue's hotspot capabilities allow system integrators to create links between objects in AutoVue's data model and objects in an external system. With this hotspot feature, an ABV solution can be built that integrates AutoVue tightly into other applications. By clicking on an area of a document in AutoVue, a visual action is triggered and/or information displays in other applications. With visual dashboards, you can expose data from enterprise systems visually by changing the hotspot color.

This chapter provides information on how to create visual dashboards and actions, and how to define text, 3D, regional and Web CGM hotspots.

Note: For an overview of ABV's hotspot capabilities, refer to the *Oracle AutoVue Integration Guide*.

10.1 Creating a Visual Dashboard

Figure 10–1 Visual dashboard



Your enterprise application can highlight hotspots in your document based on its Enterprise Resource Planning (ERP) data, creating a visual dashboard. The visual dashboard displays structured data (enterprise application data) on top of a drawing by using color-coded hotspots based on the business data of a document.

To create a visual dashboard, you can highlight various hotspot entities in specific colors:

- Hotspots of all types can be highlighted by providing the hotspot definition key, the appropriate hotspot key, and the desired color.
- Each hotspot entity must be mapped to the appropriate color by the ABV integration.

10.2 Creating a Visual Action

A visual action is a hotspot that triggers actions in your enterprise application. These actions can include highlighting an area of the document, zooming into a component, opening a browse dialog, and so on.

Actions can be defined for the following:

- Single-Click
- Double-Click
- Selecting a named action from the RMB menu.

Action handlers can be defined to retrieve the appropriate information as follows:

- Definition key for the hotspot definition used.
- Hotspot key to identify the hotspot element being acted upon.
- The action to perform.
- Any modifiers keys (Shift, Alt, and so on) that are active when the action is started.

10.3 Hotspot Features

AutoVue supports the following user interactions with hotspots:

- Tooltips
- Triggering Actions

10.3.1 Tooltips

An active hotspot highlights to indicate that it has an action when a mouse cursor hovers over it. Additionally, a tooltip appears describing the hotspot's functionality.

In the event there are multiple layers of tooltips (markup, measurement, hotspot, and so on) that are associated to an object in the drawing, only one tooltip appears. Which tooltip appears depends on the tooltip's priority ranking in the stack of tooltips.

Note: The markup tooltip has top priority.

- Markup tooltip
- Measurement tooltip
- Hotspot tooltip
- EDA entity information tooltip
- Hyperlink tooltip

10.3.2 Triggering Actions

When a user clicks on a hotspot, a notification is fired to the ABV integration with the information identifying the clicked hotspot and the mouse action--single-click, double-click, and right mouse button (RMB) action--as well as keyboard modifiers (Ctrl, Shift, Alt).

As with tooltips, when triggering an action the following precedence rules are used:

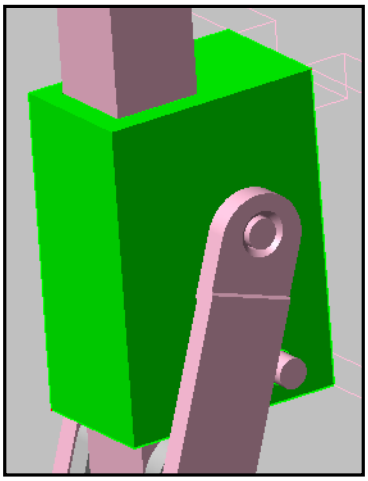
Note: The markup entity has top priority.

- Markup: Consumes the mouse action.
- Measurement: Consumes the mouse action.
- Hotspot: Notifies the external application but does not consume the mouse action and allows the subsequent layers to process the mouse clicks as well.
- Hyperlink: Does not consume the mouse action.
- EDA Entity selection, 3D Entity selection, Entity properties on double-click, and so on.

10.4 3D Hotspots

In 3D files, hotspots are defined by the attribute name. Optionally, an attribute value can be defined. If no attribute value is provided, then AutoVue identifies all parts with the attribute name as a hotspot. That is, the attribute value is used by AutoVue as a key to identify the hotspot attached to the owner part.

Figure 10–2 3D Hotspot



3D hotspots can be used to connect a 3D model to unstructured data such as order status, delivery dates, and so on. By setting up a visual dashboard in the 3D model, all this information can be pulled from an ERP and displayed in real-time.

The following sections describe how to initialize a 3D hotspot and design recommendations.

10.4.1 Defining a 3D Hotspot

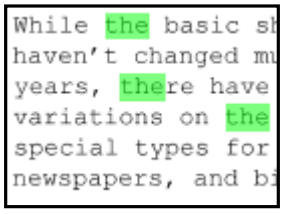
Consider the following when defining 3D hotspots:

- Hotspots are not supported on 3D PMI entities.
- 3D hotspot definitions cannot contain regular expressions in attribute names or values. Additionally, leading or trailing spaces are not permitted in attribute names/values and should exactly match the attribute names/values in the model.
- AutoVue supports attribute names/values that contain a semi-colon (;). You must precede the semi-colon with a backslash (\).
- Internal attributes that AutoVue displays in 3D models (for example, Mesh Resolution, Transparency, and Layers) should not be used when defining hotspots.
- To prevent conflicts in highlight color, it is recommended to use the Bounding Box Highlight for a 3D selection (default AutoVue setting) instead of the Entity Highlight.
- If a hotspot is defined with density as an attribute, then the specified density value must be the same value saved in the native file without measurement units.
- It is not recommended to define hotspots with attributes that the user can modify after the model loads (for example, Color, Transparency, Display/Render Mode, Visibility, Highlight Color, and Bounding Box Color). If these attributes are used and changed by the user during the file session, then the hotspots may behave inconsistently.

10.5 Text Hotspots in 2D and EDA Documents

Text hotspots are supported in 2D and EDA documents. They are based on regular expressions filtering graphical text strings based on AutoVue's text search. You can use regular expressions in the hotspot definition.

Figure 10–3 Text Hotspot



Text hotspot can be used to trigger actions such as Create Work Order or Open Detailed Parts Diagram from assets in piping and instrumentation diagrams. These hotspots can be clicked to retrieve and display asset information such as failure and repair history, working status, and so on.

The following sections describe how to define a text-based hotspot, what types of text and file formats are supported, and design recommendations.

10.5.1 Defining a Text Hotspot

You must use regular expressions in the hotspot definition in order to search for text in the document. Since AutoVue uses the Java library, it relies on Java's regular expression guidelines. For more information, refer to the following Java regular expression guidelines:

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html
http://docs.oracle.com/javase/tutorial/essential/regex/index.html

Consider the following when defining text hotspots:

- Text hotspot support is not available for raster formats, archive formats, Microsoft Word, Excel, RTF, and Outlook formats.

  Note: Support for Outlook format is deprecated in Release 21.0.1.

- Since text hotspots can only be detected on searchable text, text stored in Windows Metafiles (WMF/EMF) cannot be used for hotspotting.
- If there are multiple occurrences of a text, then they are all handled as valid hotspots.
- To be recognized as a text hotspot, characters in a string must share the same baseline. For example, a string with a normal text and a superscript text cannot be recognized as a single hotspot as they have different baselines. Alternately, regional hotspots can be applied.
- For PDF text with large spacing between characters, it is recommended to use the ADVANCEGAP INI option. For more information, refer to Section 11.1.1, "PDF Text Hotspot."
- Strings that include curved text (curved baseline) cannot be used as a text hotspot. Alternately, regional hotspots can be applied to include the curved text.

  Note: PDF documents generated through OCR are not supported for hotspots.

10.6 Regional Hotspots

Regional hotspots can be defined in 2D, EDA, and Raster files. The hotspots can either be drawn as a box or a polygon. The dimensions/extents for the hotspots are based on the coordinates displayed in the AutoVue status bar. Optionally, a user key can be used by AutoVue as an identifying key for the hotspot. If the user key is not provided, then an empty (string) key is used.

Figure 10–4 Box Hotspot

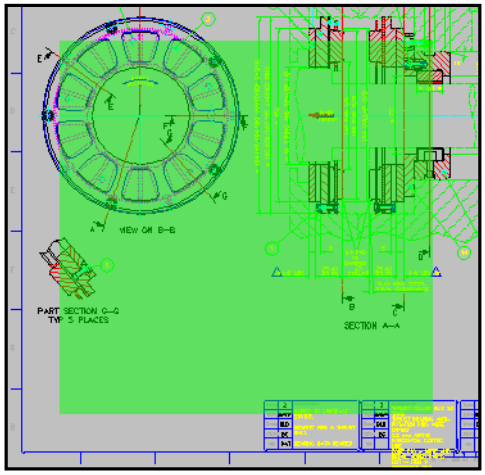
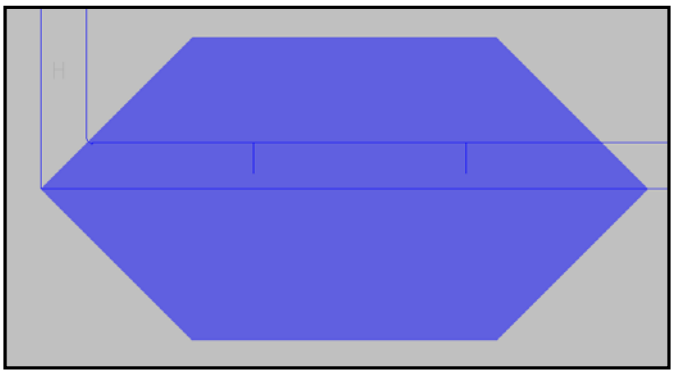


Figure 10–5 Polygon Hotspot



Consider the following when defining regional hotspots:

- Regional hotspots are not supported for archive formats, Microsoft Word, Microsoft Excel, and RTF formats.
- Vector files and raster files do not use the same World Coordinate System in AutoVue. Vector files use the bottom-left corner of the client area as the origin and the Y-axis oriented bottom-top, while the raster files use the Top-Left corner as the origin and the Y-axis oriented bottom-top. This mismatch is already exposed in AutoVue with the current user interface (UI) because the mouse position is reported in World Coordinates System on the Status Bar of the UI. Since regional hotspots are provided relative to World Coordinate System, the regional definitions need to consider this difference between raster and vector files.

10.6.1 Defining Page-Specific Regional Hotspots

When working with multi-page documents, it may be required to define page-specific regional hotspots. For example, a floor plan of interest may be on the second or third page of PDF. As a result, a new parameter allows the administrator to specify the pages where to apply the defined regional hotspot. Refer to AutoVue Hotspot API for information on the DEFINITION_PAGE parameter.

10.6.2 Defining Coordinates of a Box/Polygon

To define the coordinates of a box/polygon in a drawing, you can outline the box/polygon with a markup entity and then dump the coordinates to the regional hotspot definition. The status bar displays the world coordinates of the mouse position. The box/polygon hotspot can be manually defined to use these coordinates. For more information, refer to Polygon Hotspot for an example of using a markup entity to create a polygon hotspot.

10.6.3 Defining a Box Hotspot

A box hotspot is defined by minimum and maximum points. Where {X1, Y1} and {X2, Y2} are the coordinates of the box minimum and maximum points, respectively. Refer to AutoVue Hotspot API for information on the DEFINITION_BOX parameter.

10.6.4 Defining a Polygon Hotspot

A polygon hotspot can include an arbitrary number of sides.You can define as many sides as required for a polygon hotspot: (x1, y1), (x2, y2),..., (x*n*, y*n*). Where *n* is the number of sides of the polygon. Refer to AutoVue Hotspot API for information on the

DEFINITION_POLYGON parameter.

10.6.5 Invoking performHotspot()

To perform an action on a regional hotspot, the definition key and hotspot key parameters must be defined for `performHotspot()`. The hotspot key for regional hotspots is the user key. If the user key is not provided then an empty (string) key is used. For more information, refer to Perform an Action on a Hotspot.

10.7 Web CGM Hotspots

In Web CGM files, hotspots are defined in the native file. The hotspot information contains three attributes:

- Name
- ID
- URI

External systems can interact with these hotspots using the AutoVue ABV API with a given name. AutoVue matches the name to the ID property of the hotspot. If this fails, AutoVue matches the name to the Name property in order to highlight a specific hotspot. The definition key is always provided by the user (as with all hotspot definitions). The Web CGM hotspots include a hotspot key and definition key, and are handled in the same manner as all other hotspots.

11

AutoVue Hotspot API

The AutoVue Application Programming Interface (API) is a Java-based toolset that provides tools to modify the functionality of Oracle's AutoVue client, and allows you to create your own customized Java applications based on AutoVue API components. For more information on the AutoVue API, refer to the *Java API Guide*.

The AutoVue API's jVueApp class includes two methods that handle hotspots:

- `setHotSpotHandler()`: Defines a hotspot.
- `performHotSpot()`: Performs an action on a hotspot.

> **Note:** It is possible to extend the AutoVue client using the `VueAction()` method to implement a hotspot action. Refer to Custom VueAction for a `VueAction()` hotspot example.

11.1 Hotspot INI Options

When working with 2D, EDA, PDF and graphic documents, through the use of the Augmented Business Visualization (ABV) integration framework, you can add AutoVue's hotspot capabilities to create links between objects in AutoVue's data model and objects in an external system.

The following sections list the configuration options for hotspots provided by AutoVue:

- Section 11.1.1, "PDF Text Hotspot"
- Section 11.1.2, "PDF Text Hotspot INI Options"

11.1.1 PDF Text Hotspot

Syntax and additional information for the option described here is in section Section 11.1.2, "PDF Text Hotspot INI Options."

AutoVue provides ADVANCEGAP INI option to recognize PDF text with large spacing between characters as a single hotspot. The value of the option should be less than the maximum number of spaces between consecutive strings. That is, if the gap between two consecutive strings is less then the ADVANCEGAP value, then the strings are recognized as a single hotspot. However, if the gap between the two strings is larger than the value specified, they are not recognized as a single hotspot.

For more information on available INI options, refer to the *Oracle AutoVue Viewing Configuration Guide*.

11.1.2 PDF Text Hotspot INI Options

The following option should be placed in the [HOTSPOTS] header of the INI file.

Parameter	Description	Default
ADVANCEGAP =[integer]	Specify the maximum number of spaces between consecutive text strings.	3

11.2 Define Hotspots

```
setHotSpotHandler (final String definitionType, final String
definitionKey, final String Definition)
```

This method sets the hotspot handler for a given hotspot definition. This should typically be called before opening the file. It initializes hotspots in the files opened in AutoVue based on external application data.

Parameter	Description
definitionType	The hotspot definition type. Specify if the hotspot is a WebCGM hotspot, text search hotspot, box/polygon hotspot, or a 3D hotspot.
definitionKey	The hotspot definition key. This is the identifier for the hotspot.
definition	A string separated by semicolons specifying hotspot definition parameters. For example: name1 = value1; name2 = value2.

11.2.1 Hotspot Definition Types

Hotspot definition types supported in `setHotSpotHandler()`:

Parameter	Description
DEFINITION_TYPE_NATIVE	Native Web CGM hotspot.
DEFINITION_TYPE_TEXT	Text search hotspot.
DEFINITION_TYPE_BOX	Box hotspot.
DEFINITION_TYPE_POLYGON	Polygon hotspot.
DEFINITION_TYPE_3D_ATTRIBUTE	3D entity hotspot.

11.2.2 Hotspot Definition Parameters

The following are hotspot definition parameters supported in the key-value string parameter (definition) of the method `setHotSpotHandler()`.

11.2.2.1 Common Definition Parameters

The following are definition parameters that are common for all hotspots.

Parameters	Description
DEFINITION_TOOLTIP	The tooltip that displays when a mouse cursor hovers over a hotspot defined by the handler.
DEFINITION_ONINIT	The JavaScript method to call when page is loaded and ready to interact.
DEFINITION_FUNCTION	The JavaScript function to call when user performs an action on the hotspot.

Parameters	Description
DEFINITION_ACTIONS	Popup actions to show when user right-clicks on a hotspot.
DEFINITION_COLOR	The highlight color to use when user hovers the mouse cursor over a hotspot. Note that AutoVue parses the RGBA value as a string. Example: (R, G, B, [A]) Refer to the 3D and Box hotspots examples in Chapter 12, "Hotspot Samples" for more information. Note that integer-based colors (for example, 1627283) can are also supported.

11.2.2.2 Text Definition Parameters

The following are definition parameters for text hotspots.

Parameters	Description
DEFINITION_REGEX	Regular expression to use only in Text Search Hotspot handlers. For more information, refer to the following Java regular expression guidelines. Pattern Class: http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html Java Tutorial: http://docs.oracle.com/javase/tutorial/essential/regex/index.html
DEFINITION_MATCHCASE	Specify whether to handle case sensitivity. `Syntax: DEFINITION_MATCHCASE=[TRUE|FALSE]`
DEFINITION_SCALE	Specify the scaling bounds for text hotspots. Possible values: ■ 1: No effect. ■ 1.1: The text hotspot bounds is 10% larger. ■ 2: The text hotspot bounds is 2 times larger. ■ x: The text hotspot bounds is x times larger.

11.2.2.3 3D Definition Parameters

The following are definition parameters for 3D hotspots

Parameters	Description
DEFINITION_ATTRIB_NAME	The attribute name assigned to a 3D entity on the model.
DEFINITION_ATTRIB_VALUE	The attribute value assigned to a 3D entity on the model. (Optional) If this parameter is not specified, then all parts with an attribute of the specified name will be made into a hotspot.
DEFINITION_MATCHCASE	Whether to handle case sensitivity when searching name and value attributes assigned to 3D entities. Syntax: DEFINITION_MATCHCASE=[TRUE|FALSE]

11.2.2.4 Regional Definition Parameters

The following are definition parameters for box and polygon hotspots.

Parameters	Description
DEFINITION_BOX	Define the bounds of the rectangular box given the minimum and maximum points. Where {X1, Y1} and {X2, Y2} are the coordinates of the box minimum and maximum points. Note that the points are based on the world-coordinates of the page. Syntax: DEFINITION_BOX=#X1#Y1#X2#Y2 Example: _boxDef = "DEFINITION_BOX=#0 #0 #100 #100; DEFINITION_USER_KEY=box1; DEFINITION_PAGE=1"
DEFINITION_PAGE	Restricts box and polygon hotspot definitions to the page specified by this parameter. If no page is specified, then the hotspots apply to all pages. The following example defines the hotspot on page 2. Example: DEFINITION_PAGE = 2
DEFINITION_POLYGON	Define the bounds of the polygon as a set of points in the world coordinates in the following format: #(x_1, y_1) #(x_2, y_2)...#(x_n, y_n) where n is the number of points. Note that the minimum number of points for a polygon is 3 and that it is treated as a closed polygon (do not have to repeat the final point). Example: _defPoly = "DEFINITION_POLYGON=#(0,0) #(50, -50) #(150, -50) #(200, 0) #(150, 50) #(50, 50)" ; "DEFINITION_USER_KEY=box; DEFINITION_PAGE=1"
DEFINITION_USER_KEY	Define a user key for the box/polygon. This user key allows you to link multiple boxes with various definitions to the same external object. This is the hotspot key used for the hotspot. (Optional) If the user key is not defined, then the hotspot key is an empty string. Syntax: DEFINITION_USER_KEY=box1

11.2.3 Perform an Action on a Hotspot

```
performHotSpot (final String definitionKey, final String hotspotKey, final String
action, final String params)
```

The method performs a hotspot action on the given hotspot. This method should be called during the file session when the hotspots have been already initialized (only after the external application is notified that hotspots have been initialized in the file).

Parameters	Description
definitionKey	The hotspot definition key (the hotspot identifier) provided at creation.

Parameters	Description
hotspotKey	The hotspot property key string found based on the definition key.
action	The action to perform on the hotspot. Refer to Section 11.2.3.1, "Hotspot Actions."
params	A string separated by semicolons specifying hotspot action parameters. For example: name1 = value1; name2 = value2.

11.2.3.1 Hotspot Actions

The hotspot actions supported in `performHotSpot()` and their arguments are as follows:

Action Name	Description	Arguments
HIGHLIGHT	Perform a highlight action.	HOTSPOT_COLOR: The color for a highlight to add (RGBA Format). If this argument is not provided, the action is interpreted as a Highlight Removal.
ZOOMTO	Zoom to all hotspot instances.	
ZOOMNEXT	Zoom to the next hotspot instance.	
ZOOMPREV	Zoom to the previous hotspot instance.	

11.3 AutoVue API for ABV Integration

The ABV integration can call the AutoVue API for manipulating hotspots from the following user actions:

- Highlight (Multiple Selection, Add/Remove)
  - Text Highlight as used in text search.
  - 2D Entity Highlight for Web CGM format.
  - 3D Entity Highlight for 3D formats.
  - Regional Highlight for regional hotspots.
- Zoom to a hotspot, or the hotspots associated with a specific external object.
- Browse the hotspots associated with a specific external object using **Zoom Previous/Zoom Next**.

> **Note:** When a user selects a hotspot, all hotspots associated with the same ABV integration may be selected by using the highlight mechanism provided above.

11.4 Interactions with Hotspots from JavaScript

The following is a code prototype for a custom JavaScript function call to initialize hotspots when the file/page loads:

```
initialization_script(String definitionKey)
```

The following is a code prototype for a custom JavaScript function call when a user interacts with hotspots:

```
notification_script(String definitionKey, String hotspotKey, String action, int
keyModifiers, String properties)
```

keyModifiers describes the status of the Shift, Alt and Ctrl keys.

action may be a custom action sent during the definition of the hotspot handler (RMB actions) or one of these two predefined actions:

OnHotSpotClicked	To send when user clicks on the hotspot.
OnHotSpotDoubleClicked	To send when user double-clicks on the hotspot.

properties that could be sent to the external application notification script are as follows:

PROPERTY_ID	Property ID.
PROPERTY_NAME	Name of native WebCGM hotspots.
PROPERTY_URI	URI of native WebCGM hotspots.

12

Hotspot Samples

This chapter provide sample code on how to implement AutoVue's hotspot capability for your enterprise application.

For a detailed ABV integration example, refer to Doc ID 1472899.1 in the Oracle Customer Support knowledge base:
https://support.oracle.com/oip/faces/secure/km/DocumentDisplay.jspx?id=1472899.1

12.1 Adding a Hotspot

The following hotspot example shows how the `setHotSpotHandler()` and `performHotSpot()` methods are implemented to add hotspot capability to AutoVue. Note that this example only adds one definition, but it is possible to add multiple definitions.

1. In the html file where the AutoVue JavaScript Object is instantiated:

```
<script>
   var myAvApp = new AutoVue(JNLP_HOST, CODEBASE_HOST, CLIENT_PORTS,
   INIT_PARAMS, ENCRYPT_COOKIES, VERBOSITY, STARTUP_DELAY)
</script>
```

   Initialize the hotspots with the onInit parameter of the start API invoked to launch the client:

```
<script>
   myAvApp.start(onInit, ...);
</script>
```

   This parameter is a JavaScript callback invoked when AutoVue client is started, has been initialized and has started to listen to scripting commands.

> **Note:** If a newly added definition key already exists, then the existing definition is replaced by the new one.

```
<script>
function onAppletInit() {
    var handlerStr = "DEFINITION_REGEX=AutoVue;
    DEFINITION_TOOLTIP=AutoVue 2D Professional";
    // The following function is called once when AutoVue is ready to
    // interact with a hotspot.
    handlerStr += ";DEFINITION_ONINIT=onHotSpotInit";
    // The following function is called each time a hotspot is fired.
    handlerStr += ";DEFINITION_FUNCTION=onHotSpot;
```

```
    DEFINITION_ACTIONS=Menu1, Menu2";
    color = "(0,0,255,128)";
    handlerStr += ";DEFINITION_COLOR=" + color;

    //The following call sets up the hotspot definition.

    myAvApp.setHotSpotHandler("DEFINITION
        _TYPE_TEXT", "AV2D", handlerStr);
}
</script>
```

2. Method `onHotSpotInit()` is called for each definition when the current page is loaded and ready for hotspot interactions.

   Note that the method name should be exactly the same as the one specified in the hotspot definition `DEFINITION_ONINIT` in step 1.

```
function onHotSpotInit(hotspotDefinitionKey) {
    alert("HotSpot definition initialized: " + hotspotDefinitionKey);
}
```

3. The following `onHotSpot()` method is invoked when a hotspot is fired when the user either clicks on the hotspot or by selecting one of the Hotspot menu items defined in variable `DEFINITION_ACTION` in step 1.

```
function onHotSpot(hotspotDefinitionKey, hotspotKey, action, modifiers,
properties) {
 if (equalsIgnoreCase(action, "onHotSpotClicked")) {
        alert("User clicked on hotspot: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "onHotSpotDoubleClicked")) {
        alert("User double clicked on hotspot: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "Menu1")) {
        alert("User Peformed Menu1 action: " + hotspotKey);
 } else if (equalsIgnoreCase(action, "Menu2")) {
        alert("User Peformed Menu2 action: " + hotspotKey);
    }
}
```

   Note: The method name should be exactly the same as the one specified in the hotspot definition `DEFINITION_FUNCTION` in step 2. The `onHotSpotClicked()` and `onHotSpotDoubleClicked()` methods are predefined keys when the user clicks on the hotspot.

4. The following code performs specific actions on the clicked hotspot such as Highlight, Zoom, and so on.

```
// Highlight the "AutoVue" hotspot, "AV2D" is the definition key.
// Color : (R,G,B,A)
// myAvApp refers to the AutoVue JavaScript Object created in item 1
params = "HOTSPOT_COLOR=" + (((128 & 0xFF) << 24) | ((255 & 0xFF) << 16) |
((255 & 0xFF) << 8) | ((0 & 0xFF) << 0));
japplet.performHotSpot("AV2D", "AutoVue", "Highlight", params);

// To clear the hotspot highlight simply set the params (color) to null.
japplet.performHotSpot("AV2D", "AutoVue", "Highlight", null);

// To clear the definition highlights, set the hotspot key to null.
japplet.performHotSpot("Highlight", "AV2D", null, null);
```

```
// To clear all hotspot highlights, set the definition key to null.
japplet.performHotSpot(null, null, "Highlight", null);

// Zoom to the next "AutoVue" hotspot.
japplet.performHotSpot("AV2D", "AutoVue", "ZoomNext", null);

// Zoom to the previous "AutoVue" hotspot.
japplet.performHotSpot("AV2D", "AutoVue", "ZoomPrev", null);
```

12.2 3D Hotspot

The following example defines a 3D hotspot.

1. Get myAvApp.

   myAvApp refers to the AutoVue JavaScript Object mentioned in Advanced Scripting.

2. Define a 3D hotspot. The following code snippet defines a hotspot matching a part number in a 3D unigraphics assembly file. The sample file is included with the AutoVue Client/Server Deployment installation: <AutoVue Installation Folder>/samples/3D/Unigraphics/3DUnigraphics_iLearn-Assy.prt.

   ```
   //Turn the part with PART_NUMBER = ITEM-UG-00003 into a hotspot. You can leave
   //out the ATTRIB_VALUE if you want to highlight everything with the PART_NUMBER
   //attribute
   item00003Def = "DEFINITION_ATTRIB_NAME=PART_NUMBER; DEFINITION_ATTRIB_
   VALUE=ITEM-UG-00003;"
           + "DEFINITION_TOOLTIP=Board;"
           + "DEFINITION_ONINIT=onHotSpotInit;"
           + "DEFINITION_FUNCTION=onHotSpot;"
           + "DEFINITION_ACTIONS=Add Part, Remove Part;"
           + "DEFINITION_COLOR=(255, 0, 0)";
   ```

3. Set the 3D hotspot handler.

   ```
   myAvApp.setHotSpotHandler("DEFINITION_TYPE_3D_ATTRIBUTE", "item00003",
   item00003Def);
   ```

12.3 Box Hotspot

The following example details how to define a box hotspot.

1. Get myAvApp.

   myAvApp refers to the AutoVue JavaScript Object mentioned in Advanced Scripting.

2. Define a box hotspot. The following code snippet defines a box hotspot that encloses the Oracle logo from the PDF sample file included with the AutoVue Client/Server Deployment installation: <AutoVue Installation Folder>/samples/Desktop-Office/Basell_Autovue_Case_Study.pdf.

   Note: The box coordinates are defined by #minX #minY #maxX #maxY. Each coordinate must be preceded by a hash (#).

   ```
   oracleDef = "DEFINITION_BOX=#6.4 #0.7 #8.1 #0.4; DEFINITION_USER_KEY=oracle;"
           + "DEFINITION_TOOLTIP=www.oracle.com;"
           + "DEFINITION_ONINIT=onHotSpotInit;"
   ```

```
        + "DEFINITION_FUNCTION=onHotSpot;"
        + "DEFINITION_ACTIONS=Open Link;"
        + "DEFINITION_COLOR=(0, 0, 255, 64)";
```

3. Set the box hotspot handler.

```
myAvApp.setHotSpotHandler("DEFINITION_TYPE_BOX", "oracleBox", oracleDef);
```

12.4 Polygon Hotspot

1. Define a polygon hotspot. The code snippet provided in Example 12–1 defines a polygon hotspot that encloses the complete drawing on top right corner case from the DGN sample file included with the AutoVue Client/Server Deployment installation: <AutoVue Installation Folder>/samples/2D/MicroStation.dgn.

> **Note:** The polygon is defined by the coordinates of its points #(pt1.X,pt1.Y)#(pt2.X,pt2.Y)… #(ptN.X,ptN.Y). Each point coordinates must be preceded by a hash (#).

Example 12–1 Code Snippet that defines Polygon Hotspot

```
drawingDef="DEFINITION_POLYGON=#(666.120514,309.60045)#(928.817686,469.33385)
          + #(1115.035505,167.614443)#(852.338328,7.881023);"
          + "DEFINITION_USER_KEY=fullDrawing;"
          + "DEFINITION_TOOLTIP=The complete drawing;"
          + "DEFINITION_ONINIT=onHotSpotInit;"
          + "DEFINITION_FUNCTION=onHotSpot;"
          + "DEFINITION_ACTIONS=zoomNext;"
          + "DEFINITION_COLOR=(0,0,255,64)";
```

2. Set the polygon hotspot handler.

```
myAvApp.setHotSpotHandler("DEFINITION_TYPE_POLYGON","FullDrawingPoly",
drawingDef);
```

12.5 Text Hotspot

The example details how to define a text hotspot.

1. Get myAvApp.

   myAvApp refers to the AutoVue JavaScript Object mentioned in Advanced Scripting.

2. Define a text hotspot. The following example defines a text hotspot (regular expression) matching the AutoVue string. The PDF sample from Polygon Hotspot includes the AutoVue string in multiple locations.

```
autovueDef = "DEFINITION_REGEX=AutoVue; DEFINITION_MATCHCASE=false;"
        + "DEFINITION_TOOLTIP=AutoVue Professional;"
        + "DEFINITION_ONINIT=onHotSpotInit;"
        + "DEFINITION_FUNCTION=onHotSpot;"
        + "DEFINITION_ACTIONS=AutoVue 2D, AutoVue 3D, AutoVue EDA,
           AutoVue Electro-Mechanical;"
        + "DEFINITION_COLOR=(0, 255, 0, 128)";
```

3. Set the text hotspot handler.

```
myAvApp.setHotSpotHandler("DEFINITION_TYPE_TEXT", "AutoVue", autovueDef);
```

12.6 Text Hotspot with Visual Actions and Visual Dashboard

This example details how to define a text hotspot that utilizes the visual action and visual dashboard features.

1. Define the text hotspot.

```
//Turn strings starting with CV into a Control Valve hotspot
handlerStr  = "DEFINITION_REGEX=CV.*;";
handlerStr += "DEFINITION_MATCHCASE=false;";
handlerStr += "DEFINITION_TOOLTIP=Control Valve;";
handlerStr += "DEFINITION_ONINIT=onHotSpotInit;";
//Actions are handled by JavaScript function OnHotSpot
handlerStr += "DEFINITION_FUNCTION=onHotSpot;";
//When a hotspot is right-clicked, a menu appears with the following options:
//View Detailed Parts Diagram, Create Work Order and View Safety Information
handlerStr += "DEFINITION_ACTIONS=View Detailed Parts Diagram, Create Work
Order, View Safety Information;";
//Color : (R,G,B,A). When a mouse hovers over a hotspots, //they are
highlighted in 50% transparent blue
color = "(0,0,255,128)";
handlerStr += "DEFINITION_COLOR=" + color;
myAvApp.setHotSpotHandler("DEFINITION_TYPE_TEXT", definitionKey, handlerStr);
```

2. Create the hotspot actions.

```
//Variable hotspotKey contains the identifier for the hotspot entity that
//triggers the action when click.
function onHotSpot(defKey, hotspotKey, action, modifiers, properties) {
//If the hotspot entity is clicked, the side panel updates with information on
//the entity.
    if (equals (action, "onHotSpotClicked")) {
        updateSidePanel(defKey, hotspotKey, modifiers);
//Otherwise, if the RMB is clicked, an action can be selected from the  menu.
    } else if (equals (action, "Create Work Order")) {
        createWorkOrder(defKey, hotspotKey);
    } else if (equals (action, "View Detailed Parts Diagram")) {
        showDetailsPartsPage(defKey, hotspotKey);
    } else if (equals (action, "View Safety Information")) {
        showSafetyInfo(defKey, hotspotKey);
    }
}
```

Note that each function must know how to retrieve the appropriate information and/or to trigger the appropriate actions in the backend systems.

3. Define the highlighted hotspots for the visual dashboard. The ABV integration identifies which entities need to be highlighted and their specified color. Each entity is then passed to `performHotspot()` to highlight the entity appropriately.

```
function showHighlights{
  data = getData(); // returns array of JSON objects
  for (i=0 ; i < data.length() ; i++) {
    entity= data[i];
    myAvApp.performHotSpot("Highlight", entity.defKey, entity.hotspotKey,
entity.color);
  }
}
```

12.7 3D Hotspot with Visual Actions and Visual Dashboard

This example details how to define a 3D hotspot that utilizes the visual action and visual dashboard features.

1. Define the 3D hotspots.

```
//Turn 3D parts with ASSET_ID attribute into hotspots.
handlerStr  = "DEFINITION_ATTRIB_NAME=ASSET_ID;";
handlerStr += "DEFINITION_TOOLTIP=ASSET;";
handlerStr += "DEFINITION_ONINIT=onHotSpotInit;";
//Actions are handled by JavaScript funtion onHotSpot.
handlerStr += "DEFINITION_FUNCTION=onHotSpot;";
//When a hotspot is right-clicked, a menu appears with the following options:
//View Detailed Parts Diagram, Create Work Order and View Safety Information
handlerStr += "DEFINITION_ACTIONS=View Detailed Parts Diagram, Create Work
Order, View Safety Information;";
//Color : (R,G,B,A). When a mouse hovers over a hotspots, they are highlighted
//in 50% transparent blue
color = "(0,0,255,128)";
handlerStr += ";DEFINITION_COLOR=" + color;
myAvApp.setHotSpotHandler(
        "DEFINITION_TYPE_3D_ATTRIBUTE", definitionKey, handlerStr);
```

2. Create the 3D hotspot actions.

```
//Variable hotspotKey contains the identifier for the hotspot entity that
//triggers the action when click.
function onHotSpot(defKey, hotspotKey, action, modifiers, properties) {
//If the hotspot entity is clicked, the side panel updates with information on
//the entity.
    if (equals (action, "onHotSpotClicked")) {
       updateSidePanel(defKey, hotspotKey, modifiers);
//Otherwise, if the RMB is clicked, an action can be selected from the menu.
    } else if (equals (action, "Create Work Order")) {
        createWorkOrder(defKey, hotspotKey);
    } else if (equals (action, "View Detailed Parts Diagram")) {
        showDetailsPartsPage(defKey, hotspotKey);
    } else if (equals (action, "View Safety Information")) {
        showSafetyInfo(defKey, hotspotKey);
    }
}
```

> **Note:** Each function must know how to retrieve the appropriate information and/or to trigger the appropriate actions in the backend systems.

3. Define the highlighted hotspots for the visual dashboard. The ABV integration identifies which entities need to be highlighted and their specified color. Each entity is then passed to `performHotspot()` to highlight the entity appropriately.

```
function showHighlights{
  data = getData(); // returns array of JSON objects
  for (i=0 ; i < data.length() ; i++) {
    entity= data[i];
    myAvApp.performHotSpot("Highlight", entity.defKey, entity.hotspotKey,
    entity.color);
  }
}
```

13

VueAction Sample

The VueAction sample included with the AutoVue installation illustrates how to implement a custom hotspot action in Java. This sample is ready to be tested out of the box, but has limited application as it is not integrated with an enterprise visualization system. It is presented solely as a skeleton framework to show how hostspots can be applied.

The sample includes the following files:

Table 13–1 Files in VueAction Sample

File	Description
PartCatalogueAction.java	This is an example of how to write a custom action for AutoVue. This example illustrates implementation of an action that does more than one thing. It consists of several related sub-actions that access information about parts of a product. This action is added to two components to the AutoVue GUI: AutoVue toolbar buttons and hotspot RMB menu items. AutoVue toolbar buttons: ■ **None**: Disables mouse detection over hotspots. ■ **Description**: Enables mouse detection over hotspots and displays hotspot description as a tooltip. ■ **ID**: Enables mouse detection over hotspots and display hotspot ID as a tooltip. Hotspot RMB menu items: ■ **Order Part**: Displays a dialog that includes part information and a quantity order field. Note that this dialog does not actually retrieve any part information. It is only used to display possible RMB menu actions. ■ **Show Part Information**: Displays part name and ID.
PartListAction.java	This is an example of how to write a custom action for AutoVue. This action performs a single task and is added to the List Product Parts option of the Analysis menu: ■ List Product Parts: Lists the hotspots that user double-clicked.
PartInfo.java	This class provides product part information. It contains the catalog ID, part ID and part description.
VueActionSample.jar	JAR file for the VueAction sample.
JavaDocs	Provides detailed information on the classes included in the sample.

Table 13–1 Files in VueAction Sample

File	Description
custom.gui	Defines the custom user interface of AutoVue. It adds PartCatalogueActions to the AutoVue toolbar and Hotspot RMB menu and the PartListAction to the Analysis menu.
hotspots.txt	Contains the hotspot definitions. For information on how to define hotspots, refer to Chapter 10, "Hotspots."
customjvue.bat	Batch file that runs the sample. Note that the file illustrates how to bundle the custom action with the custom user interface.
Basell_AutoVue_Case_Study.pdf	Sample file to be used with the VueAction sample. It is located in <AutoVue Installation>\html\samples\Desktop-Office directory.
PartCatalogueAction_de.properties	German resource files.
PartCatalogueAction_en.properties	English resource files.
PartCatalogueAction_fr.properties	French resource files.

Note: For detailed information on PartCatalogueAction.java, PartInfo.java or PartListAction.java, refer to the Javadocs included with the VueAction sample.

13.1 Running the VueAction Sample

The following steps detail how to test the VueAction sample.

1. Double-click customjvue.bat.
   AutoVue launches and populates the toolbar with the **None**, **Description** and **ID** buttons.
2. To test the hotspot implementation, open the hotspot sample file, Basell_AutoVue_Case_Study.pdf.

   Note: The hotspot definition file, hotspots.txt, is configured for Basell_AutoVue_Case_Study.pdf. If you want to load another file, you must update the hotspot definitions in hotspots.txt.

3. Click **Description** to allow hotspot detection and to view tooltips. Alternately, you can click **ID** to allow hotspot detection for the hotspot ID.

   The following regular expressions are defined in hotspots.txt: *AutoVue.** and *Document*. That is, when you hover the mouse cursor over *AutoVue*, the string along with any inline text that follows it is highlighted and a *AutoVue 2D Professional* tooltip appears. For the *Document* string, the string is highlighted and the *Basell Document* tooltip appears.

13.2 Customizing the VueAction Sample

The VueAction sample can be customized to be used with a different file and with data from enterprise visualization systems. Take note that all hotspots are defined in hotspots.txt. In this file, you can specify the definition key, regular expression, whether

the text search should match case, define a tooltip, and so on. For more information on defining hotspots, refer to Hotspot Definition Parameters.

The following steps describe how to update the VueAction sample with a customized hotspots.txt:

1. Updated hotspot definitions in hotspots.txt.
2. Save hotspots.txt.
3. Extract the files from VueActionSample.jar.
4. From the extracted JAR file, replace hotspots.txt with your customized file.
5. Create a new JAR file, VueActionSample.jar.
6. Run the batch file, customjvue.bat.

14

ABV Design and Security Recommendations

This chapter provides design and security recommendations for ABV.

- Store hotspot definitions in a database and set them dynamically rather than hard-coding in the html.
- Use JSON or XML and XMLHttpRequest to pass visual dashboard information from ABV server component to ABV client components.
- Be aware that if using regional hotspots, the hotspots may need to be updated when the drawing changes
- Ensure JavaScript and Java logging is on - need to set VERBOSE=true in client parameters.
- If you are using custom GUIFILEs - ensure VueActionHotspots is included in all GUIFILES.
- Be aware that `performHotSpot()` method does not generate errors if given an invalid hotspotKey—ensure that your hotspotKey is correct if things are not working.

A

Feedback

If you have any questions or require support for AutoVue, please contact your system administrator. If the administrator is unable to resolve your issue, please contact us using the links below.

A.1 General AutoVue Information

Web Site	http://www.oracle.com/us/products/applications/autovue/index.html
Blog	http://blogs.oracle.com/enterprisevisualization/

A.2 Oracle Customer Support

Web Site	http://www.oracle.com/support/index.html

A.3 My Oracle Support AutoVue Community

Web Site	https://communities.oracle.com/portal/server.pt

A.4 Sales Inquiries

E-mail	https://www.oracle.com/corporate/contact/global.html