Reporting Developer's Guide for Siebel Communications Billing Analytics

Version: 5.1.1

Pub Date: 09/30/2006

ORACLE®

Contents

1 Reporting Engine Overview

About the Reporting Engine

The reporting engine is used for much more than just reporting. It is Siebel's Self-Service next-generation, data-mart based, powerful presentation engine. The reporting engine can present any data you can retrieve from any data sources (for example, RDBMS or CSV files).

The following list shows some possible use cases supported by the reporting engine:

- Viewing statements, invoices, etc.
- Analytic reports, such as the 10 most expensive calls
- Cost center reports (hierarchy report), such as cost summary by cost centers
- System reports like the most frequent visited users or logging analysis report
- Address book
- Email content composition.
- AR file generation

The Reporting engine offers great tools to help you implement these use cases. It uses XML to describe how you want to present a report. Then the reporting engine does the rest of the work for you, including retrieving data from data source, formatting, and then presenting the data to the end user through Velocity templates.

The reporting engine is designed to be:

- **Easy to use**: Simply create an XML file to describe the report you want to create; that's it. The reporting engine automatically generates that report for you, in variety of formats, such as HTML or CVS.
- **Easy to extend and customize**: You can easily extend the reporting engine to support any UI customization. The Reporting engine uses Velocity templates, which is a powerful MVC based presentation tool that is well suited for this purpose.
- **Easy to maintain**: The Reporting engine itself is an MVC-based, and offers the best separation of presentation logic and business logic, which greatly improves maintainability.

Reporting Engine Features

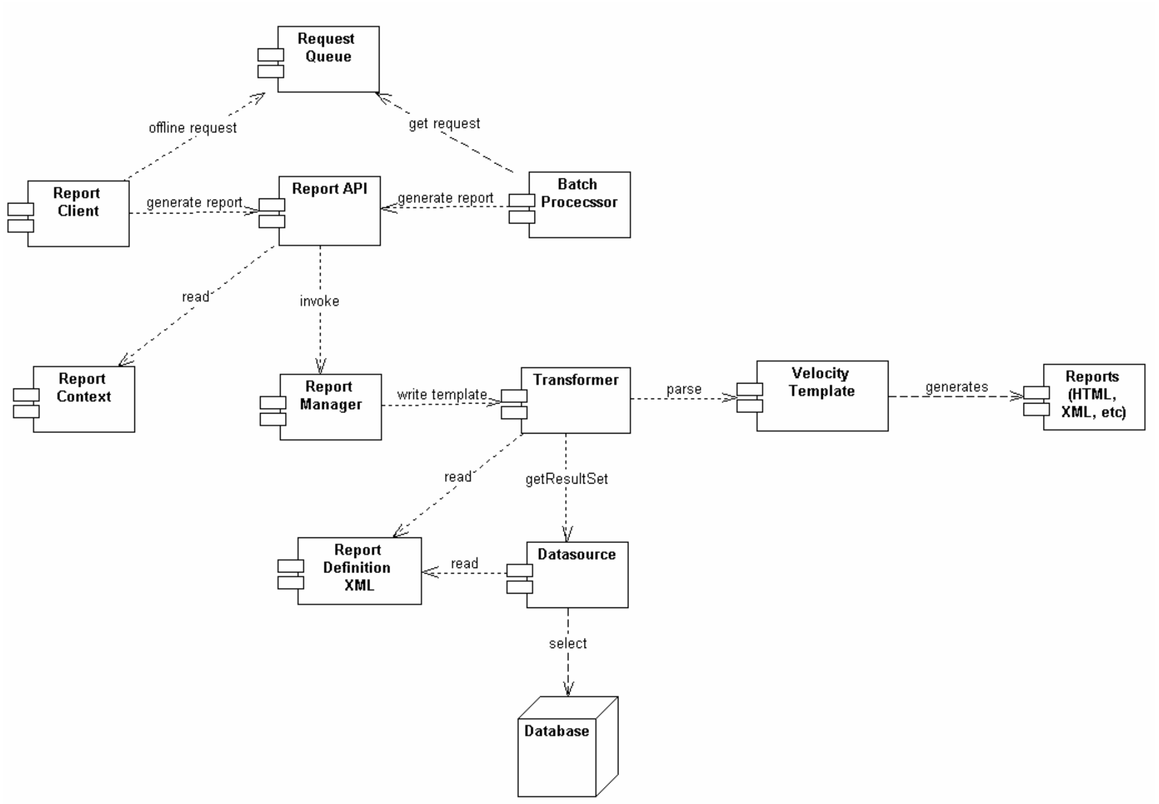
The following list shows the major features offered by the reporting engine:

- **Multiple data sources**: The reporting engine connects to multiple data sources, including SQL data source, object data source and CSV data source.
- **Prompts**: Prompts allow you to select desired data from data source.

- **Interactive sorting**: Sorting can be case sensitive or insensitive.
- **Interactive grouping:** Data is grouped by a particular column's values.
- **Calculator operations**: Summary, maximal, minimal, average and count operations are supported.
- **Charting**: This feature supports both bar chart and pie chart
- **Template**: Template-based presentation for both web-based and non-web based applications.
- **Formatting**: Support is provided for locale-based format for numeric values and dates.
- **Printer friendly view**: This feature allows you to generate a printer friendly view for printing.
- **CSV download**: CSV download allows you to download the report in CSV format.
- **Paging**: Pages through large set of data.
- **Custom report**: Custom reports allow users to create their own reports and save them for later retrieval.
- **Internationalization**: Standard Java resource bundle based internationalization; easy to understand and use.
- **Drill down and Breadcrumb links**: The Report engine offers a powerful yet simple way to drill down to different reports and drill back through breadcrumb links.
- **Seamless integration with Struts/Tiles**: The Reporting engine is not tied to a particular presentation framework, but offers excellent support for Struts and Tiles.
- **Batch report:** When it takes a long time to generate a report online, you can use the batch report feature to send a request which will be processed offline.
- **Unlimited Paging**: If the data source has too many rows and if there is a performance issue to retrieve all the rows, the reporting engine can retrieve them in batches. The paging though these batches is seamless and retrieving result set in batches is invisible to the end user.

Reporting engine architecture

The following diagram shows the reporting engine architecture based on the UML component model.



The overall reporting engine architecture follows the MVC model: the data source is the model, the report manager, transformer and report XML are the controller, and the template is the view.

Report Client: This client calls the Report API to generate reports. The client can be a Web Client, such as JSP/Servlet or Struts/Tiles, or it can be a regular standard alone application.

Report API: This is a set of APIs that the reporting client can use to generate a report. See the Javadoc for how information about how this API works.

Report Context: The Report context is used by the Report Client to exchange information with the Report Engine. It includes the information passed from the client that is used to bind the SQL query parameters and parse the templates. For example, the context may contain user session information, such as login name, current role and organization level. Or it may contain report input information, such as the date range used to generate reports. All the objects in the context can be accessed using Velocity templates.

Request Queue: This queue holds all offline batch report requests. Users can generate reports immediately, or they request that the reports be generated off-line. Off-line reports send email notification when the reports are ready. The Request Queue is a JMS queue, and holds all offline report requests.

Batch Processor: The processor gets offline report requests from the Request Queue, and sends them to the Report Engine for processing. The batch processor is a batch job that runs in Command Center.

Report Manager: The Report Manager is the central controller of report engine system. It receives reporting requests from the client, and invokes the appropriate data source and transformer to perform the desired processing.

Data source: This item represents the data source. The data source can be an SQL statement, an Object or a CSV file.

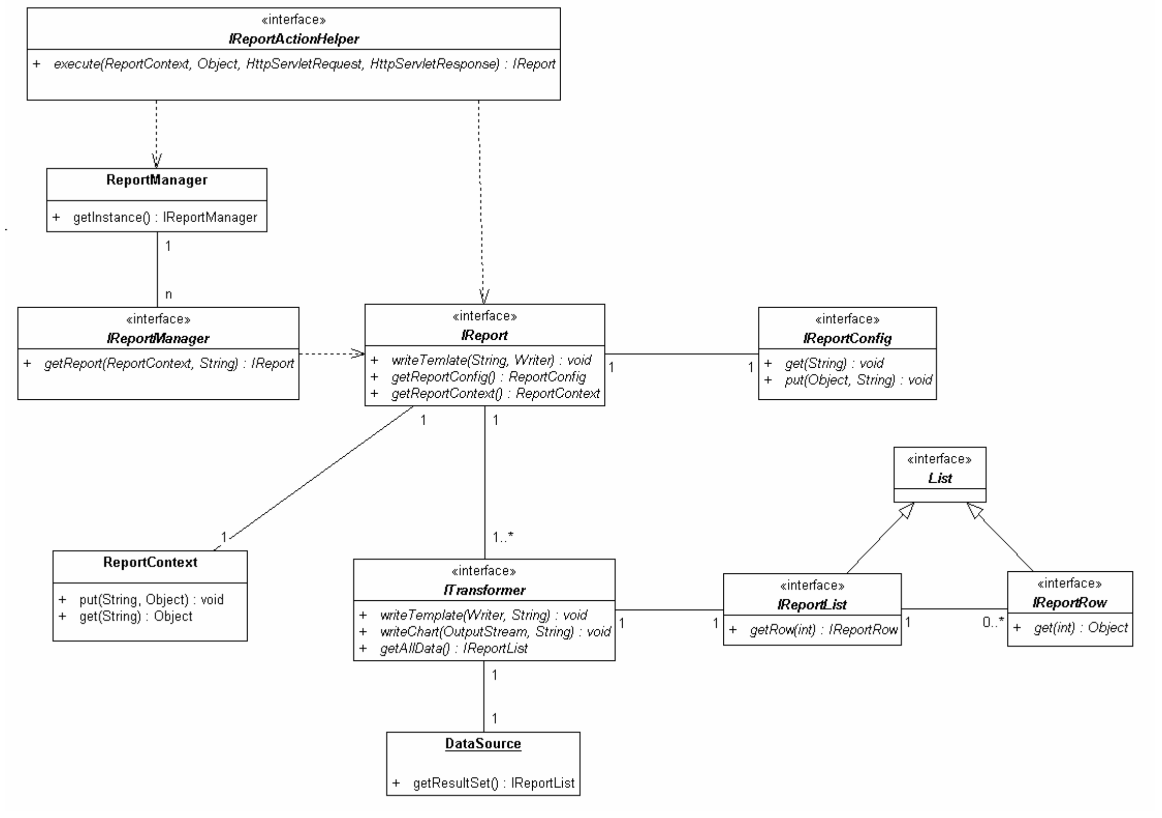
Transformer: The transformer transforms the query result from presentation, and applies a set of computations on it, including sorting, grouping, paging, aggregation (summary, average, maximal, minimal, count), and formatting. The transformer may also cache the data retrieved from data source so that the operations can be performed in the cache data (which reduces database accesses).

Velocity Template: Templates are used to generate desired report output views The templates are based on Velocity, and can generate any text reports, such as HTML or CSV. However, it is not currently possible to use Velocity to generate binary reports.

Report Definition XML: Report XML files control how reports are generated. To create your own report, create a report definition in a report XML file. You can have multiple report XML files, and each report XML file can define multiple reports.

Reporting Engine Object Model

The following diagram shows the reporting engine object model. Only the main objects are shown.


«interface»
IReportActionHelper
+ execute(ReportContext, Object, HttpServletRequest, HttpServletResponse) : IReport
ReportManager
+ getInstance() : IReportManager
1
n
«interface»
IReportManager
+ getReport(ReportContext, String) : IReport
«interface»
IReport
+ writeTemlate(String, Writer) : void
+ getReportConfig() : ReportConfig
+ getReportContext() : ReportContext
«interface»
IReportConfig
+ get(String) : void
+ put(Object, String) : void
«interface»
List
ReportContext
+ put(String, Object) : void
+ get(String) : Object
«interface»
ITransformer
+ writeTemplate(Writer, String) : void
+ writeChart(OutputStream, String) : void
+ getAllData() : IReportList
«interface»
IReportList
+ getRow(int) : IReportRow
«interface»
IReportRow
+ get(int) : Object
0..*
1..*
DataSource
+ getResultSet() : IReportList

ReportActionHelper: This class was designed to be called by the Servlet or Struts action class. It performs a number of tasks, such as parsing the request parameters, and then does the sorting, paging, etc. It returns an `IReport` object, which you can use to render a report, or manipulate further before rendering it. Though it is possible to avoid using this class by using other APIs, we strongly recommend that you use this class to reduce your customization work.

ReportManager: Use this class to get an instance of `IReportManager`.

IReportManager: This is the entry class to the reporting engine APIs. For example, to get an instance of IReport and other objects.

ReportContext: This class is actually a Map, which allows the reporting engine client to pass information to the reporting engine. For example, the binding values to SQL "?" parameters, and the objects used in Velocity templates.

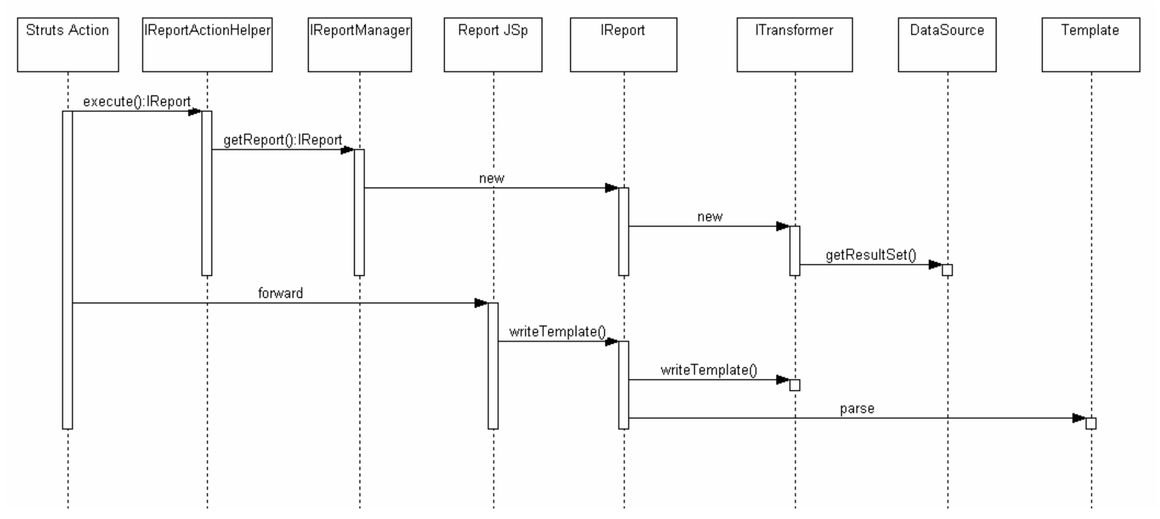
IReportConfig: This interface represents the report XML definition. For example, the SQL used to query, instructions to bind the report context objects to the SQL, instructions to format the report. There are a set of Config objects related to this class that represent the report XML elements. See the API javadoc for details.

ITransformer: This object represents the "transformer" defined in the report XML. It offers a set of APIs that manipulate the format, such as format value, write the template, etc.

DataSource: This API is not a public. It represents the "datasource" defined in the report XML, and allows you to retrieve report data from that data source.

IReportList and IReportRow: The report data retrieved from `DataSource` is represented as `IReportList`, which is a `java.util.List`. `IReportList` includes a list of `IReportRow` objects, which represents rows in report. The objects in `IReportRow` are basic Java objects, such as Integer, Double, String, Date, etc. For more details, please check Java APIs of reporting engine.

The object model of reporting engine is straightforward. The following sequence diagram shows how the reporting engine objects interact with each other to generate a report:



The Customization section describes how to write action class and report JSP pages in more detail .

2 Anatomy of Reporting Engine

The key to understanding and using the reporting engine is these three components:

- Report XML
- Template
- Reporting API

This chapter describes these components.

Reporting Package

The reporting engine is packaged as part of the Billing Analytics application. However, the reporting engine is an individual component which can be used in any other application.

Inside the application EAR file, under the XMA directory, there are two files, api-<version-number>.jar and reporting-<version-number>.jar. The api.jar includes the public APIs of reporting engine under `com.edocs.common.api.reporting`, which are the APIs you use to build your own reports. The file reporting.jar contains the implementation classes.

The reporting related components or the Billing Analytics EAR are listed in following diagram:

```
ear-tbm-tam.ear
         |
         |---------lib
         |         |
         |         |-------- velocity-1.4.jar
         |
                   |---------xma
         |         |
         |         |-------- api-1.1.1.jar
         |         |-------- reporting.1.2.1.jar
         |         |-------- reporting-ext.1.2.1.jar
         |         |-------- app-resources.jar
         |
         |---------war-tbm-b2b.war
                   |
                   |--------WEB-INF
                   |         |
                   |         |-------- struts-config-tam.xml
                   |         |--------  tiles-defs-reports.xml
                   |         |
                   |         |--------lib
                   |                   |
                   |                   |--------reporting-web-1.2.1.jar
                   |
                   |--------tam
                             |
                             |-------- _assets
                             |         |
                             |         |--------skin.css
                             |
                             |--------reporting
                                       |
                                       |--------report.jsp, *.jsp
```

The following list describes the components of the EAR:

- **Velocity-<version-number>.jar:** This archive contains the velocity template engine. Note, the property file for the Velocity engine has been updated for Billing Analytics; do NOT replace it with any other version of velocity JAR files.
- **Api-<version-number>.jar:** This archive includes the public APIs that a reporting engine client can use to access the reporting engine. They are under com.edocs.common.api.reporting. Please see the JavaDoc for details of these APIs.
- **Reporting-<version-number>.jar:** This archive includes the reporting engine implementation classes.
- **Reporting-ext-<version-number>.jar:** This archive includes the Struts `ActionForm` classes. This JAR file is stored at the EAR level instead of the WAR level in order to support the Batch Reporting functionality. The action forms must be accessible by the Event module handler, which is at the EAR level. If you are not using the Batch Reporting feature, you can package this JAR in the WAR file instead.

- **App-resources.jar:** This archive includes the resource bundles used by the Billing Analytics application. The resource bundles are loaded at the EAR level instead of the WAR level in order to localize the batch reports, which are generated offline. Currently, all reporting related resource bundles are in the com.edocs.app.reporting.resources.ApplicationResources*.properties files.
- **struts-config-tam.xml:** The Billing Analytics UI is defined as a Struts module called "tam". All the Billing Analytics related Struts configurations, such as the resource bundles, are defined in this file.
- **tiles-defs-reports.xml:** This archive includes the Tile definitions for the Billing Analytics reporting UI.
- **Reporting-web-<version-number>.jar:** This archive includes all the action classes and supporting classes that support the Billing Analytics UI. Note: the `ActionForm` classes are in the reporting-ext-<version-number>.jar because of the batch report requirement.
- **Skin.css:** This file defines the report UI related CSS.
- **Report.jsp and other jsp files:** This item consists of the Billing Analytics reporting related JSP files. The report.jsp renders the major reporting UI. You simply call `IReport.writeTemplate()` to invoke Velocity template parsing. The actual view rendering is done through Velocity templates.

In addition to the EAR file, which includes classes and JSPs, there is a set of files packaged outside the EAR. These files are Velocity template files and report XML files. For example, by default, on Windows, Billing Analytics is installed into C:/Siebel/CBA. This directory, C:/Siebel/CBA/estatement, is called the <EDX_HOME> directory, which is where the report XML files and templates are.

```
Siebel
   |
   |-------CBA
          |
          |------- estatement
                    |
                    |------- config
                    |         |
                    |         |------- rpt
                    |         |         |
                    |         |         |------- reportList.properties and *.xml, report definition XMLs
                    |         |
                    |         |------- chart
                    |                   |
                    |                   |------- *.properties files, chart property definition
                    |
                    |------- templates
                              |
                              |------- common
                                        |
                                        |-------- lib
                                        |          |
                                        |          |------- reporting_library.vm
                                        |
                                        |------- reporting
                                                  |
                                                  |------- *.vm, Velocity templates
```

The following list describes some of the files in the preceding directory structure:

- **ReportList.properties:** This file is read by the reporting engine to get the list of reporting XML files to be processed. You must register your report XML through this file.
- **Report definition XML files:** These files include report definitions. You must register them in reportList.properties. These files are loaded once during system startup, and are cached. You can reload them without re-starting the application server. See next chapter for information about how to do that.
- **Chart property files:** These files are used by KavaChart to configure the chart display properties.
- **Reporting_library.vm:** This file includes the Velocity macros defined for reporting.
- **Velocity template .vm files:** These are velocity templates used to produce reports.

reportList.properties

This file includes the list of report XML files to be loaded into report engine. You must have your report XML file defined in this file. The file format is:

```
name=xml_file_path
```

where name must be unique for each report XML and the XML file must be either under <EDX_HOME> or on the class path.

For example,

```
telco_xml=config/rpt/telco.xml
```

Which means there is a file, “config/rpt/telco.xml”, under <EDX_HOME> or on class path.

Reporting XML

The Reporting XML is central to the reporting engine. It describes how to generate a report. You can generate a good-looking report by simply writing a report XML.

The Reporting XML include two major parts: dataSource and transformer. The dataSource describes how to retrieve data from data source, and the transformer manipulates the data before sending it to the template.

There are samples of the report XML files in the <EDX_HOME/config/rpt directory. To get a complete list of all the valid report XML elements and attributes, see the report XSD file, report.xsd, under <EDX_HOME>/config/rpt.

The following sections explain some of the major features of the reporting engine and explain how to use report XML to implement them.

<reports>

This is the root element of report XML. This element can include <report>, <localizer>, <prompts> and <templates> elements. The following diagram shows that structure.

```
<reports>
      <templates>…</templates>
      <localizer>…</localizer>
      <prompts>…</prompts>
      <report>…</report>

</reports>
```

<localizer>

This element defines how the localization of the reports will be done. For details, see Internationalization/Localization on page 55 for more information.

<localizer> has the following attributes:

Name	Required	Description
`enableMessageResources`	N	Default is “true”. This attribute allows you to use the Struts `MessageResource` to look for the resource bundles for reports. This means you can use the same copy of resource bundle files defined in a struts config file without reloading another copy of it.
`defaultCode`	N	Default to “0. It enables you to define the default behavior if a resource is not found. "0" means to use the key as the default value; "1" means to use Struts notion of "???<locale>.<key>???" "-1" means throws exception.

Localizer can include <`resourceBundle`> as its child elements.

<resourceBundle>

This element specifies one resource bundle property file name to be used for report localization. See Internationalization/Localization on page 55 for more information.

For example:

```
<resourceBundle name="config/l10n/message" />
```

Which means the property file, config/l10/message_<locale>.properties under <EDX_HOME> will be used for localization.

<prompts>

The <`prompts`> element has the same format as the one defined under <`dataSource`>. However, because it is defined at the global level, it can be shared and referenced by other reports. This significantly reduces duplication of the report XML contents, and makes it easier to maintain report XML files.

See the <`prompts`> definitions under <`dataSource`> for more details.

<templates>

This element allows you define a list of global templates that can be included/parsed into other templates. For example, the paging.vm is used to generate paging UI and could be included by other templates, like report_body.vm.

For example, to define a template:

```
<templates>
    <template id="paging.vm" name="template/common/reporting/paging.vm"/>
    </templates>
```

which means there is a template named “paging.vm” and it is located in “template/common/reporting/paging.vm” under <EDX_HOME>.

Then we can include above paging.vm from another template like this:

```
#parse ($transformerConfig.getTemplateName("paging.vm"))
```

the method `transformerConfig.getTemplateName("paging.vm")` returns this template, paging.vm, from <EDX_HOME>/template/common/reporting/paging.vm.

Note, if you have a template that has the same id defined inside the transformer element, then the id in transformer takes precedence over the is in the global template list. This allows an individual transformer to use its own template. See “<transformer>” for detail.

<template>

This element defines a global template, which has following attributes:

Name	**Required**	**Description**
`id`	Y	A unique id among all the global templates. Note you can use the same id for transformer template id: in this case, the transformer template takes precedent of the global one.
`name`	Y	The full class path name of the template.

<report>

This element defines a report. A report may include zero or more <dataSource> elements, one or more <transformer>s, and zero or one of <customList>, <printList> and <downloadList>.

```
<report id=”reportId” name=”MyReport”>
        <downloadList>…</downloadList>
        <printList>…</printList>
        <customList>…</customList>
        <dataSource>…</dataSource>
        <transformer>…</transformer>
</report>
```

The <report> element has two attributes:

- **id:** The id identifies this report. All the reports defined in the report XML files in reportList.properties must have unique ids. This id must start with an alphabetic character, and can include numbers and underscores.
- **Name:** This is the name of the report. This name is used to search the report bundle to get a localized version of the report name. For example, in the Report List page, the names of reports are from this attribute.

<dataSource>

This element defines how to retrieve data from the data source.

```
<dataSource id=”” uri=”jdbcJNDI:edx.report.databasePool”>
        <query dynamic=”true”>
        </query>
        <columns>
                <column id=”” type=””/>
        </columns>
        <inputBindings>
                <inputBinding />
        </inputBindings>
</dataSource>
```

The data retrieved from the data source is represented as a List of Lists of simple Java objects, such as Strings, Date/Time/Timestamp or Numbers. We are not using a two-dimensional array because: a List of Lists gives us the potential to increase its size easily if needed, and Velocity doesn’t support accessing array elements through the [ ] operator.

The <dataSource> element has following attributes:

- **id:** A unique id identifies this data source in this report. You must define it even there is only one data source. It is not required that the id be unique across all reports. This id must start with an alphabetic character, and can include numbers and underscores.
- **uri:** A Universal Resource Identifier identifies the location of the data source. Currently we support three data sources: SQL data source, object data source and DSV data source. This example focuses on the SQL data source. For information about object data sources, see Object Data Source on page 59 and for DSV data source, see DSV Data Source on page 59.

  For an SQL data source, there are three URIs:

  - **jdbcJNDI:<dataSource_JNDI_NAME>**
    The “jdbcJNDI” indicates that this is a JDBC data source identified by its JDNI name. For example, “jdbcJDNI:edx.report.databasePool” means there is a JDNI data source named “edx.report.databasePool”.
  - **jdbcRef:<dataSource_REF_NAME>**
    The “jdbcRef” indicates that this is a JDBC data source identified by its local reference name, either defined in web.xml or ejb-jar.xml. For example, you may have an entry similar to this in web.xml:

    ```
    <resource-ref>
       <res-ref-name>jdbc/rptDataSource</res-ref-name>
       <res-type>javax.sql.DataSource</res-type>
       <res-auth>Container</res-auth>
    </resource-ref>
    ```

    With this entry, you can use following URI: “jdbcRef:jdbc/rptDataSource”. You must also resolve this local reference through weblogic.xml or another vendor specific XML file.

- **jdbcDirect:<jdbc_config_property_file_class_path>**
  The “jdbcDirect” means that there is no connection pool and reporting engine needs to make a direct JDBC connection to the database. You must specify the class path to the DB config file. For example, “jdbcDirect:config/db/jdbcConfig.properties”. For the format of the config file, look at the sample jdbcConfig.properties file coming with the product. Avoid using this URI if your application can access a connection pool.

  This element can include <query>, <inputBindings>, <prompts> and <columns> elements.

<query>

This element defines the query used to retrieve data from the data source. It applies to an SQL data source but not an object data source.

```
<query dynamic=”false” maxRows=”1000”> <![CDATA[select name, amount from
summary where user_id=? ]]></query>
```

The value for <query> is enclosed in a “CDATA” section, which can include any SQL.

The “?” in the SQL means that a variable must be resolved (bound) before the SQL can be executed. They are resolved through the <inputBindings> element.

The following table lists the attributes of this element:

Name	Required	Description
`dynamic`	N	True/false, default is “false”. This attribute indicates whether to parse this SQL as a Velocity template before execution. This allows us to use a Velocity template to generate a SQL dynamically. For information about how to write dynamically generated SQL, see Dynamic SQL on 54.
`maxRows`	N	An integer, default is 1000. This attribute indicates the maximal number of rows will be retrieved from the data source.

<inputBindings>

This element defines a list of input bindings that are used to bind the SQL variables defined in the <query> element. It has no attribute, and includes an <inputBinding> element.

<inputBinding>

This element defines a single input binding. There are two kinds of bindings: objects and prompts. The order of the <inputBinding> elements is the same as the order of the SQL variables. That means the n-th <inputBinding> is used to bind the n-th SQL variable. Object binding means binding an object or its property to an SQL variable.

For example:

```
<inputBinding object="bean" property="userId" />
```

This means there is an object called “bean” in the report context, this object is a JavaBean, and it has a property named “userId”. The value returned by bean.getUserId() will be used to bind the SQL variable. Usually, the “bean” is a Struts ActionForm object. If the object returned by the property is a Collection, then each element in the Collection will be used for binding.

```
<inputBinding object="myObejct" />
```

In this case, there is no property defined, so myObject is not assumed to be a JavaBean. If the myObject is not a Collection, then myObject is used to bind to the SQL variable directly. If the myObject is a Collection, then each element in the myObject Collection will be used to bind to the SQL variable(s) in its natural order in the collection. This latter case is very useful where the number of SQL variables is dynamic. For example, a “where name in (?...?)” clause. For more information, see Dynamic SQL on page 54.

Prompt binding is a special case of object binding. Prompt binding means that the binding object is from the user prompt, which allows you to bind the value of the prompt to a SQL variable.

```
<inputBinding object="form" property="<bean_property>" prompt="<prompt_id>" />
```

You can use a map-backed ActionForm also. For example, the ReportForm from the Billing Analytics application is a map-backed form. It has map-methods, such as getParameter(String name) and setParameter(String name, Object value). You can use this syntax in a property or prompt attribute:

```
<inputBind object="form" property="parameter(callType)" />
```

or

```
<inputBind object="form" property="parameter(callType)"
prompt="parameter(callType)"/>
```

The following table describes the attributes of <inputBinding>:

Name	Required?	Description
object	Yes	The name of the object in the report context used for binding. This object must be put into report context. In the case of prompt binding, the reporting engine automatically gets the prompt value from the prompt form, and puts this object into the report context. The ReportActionHelper class puts the value of the prompt into the context with that name. In the non-prompt case, the caller of Report engine must put this object into context.
property	No	This attribute is optional. When it appears, it means the object is a JavaBean and the value of the property of this bean is used to bind SQL variable. If this property is not there, then it means the object identified by object attribute is used for binding. Note, a map-backed property is supported, such as “parameter(callType)”.

Name	Required?	Description
`prompt`	No	This attribute indicates that this input binding is from a prompt, and the value of it must be the id of the prompt defined in the <prompts> element.

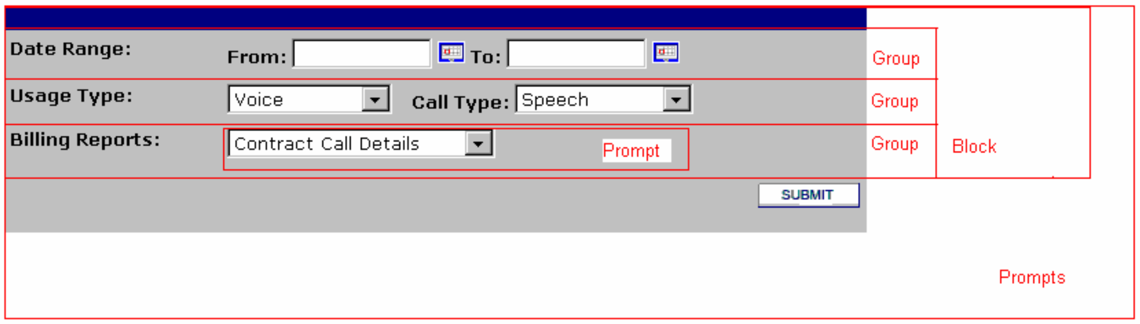
Note, the object name for the prompt form is fixed to "form" and you must use object="form" for prompt.

<prompts>

This element defines an HTML form whose input is used for data source input bindings. Each input field in the form is called a "prompt". You configure where the prompt gets its original data (from a database or from a fixed value list), and how it will be presented by the report XML. The reporting engine builds the report prompt (input) UI, which is fully customizable (it uses a template to generate the UI).

To easily control the look and feel of prompts, reporting uses a technique similar to tiles; layout.vm controls the layout format, and prompt.vm controls the prompt rendering.

The default prompt layout (layout.vm) allows you to produce a prompt layout like this:



<prompts> has a list of prompt blocks. Each block is separated by that dark blue bar at the top, and you can define a label for each blue bar. Inside each block, you can define a list of groups, where each group has a list of prompts. Each prompt group acts like `<tr>` in an HTML table, and all prompts within a prompt group display horizontally in a row. Each prompt must belong to a group. Prompts can be HTML input or a plain label. In the preceding example UI, Data range is a group with two prompts: the start date and end date. Usage type is another group that has two prompts: usage type and call type.

The `<prompts>` definition used to generate the example UI is:

```
<prompts id="prompts1" formName="reportForm" action="report.do"
      method="post" templateID="layout.vm">
       <block>
<group label="Date Range:" >
<text id="fromDate" size="12" value="1/1/2004"
    imgSrc="_assets/images/calendar.gif" label="From:"
    labelPosition="top"/>
             <text id="toDate" size="12" value="12/1/2004"
                 imgSrc="_assets/images/calendar.gif" label="  To:"
                labelPosition="top"/>
      </group>
      <group>
             <select id="parameter(usageType)" report="prompt_usageType"
    displayColumnId="usage_type_name"
    valueColumnId="usage_type_key" value="2"
    label="Usage Type:"/>
             <select id="parameter(callType)" report="prompt_callType"
    displayColumnId="call_type_name"
    valueColumnId="call_type_key" value="2"
    label="Call Type:"/>
            <image name="display" src="_assets/images/display.gif" />
       </group>
       <group label=" Billing Reports">
              <select report="prompt_reportList" value="first" name="reportId"
onChange="cleanupHiddenValues()"/>
      </group>

       </block>
</prompts>
```

You can define <prompts> under <reports> and it will be global. To refer to a global <prompts> from inside <dataSource>, use:

```
<prompts id="billingPrompts"/>
```

Which means that there is a global prompts whose id is “billingPrompts”. If the same <prompts> is used across multiple data sources, the global <prompts> helps you to maintain only one copy of it.

This element has following attributes:

Name	Required?	Description
`id`	Y	A unique id is used to identify this prompts list in this data source. Currently we only support one prompts element per data source.
`formName`	N	The name of the HTML form and default “reportForm”. It is only useful if you want to use JavaScript to manipulate the form.
`action`	Y	Action of the HTML form. Use “report.do” for the action since it is used as the default. If you change the action name defined in Struts config xml, you may need to search all your JSP pages and velocity templates to replace it.
`method`	N	Default is “post”.

Name	Required?	Description
`templateID`	Y	Specifies the layout template ID. The template must be either defined in corresponding transformer's <templates> or in the global <templates>.
`enctype`	N	Encryption type.
`onReset`	N	Name of JavaScript being called when Reset is called on the HTML form.
`onSubmit`	N	Name of JavaScript being called when Submit is called on the HTML form.

The <prompts> elements contain one or more <block> elements.

<block>

This is an optional element. If you don't define it, then you can define "group" directly under `<prompts>`, and all the groups will be put, implicitly, under a block.

You can define a label for a block and the label will be displayed in the blue bar of the above prompt diagram.

<group>

This element defines a group of prompts. This group of prompts will be displayed horizontally in one line. Different groups of prompts will be displayed vertically.

It has following attributes:

Name	Required	Description
`label`	N	The label displayed at the beginning of the each prompt group.
`description`	N	Used for rollover question mark.

There are eight types of prompts, which correspond to input types in an HTML form (except Label).

Some supported HTML forms are: Text, checkbox, select, radio, image, submit, reset and label. Image, submit, reset, label are purely for HTML form rendering and manipulation; their values are not used for report SQL input bindings. CheckBox, select, radio and text can be used for SQL input bindings.

Attributes for prompt related configuration in XML file, most of attributes are from an HTML form, others are required by the Report Engine.

`<group>` can include: `<checkBox>`, `<select>`, `<radio>`, `<text>`, `<image>`, `<label>`, `<submit>` and `<reset>`.

<select>

This element defines a select prompt. A select prompt allows you to select one or more values from a list of values. A select prompt should associate with a report whose result set is used to populate the select list. For example:

```
<select id="parameter(callType)"
       report="prompt_callType"
       displayColumnId="call_type_name"
       valueColumnId="call_type_key"
       value="2"
       label="Call Type:"/>
```

A select list requires two types of information: display values and actual values. The display values are for display purpose, and the actual values are for query purpose. For example, you can display “May 2004”, but use an internal value “5” for a query. For example:

```
<select>
     <option value=”5”>May 2004</option>
</select>
```

To render the preceding UI, get the option’s values and display names from the associated reports. The following table describes the select options:

Name	Required	Description
id	Y	Identifies this prompt in this prompts list. The id is used as the name of the input prompt in the HTML forms, which means that it determines which `ActionForm` property is used to hold this input value. In the example, the `billPeriod` property of `ActionForm` holds the value of the select box. If there is no corresponding property in the `ActionForm` (if it is a map-backed form), you can use the Parameter property (a map-backed property) to get the value into the `ActionForm`. For example, to create a prompt for call type, which is not a property of `ActionForm`: `<inputBinding object="form" property="parameter(callType)" prompt="parameter(callType)"/>` where prompt is declared as: `<select id="parameter(callType)" label="Call Type:">`. Note, when using `parameter(calltype)` as id (hence the HTML input filed name), JavaScript may not recognize the name. In that case, you may want to extend your `ActionForm` implementation to be a regular JavaBean property, which allows you to use `<select id=”callType” >`.

Name	Required	Description
`label`	N	The label of this prompt. Used for display.
`labelPosition`	N	Display label position against the prompt. Top, bottom, left, and right are supported. "top", label is on top of the prompt "bottom", label is on bottom of the prompt "left", label is at the lest of the prompt "right", label is at the right of the prompt
`size`	N	Size of the HTML input field
`report`	Y	The id of the report, whose result set will be used to populate the Select element. The report can load data from the database or it can load from a DSV data source which is useful if the data in the list is fixed.
`displayColumnId`	N	The column id of the report, whose values will be used as the display names of the <option> fields of <select> list. The first column of the report is used when `displayColumnId` is not specified.
`valueColumnId`	N	The column id of the report, whose values will be used as the actual values of the <option> fields of <select> list. The second column of the report is used when `valueColumnId` is not specified.
`value`	N	The default value for the <select> list. It can be: "first", using the first value in the `valueColumnId` column of the report "last", using the first value in the `valueColumnId` column of the report An integer N, such as "1" or "2", which indicates the N-th value in `valueColumnId` column of the report. Note the index starts from 1.
`multiple`	N	Specifies that multiple items can be selected. True and false is supported. Default is false.
`onBlur`	N	Name of JavaScript being called for `onBlur` event
`onChange`	N	Name of JavaScript being called for onChange event
`onClick`	N	Name of JavaScript being called for `onClick`event
`onFocus`	N	Name of JavaScript being called for onFocus event

The report used to generate <prompt> should meet following requirements:

- It should have two columns: one column for display, and another for prompt value. The display column id must match the `displayColumnId` attribute defined above, and the value column id must match the `valueColumnId` attributed defined above. If the report only has only one column, you can have both `displayColumnId` and `valueColumnId` point to the same column.
- The report id of the prompt report must match the report attribute defined above.
- You can format the prompt display names by using pattern attribute of column element of the report.

<checkBox>

The checkBox prompt allows you to print the prompt values in a list of check boxes. For example:

```
<checkBox id="billPeriod" label="Bill Period:"
       report="prompt_billPeriod"
       onClick="alter('onClick')"
       displayColumnId="bill_period_name"
       valueColumnId="bill_period_key"
value="last"/>
```

In the example, the bill period prompt is defined as a set of check boxes, where you can check one or more bill periods. The display names and values of bill period come from the `prompt_billPeriod` report.

The <checkBox> has the same attributes as <select>, except multiple doesn't apply (see <select> for more information). In fact, you can think of checkBox as just another view presenting the same prompt. <checkbox> is similar to a multiple-select list.

The actual data retrieved from data source for <checkbox> must be either "true" or "false".

<radio>

This prompt presents a list of radio buttons, only one of which can be selected.

```
<radio id="billPeriod" label="Bill period:"
       report="prompt_billPeriod"
       onClick="alert('onclick')"
       value="last" />
```

In the example, the bill period prompt is defined as a set of radio buttons, where you can only check one of the bill periods. The display names and values for bill period come from the `prompt_billPeriod` report.

The <radio> has the same attributes as <select>, except multiple doesn't apply. See <select> for more information. In fact, you can just think radio as another view of presenting the same prompt. <radio> is like a single-select list.

The actual data retrieved from the data source used for <radio> must be either "true" or "false", and only one can be "true".

<text>

This element allows you to define a text box and use the user-entered value as the prompt value.

```
<prompt id="billPeriod" label="Bill period:">
       <text
              report="prompt_billPeriod"
              maxLength="10"
              onBlur="alert('onBlur')"
              onChange=" alert('onChange')"
              onFocus=" alert('onFocus')"
              onSelect=" alert('onSelect')"
              size="10"
              value="06/2004"/>
</prompt>
```

In the text prompt, size attribute determines the width of the prompt.

<image>

This element allows you to define an image. For example:

```
<image name="display" src="_assets/images/display.gif" />
```

This allows you to create an image submit button. Note, this one is different from the <img> HTML tag.

Name	Required	Description
name	Y	The display name of the image.
src	Y	The image src.
align	N	"left" or "right".

<label>

This element defines text to display in the form. For example:

```
<label name="ccc_toll_lbl" value="  and  " />
```

Name	Required	Description
name	N	Not used.
Value	Y	The text to be displayed as it is on the screen.

<reset>

This element displays an HTML reset button. For example:

```
<reset name="reset" value="reset" />
```

Name	Required	Description
name	Y	Name of the reset button
value	Y	The display value of the reset button
onClick	N	Javascript to invoke.

<submit>

This element displays an HTML submit button. For example:

```
<submit name="submit" value="ok" />
```

Name	Required	Description
name	Y	Name of the submit button.
value	Y	The display value of the submit button.
onClick	N	The javascript to invoke.

<columns>

This element, under `<dataSource>`, defines the list of columns retrieved from the data source. As described previously, the data retrieved from the data source is a two-dimensional matrix with rows and columns. For an SQL query, the rows are the rows from the SQL table, and the columns are the SQL table columns. Most of the transformer operations, such as sorting, grouping and calculation, are based on the types of the columns. Only the type of the column is important, not the definition of the column. For example, you can summarize if the type is Number; it doesn't matter if the definition is Air Fee or Toll Charge. That is the primary reason to use a List of Lists of objects to present all our data.

You must define all the columns retrieved from the data source here, in the same order as the data source. For example, if you are using a SQL data source, the order of selected columns from `Select` must be the same as the order defined in the XML element. The same is true for object data sources.

<column>

This element describes the column retrieved from the data source. You must define the type of the column here. The order of <column> elements must be the same as the order of columns retrieved from the data source and for each column in the data source, you must have one of this XML element defined for it.

`<column>` includes the following attributes:

Attribute	**Required**	**Description**
`Id`	Yes	Uniquely identifies this column in the data source.
`type`	Yes	Type of column. The legal types are all simple Java object types. A column can be sorted if its type is `java.lang.Comparable`. or it can take a calculator operation (aggregation), if its type is `java.lang.Number`. `java.lang.Object`, a generic type. Avoid using this if you want to do sorting or formatting on the column. Use a more specific type, instead. `Java.lang.Double`, Indicates this is a double value, which can be sorted and aggregated. `Java.lang.Float`, Indicates this is a float value, which can be sorted and aggregated. `Java.lang.Integer`, Indicates this is an integer, which can be sorted and aggregated. `Java.lang.Long`, Indicates this is a Long value, which can be sorted and aggregated. `Java.lang.Short`, Indicates this is a Short value, which can be sorted and aggregated. `Java.lang.BigDecimal`, Indicates this is a `BigDecimal`, which can be sorted and aggregated. `Java.long.String`, Indicates a String value, which can be sorted. `Java.sql.Date`,Indicates a Date value (a Date has no time information). It can be sorted. `Java.sql.Time`, Indicates a Time value (a Time has do date information). It can be sorted. `Java.sql.Timestamp`, Indicates a Timestamp value, which includes both date and time information. It can be sorted. `Java.lang.Boolean`, Indicates a Boolean value, which can be sorted. `Java.lang.Byte`, Indicates a Byte value, which can be sorted.

Attribute	Required	Description
`default`	N	Note, this feature is only available in release 2. This attribute indicates the default value for this column ,if the value returned from data source is null. If the type or the column is: Number, then the default value is parsed as a Number string by invoking the `parseXXX` method on the corresponding java class. For example, use `Double.parseDouble()` if the it is a double. It can only include digits and decimal point. Timestamp, then you must supply the default value formatted as "yyyy-mm-dd hh:mm:ss". Date, then you must supply the default value formatted as "yyyy-mm-dd". Time, then you must supply the default value formatted as "hh:mm:ss". String, then the default value is used as it is. Boolean, then the default value can be "true" or "false".

<transformer>

This element defines a transformer for this report. A report may include zero or more transformers. Transformer is key element of the report engine; it is responsible for transforming the data retrieved from data source into a format suitable for presentation.

`<transformer>` has following attributes:

Name	Required	Description
`id`	Y	Uniquely identifies this transformer in this report. Note, you are allowed to have two transformers with same id if they are from different reports.
`datasourceId`	N	The id of the data source where the transformer gets data. Note, a transformer is not required to have a data source. If it does, then the reporting engine is usually used as a pure Template engine, and no meaningful data transformation is done inside transformer. That means that all the reporting functionality, such as sorting and paging, won't apply. For example, in telco.xml, the transformer with report_header.vm defined has no data source.
`pageSize`	N	This attribute enables paging and defines the number of rows that will be displayed in one page. All the data will be presented in one page if this attribute is not specified.

<columns>

This element defines a list of columns for the transformer. You are not required to define a column in the transformer for each column in the data source. The order of columns in the transformer do not need to match the order of the columns in the data source. However, following those two rules will make your code easier to maintain..

This XML element has no attribute and contains <column> elements.

<column>

This XML element defines a column for the transformer. The transformer will render the columns in a table format. This is one of the most important XML elements.

```
<column
       id="myColumnId"
       name="Column Name"
       isHidden="false"
       sortable="true"
       defaultSort=”true”
       caseInsensitiveSort=”true”
       pattern="MM/dd/yyyy"
       link="report.do?reportId=myReport&#x26;parameter(myColumnId)=$col"
       localize=”true”
/>
```

The following table lists all the attributes for the <column> xml element:

Name	**Required**	**Description**
`id`	Y	This id must match one of the ids defined in the data source.
`name`	Y	The name of the column. The name will be localized and then presented as the table column name.
`isHidden`	N	True/false; defaults to false. This attribute indicates whether this column is visible or not.
`sortable`	N	True/false; defaults to false. This attribute defines whether this column is sort-able. If true, the template will generate a URL link for this column.
`defaultSort`	N	True/false; defaults to false. This attribute defines whether you want this column be sorted when the report is generated.
`caseInseisitiveSort`	N	True/false; defaults to false. This attributes defines whether you want a case-insensitive sort when the column type is `java.lang.String`.

Name	Required	Description
`pattern`	N	This attribute defines the format pattern of the column's values. If the column type is: `java.sql.Date/Time/Timestamp`, you can use any `java.text.SimpleDateFormat` pattern, for example "MM/dd yyyy". See the JDK documentation for more information. Number (Double/Float, etc), you can use any `java.text.NumberFormat` patter. For example, "¤#,##0.00", where "¤" is the locale-specific currency sign. See the JDK documentation for more information. Number, but you want to format it as a duration, you can use a pattern such as: `duration2time:<unit>:<duration_pattern>`, where `duration2time` is as defined, "<unit>" is the unit of the column type, which can be "h" for hour, "m" for minute, "s" for second and "S" for millisecond. The `<duration_pattern>` can be "[h]h<separator>[m]m<separator>[s]s". The char enclosed in "[]" is optional, and you can use any separator such as " " or "/" or ":". For example, if the column value is 1.2 and the unit is minute with a pattern as "duration2time:m:hh:mm:ss" will be formatted as "00:01:12" and a pattern as "duration2time:m:h:m:s" will be formatted as "0:1:12"
`localize`	N	Note, this feature is not completely implemented in Release 1 but will be completed in release 2. True/false and default to false. This attribute let you to localize the column value retrieved from the data source if it is true: in this case, the column value is used as the key to look up the resource bundle and the localized value is presented instead.
`link`	N	This attributes allows you to define a drilldown link, which can also being defined as a <link> element. See below for <link>.
`enableDrillUp`	N	This attribute allows you to define a breadcrumb link. See section Drilldown and Breadcrumb Link for detail.

<link>

This element allows you to define a drilldown link, which can also be defined as an attribute of the <column> element. The benefit of using it as an attribute is that you can wrap the content in CDATA without escaping the special characters.

<templates>

This element includes a list of template elements. It has no attributes, and includes only one element, `template`.

<template>

This element defines one template used by the transformer. A transformer can define one or more templates and each template represents a presentation view. For example, you can define one template for HTML, one for XML and another for CSV. You specify which view (template) to use to render the UI by passing the template id through `Ireport.writeTemplate()`.

```
<templates>
        <template
                id="HTML_TEMPLATE"
                name="template/common/reporting/report_body.vm"/>
</templates>
```

This element includes following attributes:

Attribute	Required	Description
id	Y	Identifies this template inside this transformer. Note, an id must only be unique to this transformer.
name	Y	The class path of the template name. Since the template is loaded by the class loader by default, this template must exist on the classpath (for example, on WEB-INF/classes directory or packaged into a JAR file). For example, if your template is under template/templ/my.vm and that is on the class path, then you should use "template/temp/my.vm" as name.
`localize`	N	True or false. True means this template is localized. There is one template for each locale, and the report engine will find the right template based on the locale. For example, the email template has a lot of static text. Therefore, define one template for each locale, and specify this attribute as true, to associate the right template for each locale.

<groups>

This element allows you to group the data retrieved from a data source into groups, where each group is presented inside a table. For example, you may want to group on all types, so that all the local calls are presented in one table, and international calls are presented in another table. Only single column grouping is supported.

You can define multiple groups. You can define one of them as default grouping, so when the data is retrieved from the data source, it will be grouped by that default grouping. Call `Itransformer.group()` in your calling program to switch to another group.

This element has no attributes, and can include the <group> element.

<group>

This XML element defines a single group. The <column> element defines the column(s) you want to group on. You can only define one column. For example:

```
<group id="group_by_type" default="true">
       <column id="type"/>
</group>
```

<group> has following attribute:

Name	Required	Description
id	Y	Defines a unique id that identifies this group in this transformer. The group id needs to only be unique among the groups defined in this transformer.
default	Optional	Its default value is “false”. This flag indicates that this group is the default one, so when data is retrieved from data source, the data will be grouped (only one group can be default). The data won’t be grouped if there is no default group defined.

<column>

This element is defined as part of the <group> element, and identifies the column where grouping will happen. It has following attributes:

Name	Required	Description
id	Y	This is the column id defined in data source. This id must match the id of the column of the data source where you want the grouping to happen.

<calculator>

This element defines a calculator for the report. The calculator can perform a set of operations, for example: summarize (subtotal), average, maximal and minimal. The operations are grouped together into an operation group. calculator contains one or more <operationGroup> elements. For example:

```
<calculator>
       <operationGroup name="Total">
              <operation type="sum" columnId="Charges" />
              <operation type="sum" columnId="taxes" />
       </operationGroup>
       <operationGroup name="Average">
              <operation type="ave" columnId="Charges" />
              <operation type="ave" columnId="taxes" />
       </operationGroup>

</calculator>
```

For the example, the reporting engine will generate a table similar to this:

Invoice Number	**Charges**	**Taxes**
12345	10.01	0.23
23456	12.11	1.03
Total	22.12	1.26
Average	11.06	0.63

<operationGroup>

This element defines a group of operations. Different operations in the group should operate on different columns, but it's not required they have the same operation types. That is, you can mix sum with avg in the same operation group.

In general, you should not define an operation on the first visible column of the table; that column will be used to display the name of the operationGroup. However, if you do need to define an operation on the first visible column, you can change the report_body.vm by replacing the operationGroup name with the operation value you define.

This element has one attribute, name:

Name	**Required**	**Description**
name	Yes	The name of this group of operations. The default template, report_body.vm, presents it as the first column of the operation row of the table.

This element can contain one or more <operation> elements.

<operation>

This element defines a single calculator operation on a single column. It has following attributes:

Name	**Required**	**Comments**
`type`	Y	The type of the operation. `summary`: finds the summary of all the values of the column identified by `columnId` attribute. avg: finds the average of all the values of the column identified by `columnId` attribute. `max`: finds the maximal value of all the values of the column `identified` by the `columnId` attribute. min: finds the minimal value of all the values of the column identified by the `columnId` attribute. `count`: finds the total number of rows. In this case, `columnId` is optional.
`columned`	Y	The id of the column that the operation will apply to.

<charts>

This element allows you to define one or more charts for a single transformer. For example:

```
<charts>
       <chart id="c1"
              type="columnApp"
              className="com.edocs.common.reporting.chart.statementColumnApp"
              style="config/chart/telco_std_r1.properties"
              chartTitle="Totals Per Invoice Numbers"
              xAxisTitle="Invoice Number"
              yAxisTitle="Total">
              <datasets>
              <dataset><column id="Total"/></dataset>
              </datasets>
              <xlabel><column id="Invoice_Number"/></xlabel>
       </chart>
       <chart id="c2"
              type="pieApp"
              style="config/chart/telco_std_r6.properties"
              chartTitle="Plan">
              <datasets>
              <dataset><column id="total"/></dataset>
              </datasets>
              <xlabel><column id="rate_plan"/></xlabel>
              <compress threshold="2" label="Other" append="true"/>
       </chart>
</charts>
```

<chart>

This element defines a single chart for this transformer. Currently we only support two kinds of chart: Bar chart and Pie chart. The data of the chart must come from the columns of the data source.

<chart> includes following attributes:

Name	**Required**	**Description**
id	Y	Uniquely identifies this chart among all the charts defined in this transformer. Note, you can use the same chart ids in different transformers.
type	Y	The type of the chart. Currently we only support two types of chart: columnApp: which means this is a vertical bar chart. Note, the name of columnApp comes from the KavaChart terminology. pieApp: which means this is a pie chart.

Name	Required	Description
className	N	This attribute is used to identify the kavaChart class representing this chart type. If you don't specify, then report engine will use the default kavaChart class or you can specify your own class if you want to extend the default kavaChart class. In the preceding example, com.edocs.common.reporting.chart.statementColumnApp is a chart class reporting engine offers.
style	N	Full class path to the name of the KavaChart properties file. KavaChart supports the feature of defining chart styles in a properties file. Follow this link to build your own kavaChart properties file: http://www.ve.com/editor/index.html.
chartTitle	N	Defines the title of the chart.
xAxisTitle	N	The title of the X-axis. This is only used for bar chart.
yAxisTitle	N	The title of the Y-axis. This is only used for bar chart.

The <chart> elements also include following two elements: <datasets> and <xlabel>.

<datasets>

This element allows you to define multiple data sets used to draw the chart. Only one dataset per chart is supported.

<dataset>

This element defines a data set used for charting. A data set should come from the column of the data source. Currently, you can only define one column for on dataset. It has no attributes and contains one element: <column>.

<column>

This element defines the column whose values will be used as the data set to KavaChart. For example, for the columnApp chart, the dataset is used for the Y-axis values. For pieApp chart, the dataset is used for the Pie's data.

This element only includes one attribute:

Name	Required	Description
id	Y	The id of the column where the chart will get its data. The type of the column must be a number.

<xlabel>

This element defines the values for the x-axis. The x-label must come from the data source column. It has no attributes, and contains one element: <column>. You can only define one column for each x-label.

<column>

This element defines the column used for the x-label. The values of the column are used for the x-axis values. This element only includes one attribute:

Name	Required	Description
id	Y	The id of the column where the chart will get its x-axis values.

<downloadList>

This element defines a list of downloads available for this report. For example, you may define an XML download and a CVS download. For each download, a download link will be generated by the template. You can define multiple downloads for one report. For example:

```
<downloadList name="Download">
        <download
        name="Download CSV"
        type="csv"
        description="CSV download"
        templateId="CSV_TEMPLATE" />
</downloadList>
```

It has only one attribute, <name>.

Name	Required	Description
name	N	The name of this downloadList. Depending on your template, you can use this name for different purposes. For example, you may build a dropdown list of downloads and use this name as the name of the dropdown list.

<download>

This element defines one download for the report. It has following attributes:

Name	Required	Description
type	Y	The type of the download. You can name any type you want, but since the type is used as the download file extension, you should choose something meaningful to the file system. For example, use “csv” for CSV download and use “xml” for XML download.

name	N	The name of the download. Depends on the template; it can either be shown as a URL link or as a dropdown list item.
description	N	Description of the download. Currently it is not used by template, but you can modify template to use it for a pop-up help window.
templateId	Y	The template Id used to generate the download of the report. It's possible that the same template id may appear in multiple transformers and so, all these templates will be parsed and appended together, in the order of the templates defined in XML.

<printList>

This element defines a list of print-friendly available for this report. Through it's possible, it is unlikely you will define more than one print-friendly. For each print friendly, a print friendly link will be generated through the template.

Here is one example:

```
<printList name="Print friendly">
       <print
              name="Print friendly"
              description="print friendly account details"
              templateId="PRINT_TEMPLATE" />
       </printList>
```

It has only one attribute, "name".

Name	**Required**	**Description**
name	N	The name of this `printList`. It is not used by current template.

<print>

This element defines one print-friendly for the report. It has following attributes:

Name	**Required**	**Description**
name	N	The name of the print-friendly. The default template renders it as a URL link.
description	N	Description of the print-friendly. Currently it is not used by template, but you can modify template to use it for a pop-up help window.
templateId	Y	The template Id used to generate the print-friendly of the report. It's possible that the same template id may appear in multiple transformers, so all these templates will be parsed and appended together, in the order of the templates defined in XML.

<customList>

This element defines a list of custom reports available for this report. Through it's possible, it is unlikely you will define more than one custom report. For each custom report, a custom report link will be generated through the template.

For example:

```
<customList name="Customize">
       <custom
              name="Customize"
              description="Create a custom report for contract call details"
              reportId="telco_cust_std_r4" />
       </customList>
```

It has only one attribute, <name>.

Name	Required	Description
name	N	The name of this customList. It is not used by current template.

<custom>

This element defines one custom report for the current report. Each custom report must be itself defined a report. This <custom> tag is used to build a link to that custom report. It has following attributes:

Name	Required	Description
name	N	The name of the custom report. The default template renders it as a URL link.
description	N	Description of the custom report. Currently it is not used by template but you can modify template to use it for a pop-up help window.
reportId	Y	The report id of the report used to define the custom report: the custom report itself is a report and you must define it as a report.

Reporting templates

All the report UIs are generated through Velocity templates. For information about how the Velocity templates work, see:

http://jakarta.apache.org/velocity/index.html

CAUTION: Billing Analytics has changed some of the default Velocity templates. The most important one is that inside “foreach” loop, the $velocityCount variable starts from 0 instead of the default 1.

Billing Analytics offers a set of example templates that generate useful UIs. These templates are very generic, are not tied to a particular application, and can be used as the base for your customization work.

The templates are all defined in <EDX_HOME>/template/common directory. The lib subdirectory includes some Velocity MACRO library files and the reporting subdirectory includes report template files.

The following table explains the libraries that are included with the report package:

Name	**Description**
`Lib/report_library.vm`	This file defines some common MACROs used by the reporting engine. You should use it as it is.

The following table explains the templates that are included with the report package:

Name	Description
`Common/report_header.vm`	This is the header part of the report. Note, this is not the header of the tiles. The tile header is usually the Navigation Tabs. The report header usually includes the report name and the download/print friendly/custom report links.
`Common/report_body.vm`	This template is used to render the table associated with the transformer. Since a report may define multiple transformers, the template may be parsed multiple times for a report.
`Common/paging.vm`	This template is used to render the paging navigation part, which has previous, forward buttons for a user to page through the report.
`Common/layout.vm`	This template is used to define the layout of the prompts of the report.
`Common/promt.vm`	This template is used to render each individual prompt of the report.
`Common/csv.vm`	This template is used to render the CSV format of a report. The current CSV format is very simple and doesn't consider the case of how to escape the special characters like ",". You must write code to handle that case.
`Common/print.vm`	This template is used to render the print friendly format of a report.
`Common/custom_report.vm`	The template is used for custom report: it displays the custom report detail and allows you to type a name for the report to save into database.

Pre-defined context variables

When you call the `IReportActionHelper.execute()` method to generate reports, the reporting engine puts a list of pre-defined context variables into the report context, which are then available to the Velocity template.

The following table lists some of the variables you may be interested. If the `overwrite` flag is Y, then you can pass a variable with the same name through `ReportContext` to overwrite the default values set by `ReportActionHelper`.

Name	Type	Over write?	Description
`form`	`Action Form`	N	This is the struts `ActionForm` object currently being processed.

Name	Type	Over write?	Description
gifDir	String	Y	The directory where the image files used by report are saved, for example, the paging arrow images. It is default to "_assets/images".
link	String	N	This is the URL link base of this page and it is equivalent to the html <base> tag. Its default value is: HttpServletRequest.getContextPath() + HttpServletRequest.getServletPath() and similar to this: http://host:port/<web-root>/report.do.
user	IUser of UMF	Y	The current user logged in. IUser is passed in as a session variable, "USER_PROFILE". If it is not in the session, we won't put it in the context. "user" is just used for query purpose and its absence won't affect the functionality of the reporting. For example, for some reason, you may not use UMF IUser and you can use your own user object. Note, some template, like report_header.vm, may expect IUser to get user name and if you don't supply IUser, the template may not display user name properly.
contact Profile	IContactProfile of UMF	Y	The contactProfile is a profile of IUser named as contact_profile. We currently use it to retrieve user's first name and last name and is currently only used in report_header.vm. Note, absence of this information won't affect the function of reporting engine.
locale	String	Y	The default value set by ReportActionHelper is the from http session: session.getAttribute("org.apache.struts.action.LOCALE"). Note this is locale put into session by Struts framework.
reportId	String	N	The report Id of current report.
transformer	ITransformer	N	You can use transformer object to do work such as formatting data etc. Note, you should never call ITransformer.writeTemplate() in the template.
reportConfig	IReportConfig	N	Represents the report configuration.
dataSourceConfig	IDataSourceConfig	N	Represents the data source configuration.
dataSource	A list of IDataSource	N	Represents the list of data source column

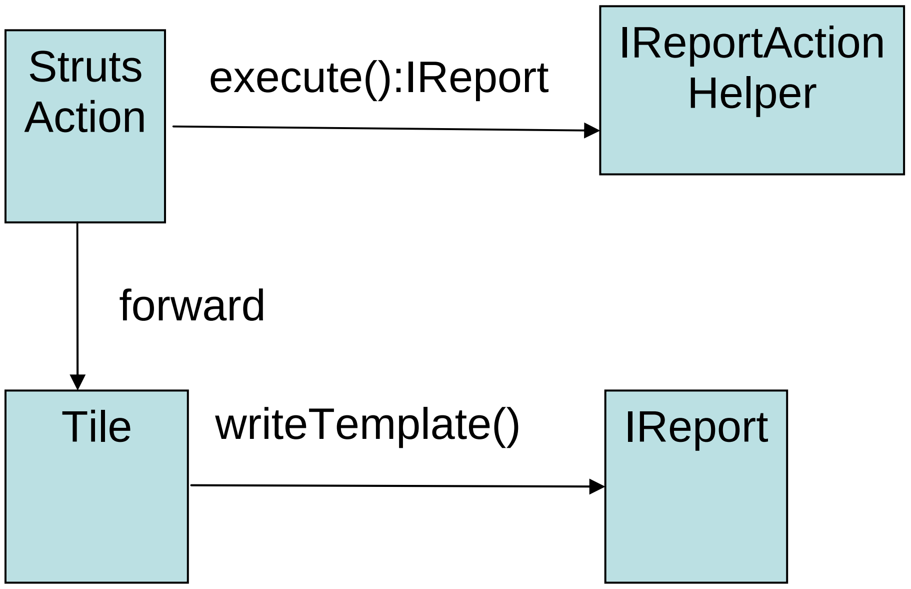
Name	Type	Over write?	Description
ColumnConfigs	ColumnConfig		configurations.
transformerConfigs	ITransformerConfig	N	Represents the transformer configuration.
transformer ColumnConfigs	A list of ITransformer ColumnConfig	N	Represents the list of transformer column configurations.
operationGroupConfigs	A list of IOperation GourpConfig	N	Represents the list of operation groups defined inside calculator for the transformer.
chartConfigs	A list of IChartConfig	N	Represents the list of chart configurations for the transformer.
templateConfigs	A list of ITemplateConfig	N	Represents the list of template configurations for the transformer.
rowlist	IReportList	N	Represents the original data retrieved from the data source. The data may be sorted and so the order may be changed. Though you cannot overwrite this variable, you can certainly change the content of the list
groupSet	Set	N	To support grouping, the transformer maintains a Map of List objects. In the case of no grouping, there is only one entry in the map, the key is the report name, and the value is the List returned from data source; In the case of grouping, the original list from the data source is re-grouped into multiple lists. Each list has the same group value, and the group value becomes the map key. This variable is looped through in report_body.vm to build the HTML table.
dataMap	Map	N	This is the map of group keys to the List as described previously.
reportContext	ReportContext	N	The ReportContext object used to generate reports. Note, you can't overwrite it, but you can change the content of it.

Name	Type	Over write?	Description
`URLEncoder`	`URLEncoder`	N	This is actually a wrapper class around `java.net.URLEncoder`, the reason we do this is that Velocity cannot invoke a static method directly through class name, and `java.net.URLEncoder` doesn't have a constructor. Use this class to encode the parameter values you passed through the URL.

Integration with Struts/Tiles

The reporting engine can be used with any presentation framework. However, since Billing Analytics is based on Struts/Tiles, the reporting engine has special extensions to help it integrate with Struts/Tiles. This section describes that integration.

The following diagram illustrates how to use the reporting engine in the context of Struts/Tiles:



Struts Action Class:

The Struts action class does following processing:

```
ReportContext ctx = new ReportContext()
ctx.put(…) //put whatever your stuff used in template
IReportActionHelper helper = ReportManager.getReportActionHelper()
IReport report = helper.execute(ctx, form, request, response); //IReport will
be in session
```

```
return mapping.findForward(" page.reports.report ");
```

It creates a `ReportContext` object which you can put your own objects into. These objects can then be used in report templates.

After that, it calls `IReprotActionHelper.execute()` method to get an `IReport` object. If this is the first time to access the report, a new `IReport` object will be created; if this is a sorting or paging operation, the `IReport` object cached in the session will be returned. In case a new `IReport` object is needed, the report data will be retrieved from the DataSource defined in the report XML of this `reportId`.

For the last action of this class, control is forwarded to the tile, `page.reports.report`, which is defined in the tiles definition file.

Tiles definition

Tiles are defined in the tiles-defs-reports.xml file in the WAR file of the EAR file.

```
<definition name=" page.reports.report "
             path="/tam/_templates/simpleLayout.jsp">
       <put name="title" value="Telco Analytics Manager" />
       <put name="header" value="/tam/_includes/header.jsp" />
       <put name="footer" value="/tam/_includes/footer.jsp" />
       <put name="actionBar" value="/tam/_includes/actionBar.jsp" />
       <put name="body"   value="/tam/reporting/report.jsp" />
</definition>
```

The key to this tile is that the body tile is report.jsp, which generates the main body of reporting UI.

Report.jsp

The report.jsp page is used to render the view. In fact, there is almost no HTML code in this page. Instead, this page just invokes the Velocity template engine to parse the templates:

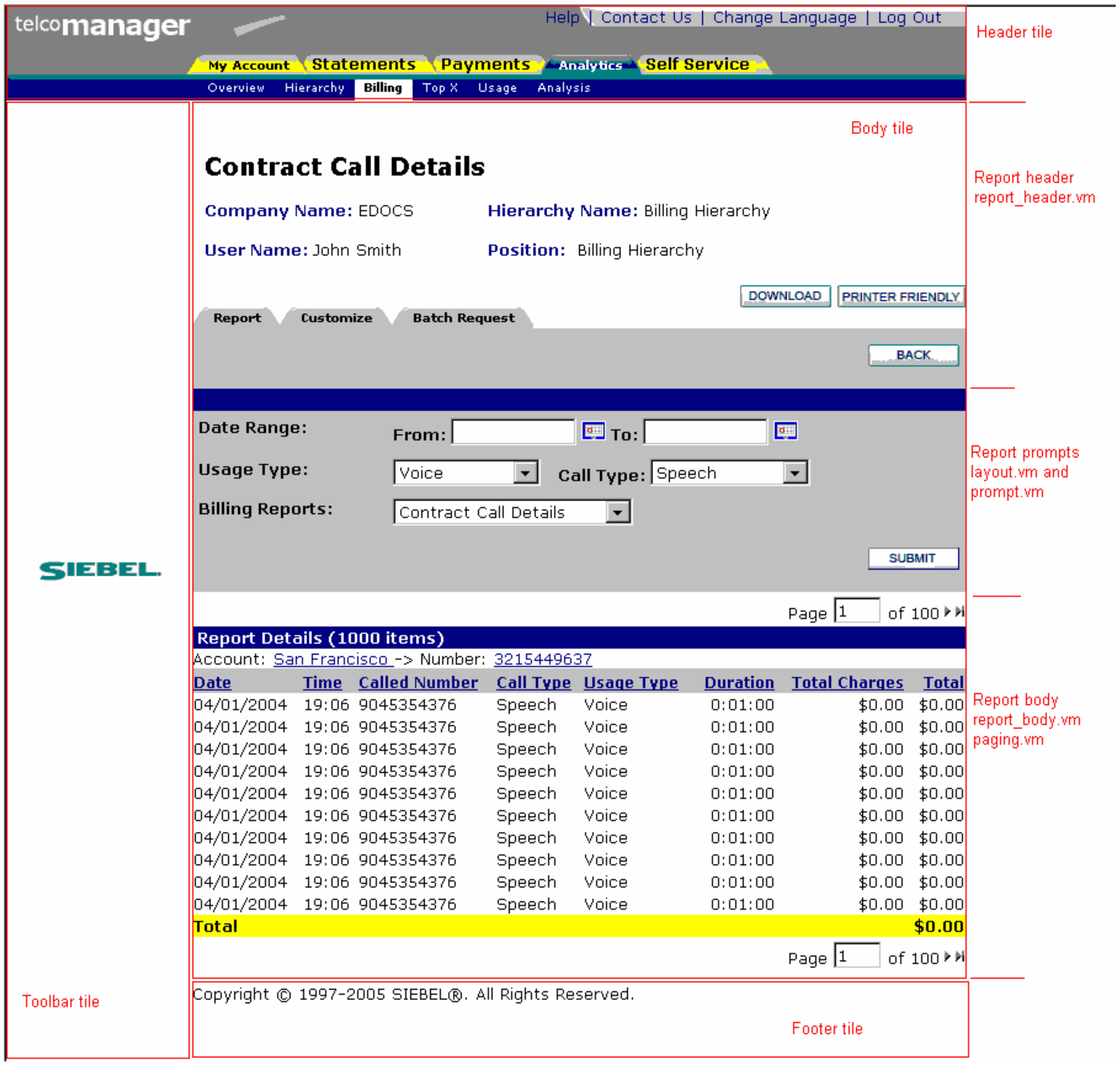
```
IReport report = (IReport)request.getSession().getAttribute(reportId);
```

```
IReport.writeTemplate(jspWriter, templateId);
//template is the one defined in report xml and default to “HTML_TEMPLATE”
```

The reporting engine goes through the Transformers defined in the report XML for this `reportId` and for each transformer, parsing the template whose ID matches `templateId`. Note a transformer will be ignored if it has no template with a matching `templateId` defined in the transformer configuration of the report XML.

The matching templates will be parsed in the same order as defined in the report XML, and the results will be written back into JSPWriter sequentially.

The following diagram illustrates the different View components of a typical Billing Analytics reporting page:



Contract Call Details

Company Name: EDOCS **Hierarchy Name:** Billing Hierarchy

User Name: John Smith **Position:** Billing Hierarchy

Date Range: From: To:

Usage Type: Voice Call Type: Speech

Billing Reports: Contract Call Details

Page 1 of 100

Report Details (1000 items)

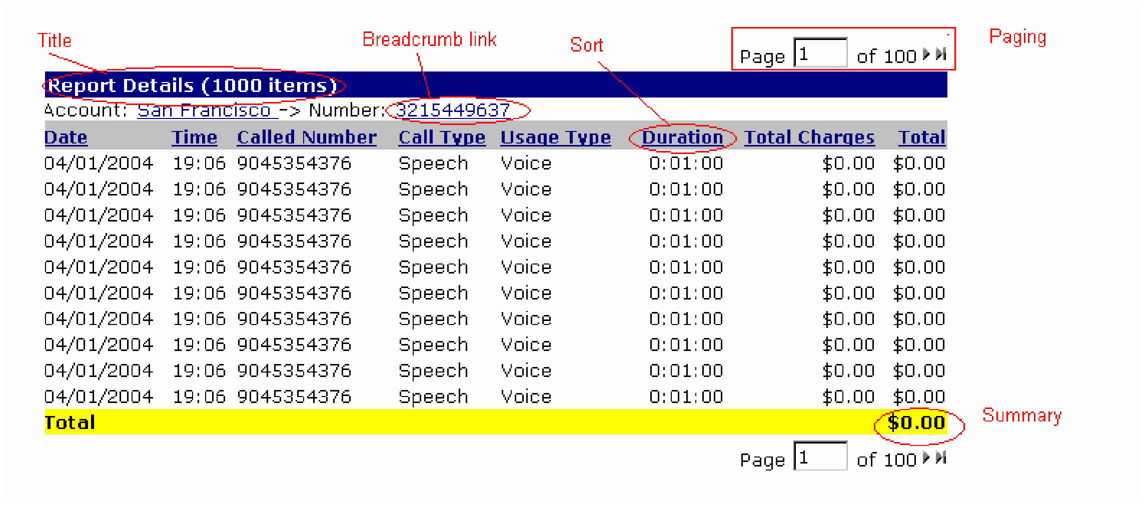
Account: San Francisco -> Number: 3215449637

Date	Time	Called Number	Call Type	Usage Type	Duration	Total Charges	Total
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
Total							**$0.00**

Page 1 of 100

Copyright © 1997-2005 SIEBEL®. All Rights Reserved.

The following diagram illustrates the details of report body:



Page 1 of 100

Report Details (1000 items)

Account: San Francisco -> Number: 3215449637

Date	Time	Called Number	Call Type	Usage Type	Duration	Total Charges	Total
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
04/01/2004	19:06	9045354376	Speech	Voice	0:01:00	$0.00	$0.00
Total							$0.00

Page 1 of 100

Reporting API

The reporting API offers an interface to interact with the reporting engine. These APIs manage common reporting features, such as sorting, grouping and paging. They also offer report clients the flexibility to customize reporting.

The reporting API is not tied to a particular presentation framework; you can use struts and tiles or servlets and JSP to access it. However, you may find that using struts and tiles is the easiest way to implement your own reporting UI, because that is the default presentation framework used for the reporting UI of Billing Analytics.

The core reporting APIs are: `ReportContext`, `IReportManager`, `IReport`, `ITransformer`, `IReportConfig` and `ReportActionHelper`. See the Reporting API JavaDoc for more information.

`ReportContext` is the carrier of information between the reporting caller and the reporting engine. `ReportManager` is a factory that gets an instance of `IReportManager`. `IReportManager` is the factory for `IReport` objects. `IReport` represents a report defined in XML. `ITransformer` represents the transformer defined inside a report in XML. `IReportConfig` represents the configuration information in XML.

For example, here is an example that shows how to generate a report:

```
ReportContext context = new ReportContext();
context.put(“form”, StrutsActionForm);
IReportManager rptmgr = ReportManager.getInstance();
IReport rpt = rptmgr.getReport(“reportId”, context);
Rpt.writeTemplate(“templateId”, Writer);
```

In the example, a Struts `ActionForm` is put into the `reportContext`, which means this object is available to the Velocity template. You can use the following syntax in the velocity template: `$form.name`, assume there is a name property in the form.

After you get an instance of `IReportManager`, call its `getReport` method to get a report. The `reported` must match the one defined in report XML. It will return an object that represents the report defined in the XML with the same `reportId`.

After you get an instance of `IReport`, it calls its `writeTemplate()` method to parse the Velocity template identified by `templateId` in the report XML, and writes the content into a Writer output. This method actually loops through all the transformers in the report and calls `transformer.writeTemplate()`. Multiple templates may be parsed, if the same template Ids appear in different transformers, then the content of the parsed templates will be appended together in the order in which they appear in the report configuration XML.

You can also call the individual APIs of `ITransformer` to do sorting, grouping or paging.

However, it is tedious to call these APIs: they are usually used for back-end based applications. For the common UI features, such as sorting/grouping/paging, the reporting API offers a web helper class, `ReportActionHelper,` to shield you from the low level APIs. This class is a façade to the Report Engine APIs. In most cases, your struts action should call this helper class instead of calling the lower-level reporting APIs. However, you can always access the report APIs directly if you want to. The action used by the product, Com.edocs.app.reporting.actions.ReportAction, call this helper class. You can do a similar thing in your action class.

3 Core Reporting Features

This chapter describes some of the most important features of reporting engine, and how to use them in your application.

Sorting

Sorting is a built-in feature of the report engine. It is available when you use the `ReportActionHelper` class from your action class. With the reporting XML and template, enabling sorting is as easy as configuring a transformer's column. For example:

```
<column sortable=”true” …/>
```

Only single column sorts are supported. The sorting is done in-memory, to eliminate accesses to the data source.

Set the column attribute sortable to true. The reporting engine reads the configuration, instructs the template to generate a sort-able link for the corresponding table column name, and the `ReportActionHelper` class calls the `ITransformer.sort()` API.

When a column is defined as sort-able, the report_body.vm template renders the column of the HTML table with a URL link. For example:

```
<a
href="$link?sortColumn=$x&reportId=$reportId&transformerId=$transformerConfig.i
d&currentSortColumn=$currentSortColumn&ascending=$ascending&currentGroup=$group
Index">
```

The following table describes the parameters in the URL:

Parameter	**Description**
`$link`	The URL context base. It is set to http://host:port/<web-context>/report.do in `ReportActionHelper` class, where <web-context> is the web context you defined in your EAR file.
`SortColumn=$x`	This is the column index of the column being sorted in the transformer configuration.
`Reported=$reported`	The report id of the report.
`TransformerId=$tranformerConfig.id`	The id of the transformer currently being sorted
`CurrentSortColumn=$currentSortColumn`	This is the current column being sorted in this transformer.
`Ascending=$ascending`	True or false; defines the sort order.
`CurrentGroup=$groupIndex`	Not used but may be used for grouping.

The web component must process the URL request, and calls the `ITransfomer.sort()` method to sort the column. The Helper class, `ReportActionHelper` does this work for you.

Just call the `ReportActionHelper` in your struts action. It processes this request and calls `Itransfomrer.sort()` to sort the column, then re-renders the newly sorted report for you.

Paging

Paging is a built-in feature of the reporting engine. Use the `ReportActionHelper` class and the default templates (or templates based on the defaults) to access that function. The main paging template is paging.vm, which is included in report_body.vm.

Paging is enabled when:

- you specify pageSize for transformer in report xml.
- `<transformer <pageSize="20" />`

Since reports are loaded and cached in the user session, paging is done on cached data. This method of paging doesn't scale when there are a large number of rows of data. For that case, you must limit the number of rows retrieved using the maxRows attribute of the <query> element.

Dynamic SQL

Some situations will require you to generate SQL dynamically. For example, you may have a report that searches the call details. One of the criteria is the call date. You want to search for call date equals a particular date, or you want to search for call dates between a start date and end date. Since the where clause is different for these two search cases, without dynamically generated SQL, we will be forced to write two reports with two SQL clauses. Dynamically generated SQL can solve this problem easily; the where clause of the SQL statement can be generated based on the current operation (equal or between), so you will only need one report.

The reporting engine allows you to write an SQL query in a Velocity template, so that the SQL query will be parsed before it is executed. You must set the dynamic attribute of <query> to true. For example:

```
<query dynamic="true"> <![CDATA[
      select * from my_table where #if ($equal) date = ? #else date >= ? and
date <= ? #end
]]></query>
```

`$equal` is a variable set by the caller through the `IreportActionCallback` interface. It is true if the user selects the “date equal” operation, and false if the user chooses the “date between” operation.

Note that the number of ?s is different based on operation types: one for equal and two for between. To solve this problem, the report engine supports binding a Collection object to ?s. The report engine loops through the Collection and binds each element to ?s.

For example, as in the preceding example, here is an example of how to bind:

```
<inputBinding object=”form” property=”parameter(dateList)” />
```

The method `form.getParameter(“dateList”)` returns a list of Date objects, and each date in the list is bound to the ?s in the query. The caller of reporting engine is responsible for collecting the list of dates and passing them to `ActionForm.setParameter(“dateList”, dateList)` (This assumes that `ActionForm` is as map-backed form, and has a pair of `setParmeter` and `getParameter` methods).

Another common use case is to generate the “in” operation in a where clause. The number of ?s will be based on the size of a Collection object.

For example, assume that we have a list of categories saved in a List, and we want to generate a where clause:

```
Where category in (?,?,..,?,?)
```

Where the number of question marks is the size of the List.

When doing the input binding, there is only one List, but loops through the List to set the ?s in the SQL as appropriate. This ensures that the number of question marks match the number of variables passed in.

There is a macro to help you generate the number of ?s based on the collection size:

```
#macro getSQLVariablesIgnoreNull($list $columnName)
```

Which generates the list.size() number of ?s. For example:

```
select * from my_table where date in getSQLVariablesIgnoreNull($dateList
“date”)
```

If the dateList size is 2, and it is Oracle database, then the result is:

```
select * from my_table where date in (NVL(?,date), NVL(?,date))
```

Where NVL means ignore this ? if it is null.

Internationalization / Localization

To support internationalization, the following points must be considered:

- Regular text on the report UI
- Some text coming from data source
- Chart: title and amount format, etc
- Date format, number format, etc.

Resource bundles are used to support internationalization. This section discusses internationalization for velocity templates.

Since the reporting engine is using Velocity templates, we cannot take advantage of the JSP <message> tag or Struts’s internationalization framework. Instead, the reporting engine has its own internationalization mechanism specially designed for velocity templates, which has following features:

- Allows a user to specify any resource bundle, just like Struts `config` does.
- Allows a user to format a string as Y does, for example, "My name is {0}".
- Provides a seamless integration with Struts if it is used. For example, sharing the same resource bundle.
- Offers a better way to handle default messages than Struts. In Struts, a not-found resource is either returned as null or as ???<locale><resource_key>???. With the reporting engine, you can configure it to return the key itself when the value of the key is not found.

Resource bundle definition

The resource bundle files used by the reporting engine templates are defined in the report XML files under the `<reports>` tag. The following example comes with Billing Analytics, and is defined in telco_global.xml.

```
<localizer enableMessageResources="true" defaultCode="1">
        <resourceBundle
        name="com/edocs/app/reporting/resources/ApplicationResources" />
</localizer>
```

You must use "/" instead of "." in the name of the resource bundle, which differs from Struts message resource.

The "<localizer>" tag defines how text will be localized. You can define multiple `<resourceBundle>` tags. Each `resourceBundle` tag defines a resource bundle file, and its name is defined by name attribute.

When the reporting engine searches for the resource bundle, it first checks whether this bundle exists as a file under EDX_HOME, or the current directory if EDX_HOME is not defined. If that fails, it will try to find it as a class.

The attribute `enableMessageResources` enables you to use Struts `MessageResource` to search for a resource.

The attribute `defaultCode` enables you to define the default behavior if a resource is not found. "0" means to use the key as the default value; "1" means to use Struts notion of "???<locale>.<key>???" and "-1" means throw an exception. The default value for `defaultCode` is "0".

The search order for finding a resource is:

1. If `enableMessageResources` is true, and the `MessageResource` does exist (it may not exist for non-Struts app), search the resource from `MessageResource`, return if found.
2. For each resource bundle defined in `resourceBundle`, load the bundle as either file or class, and then search the resource in the order it appears, return if found.
3. If nothing is found, use `defaultCode` described previously.

If you check the resource bundle name in the struts configuration file, you will notice that the same file, com/edocs/app/reporting/resources/ApplicationResources, is defined in both the Struts and report XML files. The only difference in the definitions is the file separators: reporting uses "/" and Struts uses ".". The same file is in two locations in order to support batch reporting. A batch job is not a Web application, so it does not have access to Struts `MessageResource`. This is also true if you are using the reporting engine at the EAR level. For example, you can generate an email message from an MDB event hander or from an EJB. However, if you are using Struts, and you using the reporting engine for online applications only (not batch reporting), then you don't need to define a `resourceBundle`, because the on-line web application can always find resources from `MessageResource`.

Because the same resource is defined twice, both Struts and the reporting engine load the same resource bundle and cache them (twice). Usually, this is not a problem, because a resource bundle file is small. However, if you do want to reduce memory usage, you can put all the template related resources into one file. Or, you can be more selective by putting only the batch report/email/AR related resources into one file, and load it by using the `resourceBundle` tag in report xml.

We recommend that you define the resource bundle as flat file under EDX_HOME. That allows you to modify the file easily and reload it by using this URL without restarting:

http://localhost:7001/tbmb/tam/reporting/reloadReportConfig.jsp

If you want to use a struts message source, which is loaded from the classpath, you can disable it during the development stage by setting `enableMessageResource` to false and loading a resource bundle from file system.

Setting `defaultCode` to "1" makes it easy to find all the text not being internationalized properly. You may want to set it to "0" for demonstration purposes.

Localization of Text

The localization of text in report is done through the `#localize` macro, which is defined in reporting_library.vm. It is defined as:

```
#macro (localize $name)
```

For example, in your template, you can call this macro like this:

```
#localize("name")
```

Which searches the report bundle to find a key with a value that matches "name".

All the texts defined in report.xml are treated as resource bundle keys. For example, report names and column labels. In the report template files, all the texts are localized through the `#localize` macro.

Localization of Data from a Data Source

By default, the text data retrieved from data source is not localized. You need to specially turn on this option. In this case, the text data from data source will be used as keys to search reporting resource bundles.

The localization of data from data source is done through the “localize” attribute of transformer column configuration in report xml. See report XML description above.

```
<column id=”call type” localize=”true” />
```

Which means that the column data retrieved from the database will be localized.

Localization of Charts

The chart components (chart title, labels and data) are localized by the `ITransformer.writeChart()` method. The chart tile is searched as a regular resource bundle name. Label and data are localized if the localize attribute is set to true for the corresponding columns.

Locale

To support internationalization, you must pass the Locale object to `ReportContext` by calling `setLocale()`. If `ReportContext` doesn’t have a locale defined, when you call the `IReportActionHelper.execute()` method, it puts the Struts locale object in session.

Dynamic Localization

Velocity is used to support localization. Velocity acts similar to the way `java.text.MessageFormat` does, and achieves the same result. The reporting engine parses the resource value as a Velocity template, whose resource key ends with ".vm", and returns the parsed value. For example,

```
rpt.test.vm=My name is $name.
```

Object "name" should come from the report context.

This feature can make any text in your report dynamic. For example, if you are on the account detail page, to display the report tile as “Account detail for <account_number>” instead of the default text, define the report title as a .vm resource bundle. For example:

```
rpt.accountDetail.title=Account detail for $form.accountNumber
```

Where `accountNumber` is from the Struts `ActionForm`.

Object Data Source

You may often not have access (either physically or politically) to the data base. So the reporting engine provides an API to get back a list of Objects, which can be presented in a table with paging or sorting. The reporting engine offers an Object data source to provide that feature.

The object data source is defined as:

```
<dataSource id="ds1" uri="object:reportList">
    <columns>
        <column id="id" type="java.lang.String"/>
        <column id="name" type="java.lang.String"/>
    </columns>
</dataSource>
```

This example states that there is an object called `reportList` in `ReportContext`, and you must put that object into `ReportContext` before calling `IReportActionHelper`. This object can either be a List (`java.util.List`), List of objects, or a List of JavaBean Objects.

If it is a List of List of objects, then it is assumed that the objects in the inner list are basic Java objects, such as String or Integer. They must also match the types defined in the dataSource column.

Usually, the object is a List of JavaBean objects. For example, as shown in the example XML, `reportList` is a List of `IReportConfig` objects (see the report API javadoc for more information). The reporting engine uses reflection to get the property values of the JavaBeans, whose property names match the column Ids defined in the example XML, and converts this List of JavaBeans into a List of Lists of JavaBean property objects (more precisely, into a `IReportList` of `IReportRow` objects). Here, it is also assumed that the JavaBean properties are basic Java types. In the example, for each `IReportConfig` object in the list, the report engine calls `IReportConfig.getId()` and `IReportConfig.getName()`, and converts the List of `IReportConfig` objects into an `IReportList` object. Each element in `IReportList` is an `IReportRow` object. Each `IReportRow` includes two elements, the report Ids and the report names.

Then define the rest of the report XML, including transformers, as usual.

The object data source enables the reporting engine to connect to other data sources currently not directly supported by the reporting engine. For example, you may have a CORBA interface that retrieves financial data from a legacy system. You can still use the report engine to present the data, as long as you can convert the data into a List of Lists of objects.

DSV Data Source

This feature allows you to read a delimiter-separated string as a data source. Here is the URI format of this data source:

“`dsv:inline:,:|`”

Where “dsv” stands for Delimiter Separated Values; “inline” means that the data can only be embedded in the report XML (support is not available for reading data from a file); ‘,’ means the column separator, and “|” means line separator.

For example:

```
<dataSource id="ds" uri="dsv:inline:,:|">
     <query><![CDATA[0,Business|1,Personal]]></query>
     <columns>
            <column id="value" type="java.lang.Integer"/>
            <column id="name" type="java.lang.String"/>
     </columns>
</dataSource>
```

The data source will be transferred into an `IReportList` with two `IReportRows`. The first row has values of “0” and “Business”, and second row has values of “1” and “Personal”. You can use this data source to implement the “split-billing” feature. For example, you can generate a dropdown list for call details and allow the user to change a call from personal to business or vise versa.

Drill down and Breadcrumb link:

The reporting engine allows you to build a breadcrumb link while you are drilling down from report to report.

To build drilldown link, you need to define a <link> for a transformer column:

```
<report id=”testrpt0”>
  <transformer id=”tr1” dataSourceId=”ds1”>
       <column id="invoice_number" name="Invoice number"  >
       <link title="Drill down to the invoice detail."><![CDATA[
       report.do?reportId=testrpt1c&invoiceNumber=$row.get(1)&parameter(parentNo
de)=root
       ]]></link>
       </column>
  </transformer>
</report>
```

The <link> element instructs the reporting engine to build a drilldown link for each account number. You must construct the link, which should point to another report. The link will be parsed as a Velocity template.

This link also has a “title” attribute, which allows you add an HTML “title” to the link. In most browsers, the title will be displayed as popup help.

When you click on an account number, you will drilldown to “testrpt1” report. However, by default, there is no breadcrumb link built to allow you to go back to the “testrpt0” report. To enable the breadcrumb link, add `”enableDrillUp”=true` to the column definition:

```
<report id=”testrpt0”>
  <transformer id=”tr1” dataSourceId=”ds1”>
    <column id="invoice_number" name="Invoice number" “enableDrillUp”=true >
       link title="Drill down to the invoice detail."><![CDATA[
       report.do?reportId=testrpt1c&invoiceNumber=$row.get(1)&parameter(parentNo
de)=root
       ]]></link>
    </column>
  </transformer>
</report>
```

When this flag is set to true, and you drilldown from testrp0 to testrpt1, there will be a breadcrumb link in the `testrpt1` view which allows you to go back to the `testrpt0` report.

Currently, you must drill down from one report id to another report id , but the breadcrumb link won’t work if you try to drilldown to the same report. This feature makes sense when you are viewing the same report but drilldown through hierarchy.

4 Customizing the Reporting Engine

This chapter describes how to customize the reporting engine. The examples use Struts and Tiles for the presentation framework, but the same techniques can be used for any other web presentation framework.

You may wish to customize the reporting engine to add the following features:

- Write your own Report XML
- Modify report templates
- Extending reporting engine through Reporting API

Write Your Own Report XML

The first step in creating your own report is to create your own report XML. Each report XML is project-specific. The best way to start is to use existing report as a base for your modifications

CAUTION: The reporting engine has a DTD, but is not used to validate the report XML. Therefore, make sure you do not to miss required attributes or XML elements.

You can create one report XML, which includes all the reports for your project, or you can create one XML file for each report. Remember to register all your report XML files in the reportList.properties file, and to give each XML file a unique name.

After creating your own report XML you can test it through the default template. Name your report id with a prefix of “telco_std”, which will cause it to be loaded into the standard billing report list of Billing Analytics

CAUTION: Make sure that each report has a unique name across all the reports in all report XMLs, or else a latter one will overwrite the previous one.

Customize the Report Template

After you have created a report XML and familiarize yourself with how the report engine renders the report, you may wish to customize the report template to generate the look and feel of your project.

A set of templates are provided with the report product. To customize them, make a copy of each template, put it into a new template directory, and change your report XML to point to the new directory.

Feel free to add new objects into the report context (and thereby, the velocity context) through the `IReportActionCallback` interface. But do not to overwrite the existing context variables. One technique is to use a special prefix (for example, “_”) for your custom context variables.

The CSS for the reporting HTML is defined in a file called skin.css (see Reporting Package on page 13 for more information). You can modify this file to change the CSS of the report UI.

Write Your Own Action Classes and Re portForm.

You should write your own Action class and action form for your reports. Use the `ReportActionHelper` class to take care of common issues such as sorting and paging.

When writing your own action class, you must call the `ReportActionHelper.execute()` method. See the preceding section about Integration with Struts and Tiles for details about how to invoke this method.

When defining your own Struts `ActionForm`, you can make the form map-based, which allows you to pass any parameter into the reporting engine without explicitly adding a set of get and set methods. The only downside to this method is that a map-based property cannot be passed into JavaScript for client side validation.

For example, you can define two map methods: `public Object getParameter(String name)` and `void setParameter(String name, Object value)`. To use these parameters, in an HTML form or URL, use a notion similar to this:

```
“parameter(contractNumber)=123456”
```

Which passes the contract number to struts, which calls `setParameter()` on your `ActionForm` to put the `contractNumber` into the map. This parameter can either be used as an SQL data source input binding or used in template.

To retrieve the parameter as an `inputBinding`, use:

```
<inputBinding object=”form” property=”parameter(contractNumber)” />
```

To retrieve the parameter from the template , use:

```
$form.getParameter(“contractNumber”).
```

Packaging

You can package your Struts action classes as usual at the WAR level. For struts forms, if you are not using batch report, then you can package them at the WAR level, but if you do use batch report, the forms must be accessible by non-web components such as the Common Center batch report job. In that case, you must package your report forms at the EAR level. For example, make them part of the reporting-ext.1.2.1.jar file.

You must register your report XML files in the reportList.properties file, and put the report XML files in <edx_home>/config/rpt. However, it is possible to put the report XML files under the other sub-directories of <edx_home>.

Hiding Report Columns and Manipulating IReport

After you call `IReportActionHelper` and get back an `IReport` object, you can manipulate the object before forwarding it to report.jsp.

For example, you may want to hide some columns based on certain conditions. Get the `IReportConfig` object from `IReport`, find the `ITransformerColumnConfig` of the corresponding columns, and set the "`isHidden`" attribute based on your conditions.

Unlimited Paging

By default, the reporting engine retrieves only 1000 rows from the data source due to performance issues even though there are more rows. But you can configure the number of rows the reporting engine retrieves (maxRows or fetchSize) in the report xml file.

Here is where you configure the size in the sample report.xml:

```
<transformer id="tr1" …>
        <paging fetchSize="2000"/>
</transformer>
```

If you want to retrieve all the rows from the data source without taking a performance hit, you can use unlimited paging. If you enable unlimited paging, the reporting engine gets the result set in batches and allows end users to page through across multiple batches. Each batch constitutes a fetch.

The term "unlimited paging" may be misleading. This feature gets result set rows in multiple fetches on demand when the user requests them. The user just pages through the result set as if it is a regular paging, if the requested page is not in the current fetch, the reporting engine gets the next fetch from the data source. However, all the intricacies of checking if the requested page is in the current fetch, if not getting next the fetch, are hidden from the end user.

The number of result set rows in one fetch is defined as fetch size. And you can configure fetch size and page size in report xml. Here is the sample xml where you can enable unlimited paging and to define the fetch size:

```
<transformer id="tr1" pageSize="20"  …>
        <paging unlimited="true" fetchSize="5000"/>
</transformer>
```

Currently reporting engine supports unlimited paging for SQL data source and object data source. If unlimited paging is enabled, sorting and calculator is not supported for now because we need to sort and apply calculator operations for all the result set across all the fetches rather than current fetch.

Unlimited Paging for SQL data source:

For the SQL data source you define the query as usual. The reporting engine embeds this query with in select count(*) to get the size of the total result set.

Unlimited Paging for Object data source:

For the object data source to get the result set in batches, the data source needs to provide reporting engine a call back method which retrieves the data from start position to end position. For this purpose, the reporting engine expects an object which implements the call back interface in report context rather than result set object. That means, for regular paging, you put result set list or array of objects in the report context and for unlimited paging, you put an object which implements call back interface. This call back interface is called IReportObjectResultSet and has following methods

```
public Object getResultSet (ReportObjectSearchCriteria objectSearchCriteria);

public int getResultSetSize();
```

ReportObjectSearchCriteria object has start position and end position of a fetch.

So the object you put in the result set needs to implement getResultSet and getResultSetSize().

getResultSet(ReportObjectSearchCriteria objectSearchCriteria) method returns result set rows from start position to end position defined in objectSearchCriteria.

getResultSetSize() method returns size of the complete result set that data source returns. If you do not know the result set size, you can return IReportObjectResultSet.unknownResultSetSize

Development Tips

The following list of development tips can help you using reporting engine.

Reloading report xml and templates without re-starting the server

If you change the report XML, you can use following URL to reload it:

http://localhost:7001/tbmb/tam/reporting/reloadReportConfig.jsp

When you change the Velocity templates, the templates should be loaded by the Velocity engine automatically. However, because of browser caching issues, sometimes you may need to restart the server or cleanup the browser’s cache.

If you are putting the resource bundle files under EDX_HOME and load them through `<localizer>`, then the resource bundle can also be reloaded with preceding URL.

CAUTION: The URL won’t work in a clustered environment, because it only refreshes the cache in one JVM.

5 Questions and Answers

When should I use the report engine?

If you are working on the UI, you should consider using the report engine whenever you want to present a tabular table with sorting and paging functionality. For a non-tabular based UI, you should use JSP files.

Also, the reporting engine is a great tool for generating dynamic text files, such as AR files or email content.

Is Velocity better than JSP?

This question is hotly debated. Our experience is that Velocity is much less powerful and complex than JSP and follows MVC module closely. This forces you NOT to put business logic in the template. However, because of this, you may find that Velocity is less convenient than JSP and sometimes can be awkward to use. For your application, we recommend that you stick with JSP for most parts and only use Velocity templates for reporting related UIs.

Then why use Velocity instead of JSP at first place?

The reporting engine uses Velocity instead of JSP because:

- Velocity offers a better MVC module, which keeps most of the business logic in the core reporting engine APIs.
- The reporting engine is meant to be used by both front end and back end applications. For back end applications, JSP is not available.
- The views (templates) should be publishable and versioned. This is important if you want to use the reporting engine to present bills. We can use one set of templates for bills in one period, and change to another set of templates for another period. This feature is disabled in the current application, but it will be enabled in a future version. Note that there is **no** easy way to publish JSP pages.

Can reporting engine generate a PDF view?

The reporting engine can only generate a text based view, such as HTML, CSV and XML. A future version of reporting engine may support PDF view.

Can I download a newer version of Velocity and use it to replace the one in Billing Analytics EAR?

This is not recommended. First, the new Velocity version has not been tested with Billing Analytics. Second, the default velocity.properties has been changed for Billing Analytics. These changes include: the `velocityCount` starts from 0 instead of the default 1, and the templates can be loaded as file and also as class.

Can I define my own data source?

Not currently. The easiest way to get around this is to retrieve your data as a List of objects and then use the Object Datasource feature to present it through the report engine.

Can I extend report XML to add my own custom tags?

Not currently. However, it is possible that in the future release, we may add flexible attributes to allow you pass in your custom information into reporting engine.