QQQ

Table of Contents

Introduction 1
Meta Data 2
  QQQ Tables 2
  QQQ Reports 3
Actions 7
  RenderTemplateAction 7

Introduction

QQQ is …

- Framework
- Declarative
- Easy thing easy; Hard thing possible
- Customizable

Meta Data

QQQ Tables

The core type of object in a QQQ Instance is the Table. In the most common use-case, a QQQ Table may be the in-app representation of a Database table. That is, it is a collection of records (or rows) of data, each of which has a set of fields (or columns).

QQQ also allows other types of data sources (QQQ Backends) to be used as tables, such as File systems, API's, Java enums or objects, etc. All of these backend types present the same interfaces (both user-interfaces, and application programming interfaces), regardless of their backend type.

QTableMetaData

Tables are defined in a QQQ Instance in a `QTableMetaData` object. All tables must reference a QQQ Backend, a list of fields that define the shape of records in the table, and additional data to describe how to work with the table within its backend.

QTableMetaData Properties:

- `name` - **String, Required** - Unique name for the table within the QQQ Instance.
- `label` - **String** - User-facing label for the table, presented in User Interfaces. Inferred from `name` if not set.
- `backendName` - **String, Required** - Name of a QQQ Backend in which this table's data is managed.
- `fields` - **Map of String → QQQ Field, Required** - The columns of data that make up all records in this table.
- `primaryKeyField` - **String, Conditional** - Name of a QQQ Field that serves as the primary key (e.g., unique identifier) for records in this table.
- `uniqueKeys` - **List of UniqueKey** - Definition of additional unique constraints (from an RDBMS point of view) from the table. e.g., sets of columns which must have unique values for each record in the table.
- `backendDetails` - **QTableBackendDetails or subclass** - Additional data to configure the table within its QQQ Backend.
- `automationDetails` - **QTableAutomationDetails** - Configuration of automated jobs that run against records in the table, e.g., upon insert or update.
- `customizers` - **Map of String → QCodeReference** - References to custom code that are injected into standard table actions, that allow applications to customize certain parts of how the table works.
- `parentAppName` - **String** - Name of a QQQ App that this table exists within.
- `icon` - **QIcon** - Icon associated with this table in certain user interfaces.
- `recordLabelFormat` - **String** - Java Format String, used with `recordLabelFields` to produce a label shown for records from the table.
- `recordLabelFields` - **List of String, Conditional** - Used with `recordLabelFormat` to provide values

for any format specifiers in the format string. These strings must be field names within the table.

  - Example of using `recordLabelFormat` and `recordLabelFields`:

```
// given these fields in the table:
new QFieldMetaData("name", QFieldType.STRING)
new QFieldMetaData("birthDate", QFieldType.DATE)

// We can produce a record label such as "Darin Kelkhoff (1980-05-31)" via:
.withRecordLabelFormat("%s (%s)")
.withRecordLabelFields(List.of("name", "birthDate"))
```

- `sections` - **List of QFieldSection** - Mechanism to organize fields within user interfaces, into logical sections. If any sections are present in the table meta data, then all fields in the table must be listed in exactly 1 section. If no sections are defined, then instance enrichment will define default sections.
- `associatedScripts` - **List of AssociatedScript** - Definition of user-defined scripts that can be associated with records within the table.
- `enabledCapabilities` and `disabledCapabilities` - **Set of Capability enum values** - Overrides from the backend level, for capabilities that this table does or does not possess.

QQQ Reports

QQQ can generate reports based on QQQ Tables defined within a QQQ Instance. Users can run reports, providing input values. Alternatively, application code can run reports as needed, supplying input values.

QReportMetaData

Reports are defined in a QQQ Instance with a `QReportMetaData` object. Reports are defined in terms of their sources of data (`QReportDataSource`), and their view(s) of that data (`QReportView`).

QReportMetaData Properties:

- `name` - **String, Required** - Unique name for the report within the QQQ Instance.
- `label` - **String** - User-facing label for the report, presented in User Interfaces. Inferred from `name` if not set.
- `processName` - **String** - Name of a QQQ Process used to run the report in a User Interface.
- `inputFields` - **List of QQQ Field** - Optional list of fields used as input to the report.
  - The values in these fields can be used via the syntax `${input.NAME}`, where `NAME` is the `name` attribute of the `inputField`.
  - For example:

```
// given this inputField:
new QFieldMetaData("storeId", QFieldType.INTEGER)

// its run-time value can be accessed, e.g., in a query filter under a data source:
new QFilterCriteria("storeId", QCriteriaOperator.EQUALS, List.of("${input.storeId}"))

// or in a report view's title or field formulas:
.withTitleFields(List.of("${input.storeId}"))
new QReportField().withName("storeId").withFormula("${input.storeId}")
```

- `dataSources` - **List of QReportDataSource, Required** - Definitions of the sources of data for the report. At least one is required.

QReportDataSource

Data sources for QQQ Reports can either reference QQQ Tables within the QQQ Instance, or they can provide custom code in the form of a `CodeReference` to a `Supplier`, for use cases such as a static data tab in an Excel report.

QReportDataSource Properties:

- `name` - **String, Required** - Unique name for the data source within its containing Report.
- `sourceTable` - **String, Conditional** - Reference to a QQQ Table in the QQQ Instance, which the data source queries data from.
- `queryFilter` - **QQueryFilter** - If a `sourceTable` is defined, then the filter specified here is used to filter and sort the records queried from that table when generating the report.
- `staticDataSupplier` - **QCodeReference, Conditional** - Reference to custom code which can be used to supply the data for the data source, as an alternative to querying a `sourceTable`.
  - Must be a `JAVA` code type
  - Must be a `REPORT_STATIC_DATA_SUPPLIER` code usage.
  - The referenced class must implement the interface: `Supplier<List<List<Serializable>>>`.

QReportView

Report Views control how the source data for a report is organized and presented to the user in the output report file. If a DataSource describes the rows for a report (e.g., what table provides what records), then a View may be thought of as describing the columns in the report. A single report can have multiple views, specifically, for the use-case where an Excel file is being generated, in which case each View creates a tab or sheet within the `xlsx` file.

QReportView Properties:

- `name` - **String, Required** - Unique name for the view within its containing Report.
- `label` - **String** - Used as a sheet (tab) label in Excel formatted reports.
- `type` - **enum of TABLE, SUMMARY, PIVOT. Required** - Defines the type of view being defined.

    - **TABLE** views are a simple listing of the records from the data source.
    - **SUMMARY** views are essentially pre-computed Pivot Tables. That is to say, the aggregation done by a Pivot Table in a spreadsheet file is done by QQQ while generating the report. In this way, a non-spreadsheet report (e.g., PDF or CSV) can have summarized data, as though it were a Pivot Table in a live spreadsheet.
    - **PIVOT** views produce actual Pivot Tables, and are only supported in Excel files *(and are not supported at the time of this writing)*.
- `dataSourceName` - **String, Required** - Reference to a DataSource within the report, that is used to provide the rows for the view.
- `varianceDataSourceName` - **String** - Optional reference to a second DataSource within the report, that is used in `SUMMARY` type views for computing variances.
    - For example, given a Data Source with a filter that selects all sales records for a given year, a Variance Data Source may have a filter that selects the previous year, for doing comparissons.
- `pivotFields` - **List of String, Conditional** - For **SUMMARY** or **PIVOT** type views, specify the field(s) used as pivot rows.
    - For example, in a summary view of orders, you may "pivot" on the **customerId** field, to produce one row per-customer, with aggregate data for that customer.
- `titleFormat` - **String** - Java Format String, used with `titleFields` (if given), to produce a title row, e.g., first row in the view (before any rows from the data source).
- `titleFields` - **List of String, Conditional** - Used with `titleFormat`, to provide values for any format specifiers in the format string. Syntax to reference a field (e.g., from a report input field) is: `${input.NAME}`, where `NAME` is the `name` attribute of the inputField.
    - Example of using `titleFormat` and `titleFields`:

```
// given these inputFields:
new QFieldMetaData("startDate", QFieldType.DATE)
new QFieldMetaData("endDate", QFieldType.DATE)

// a view can have a title row like this:
.withTitleFormat("Weekly Sales Report - %s - %s")
.withTitleFields(List.of("${input.startDate}", "${input.endDate}"))
```

- `includeHeaderRow` - **boolean, default true** - Indication that first row of the view should be the column labels.
    - If true, then header row is put in the view.
    - If false, then no header row is put in the view.
- `includeTotalRow` - **boolean, default false** - Indication that a totals row should be added to the view. All numeric columns are summed to produce values in the totals row.
    - If true, then totals row is put in the view.
    - If false, then no totals row is put in the view.

- `includePivotSubTotals` - **boolean, default false** - For a **SUMMARY** or **PIVOT** type view, if there are more than 1 **pivotFields** being used, this field is an indication that each higher-level pivot should include sub-totals.
    - TODO - provide example
- `columns` - **List of QReportField, required** - Definition of the columns to appear in the view. See section on QReportField for details.
- `orderByFields` - **List of QFilterOrderBy, optional** - For a **SUMMARY** or **PIVOT** type view, how to sort the rows.
- `recordTransformStep` - **QCodeReference, subclass of** `AbstractTransformStep` - Custom code reference that can be used to transform records after they are queried from the data source, and before they are placed into the view. Can be used to transform or customize values, or to look up additional values to add to the report.
    - TODO - provide example
- `viewCustomizer` - **QCodeReference, implementation of interface** `Function<QReportView, QReportView>` - Custom code reference that can be used to customize the report view, at runtime. Can be used, for example, to dynamically define the report's **columns**.
    - TODO - provide example

QReportField

Actions

RenderTemplateAction

The `RenderTemplateAction` performs the job of taking a template - that is, a string of code, in a templating language, such as Velocity, and merging it with a set of data (known as a context), to produce some using-facing output, such as a String of HTML.

Examples

Canonical Form

```
RenderTemplateInput input = new RenderTemplateInput(qInstance);
input.setSession(session);
input.setCode("Hello, ${name}");
input.setTemplateType(TemplateType.VELOCITY);
input.setContext(Map.of("name", "Darin"));
RenderTemplateOutput output = new RenderTemplateAction.execute(input);
String result = output.getResult();
assertEquals("Hello, Darin", result);
```

Convenient Form

```
String result = RenderTemplateAction.renderVelocity(input, Map.of("name", "Darin"),
"Hello, ${name}");
assertEquals("Hello, Darin", result);
```

RenderTemplateInput

- `code` - **String, Required** - String of template code to be rendered, in the templating language specified by the `type` parameter.
- `type` - **Enum of VELOCITY, Required** - Specifies the language of the template code.
- `context` - **Map of String → Object** - Data to be made available to the template during rendering.

RenderTemplateOutput

- `result` - **String** - Result of rendering the input template and context.