

QlikView



QlikView
Deployment Framework

Development Guide



Content

Version 1.3.2	4
Development Guide in a Deployment Framework context	4
Why the need for a Development Guide?	4
Company Specific Development Guide	4
Platform strategy from a development perspective	4
Standards	5
Deployment Framework	5
Container Naming Convention	5
Standard container content	6
Container Variables	6
Container variable names	6
Variable Naming Convention	7
Additional Variable standards	7
Version Control	7
Project folders (-prj)	8
Development skill set	8
Back End developers skill set	8
Front End developers skill set	8
QlikView Development Teams	8
Optimization strategy	9
Document Security	9
Back End Development	9
Scripting basics	9
Using Deployment Framework Containers when scripting	10
Get started with QDF in QlikView	10
1.Init.qvs	11
Adding additional scripts	11
Linking Containers together overview	12
How to link Systems Containers into Project Containers	13
What in the framework are applications depended on?	14
Deployment Framework Templates and Examples	14
QlikView Staging	15
Database Connection String	15
Security considerations	15
Include File	15
Global Variable	15
Security Tab (Hidden Script)	16
QlikView Security Table (Section Access)	16
Best Practices when using Section Access:	17
Reuse of script code	18
Include files	18
Sub Functions	18
Pre-Defined Functions	19
Data Modeling	26
Understanding	26
Data models	27
Multiple fact tables	27
Preceding Loads	29
Large Data Sets	30
Optimization Tips and Tricks	32
Additional scripting best practice	33
Other scripting best practices include:	33
Application logging	35

QlikView

Deployment Framework log tracing and debugging.....	35
Using binary load with Deployment Framework	35
Front End Development.....	36
UI Design	36
QlikView Developer Toolkit.....	36
Developer Toolkit is divided up into several folders of assets	36
UI Best Practice	37
Additional UI Best Practice.....	38
Color Scheme Variables	39
Variable expressions	39
Expression Optimization Tips	40
Macros	40
Actions	41
Tools	42
Variable Editor	42
Container Map Editor	42
Variable Editor Tab.....	44
Optimization tools	46
QlikView Optimizer application	46
Unused Fields application	46
Complexity Analyzer (included in Governance Dashboard).....	46
Troubleshooting & Support	47
Support Types	47
Appendix A, Checklists.....	48
Development Checklists.....	48
Optimization Checklist	49

Development Guide in a Deployment Framework context

Deployment Framework consists of the framework core and of several documents explaining how the framework works and how to develop and administrate QlikView according to best practice.

Deployment Framework is created for QlikView version 10 and 11.

The Deployment Framework documents consist of:

- **Getting Started Guide**
Get an overall understanding of the framework basics and how to start installing and develop.
- **Operations Guide**
Guide for QlikView Administrators maintaining the platform and administrating security, tasks and containers.
- **Development Guide**
Guide for Developers how to work with DF in an efficient way, naming conventions, data modeling, optimization tricks/tools and other guide lines regarding development.
- **Deployment Guide**
Guide for project manager and developers how to manage QlikView development, test, acceptance, production (DTAP) process, how to create a QlikView project and development teams and skill sets.
- **Governance Guide**
Guide for IT how QlikView platform and DF fits into different Governance models
- **Deployment Framework Core Documentation**
Detailed documentation regarding the Framework Core, folder structure, script logic and security.

Why the need for a Development Guide?

The Developer Guide is a reference manual for QlikView developers. QlikView developers are individuals who design and implement QlikView applications and their areas of expertise range from data modeling to scripting to UI design. This document is designed to facilitate much clearer understanding of the methodologies and practices that are optimal for producing highly usable, highly optimized and highly configurable QlikView applications, whether used by small departments or by large enterprises.

Company Specific Development Guide

This document is a high level guide on how to develop by using the Deployment Framework structures and pre-definitions. Best practice is that each company (QlikView Customer) creates its own Development guide based on this document in combination with customer environments specific needs, like:

- ETL and QVD strategy
- Container strategy
- Security requirements
- DTAP process

Platform strategy from a development perspective

When it comes to development, QlikView offers a wide range of flexibility. For many reasons there's a good idea to set up a corporate developing best practice. If you plan to create and use QlikView Data Files (QVD) files in your environment it's also wise to establish a QlikView Data File (QVD) strategy. A corporate developing standard doesn't only include standards for how to optimize each and every single application but does also embrace methodologies and practices like reusability and overview.



Standards

It's important to have and use standards when developing and maintaining QlikView. There are many ways of getting the same result, but not all of them are efficient and understandable. By use of Deployment Framework core structure in combination with the guide lines and standards we create consistency and a multi development environment. Standards are needed for:

- Reuse of data
- Reuse of code
- Reuse of expressions and variables
- Multiple development
- Governance
- Creating and collecting understandable metadata

Using standards will result in lower cost of ownership by making governance and maintenance much smoother.

Deployment Framework

The Deployment framework core is based on folder containers placed in the Source Document folder. Containers are identical but isolated file structures placed side by side. A container can be moved and/or renamed without changing any QlikView script or logic inside it. Each container has identical file structures and includes base script functionality inside the container. A newly installed Deployment Framework contains *0.Administration* container, it's from this container that new containers are created and managed. By default Deployment Framework also contains a shared folders container that should contain scripts and files that are reusable by all applications.

More details on containers and Deployment Framework core installation can be found in **Deployment Framework Core** documentation.

Container Naming Convention

Inside each container the subfolder names are standardized and simplified to fit as many languages and companies as possible. Before each container and subfolder there are a sequential number that makes it easier to identify containers and subfolders, especially when using Publisher. It's also used by the Initiation scripts to get uniqueness. Follow the number sequence and never use space when creating new containers or subfolders inside the container.

Standard container content

0.Template	Folder to keep custom examples and templates for easy reuse. <i>Only exists in 0.Administration Container.</i>
1.Application	QlikView Applications are resided in subfolders under 1.Applications
2.QVD	QlikView Data files are stored in subfolders under 2.QVD
3.Include	Folder where QlikView Include files are stored. These are script parts that are called from the main QlikView script.
1.BaseVariable	Stores all the variables needed to use the framework, like paths inside the container
2.Locale	Locale for different regions, used for easy migration between regions
3.ConnString	Stores connection strings to data sources
4.Sub	Store for sub routines, this is a way to reuse code between applications
5.ColorScheme	Company standard Color Scheme would be placed here
6.Custom	Store for custom include scripts
4.Mart	Resides QlikView Qvw marts (in subfolders) for data discovery usage, these folders could be shared.
5.Config	Configuration and language files like Excel and txt. This folders could be shared to make configuration changes easier
6.Script	Store for special scripts run by the publisher or scheduled tasks
7.Export	Folder used to store from QlikView exported data, probably txt or qvx
8.Import	Folder used to store import data from external systems
Info.txt	Information files describing the folder purpose and Path variable. There are Info files in every folder.
Version.xx.txt	Version Revision list

Container Variables

Each folder inside the container represents a unique environmental Global Variable inside QlikView. These variable names are the same for all containers, making it easy to move an application between containers without changing the scripts. The initiation Include file (*1.Init.qvs*) automatically creating the variables should always be added in at the start of the QlikView scripts.

Container variable names

0.Administration		
1.AcmeStore		
1.Application	<u>vG.ApplicationPath</u>	Application Folder
2.QVD	<u>vG.QVDPath</u>	QlikView Data files (QVD) repository
3.Include	<u>vG.IncludePath</u>	QlikView Include files repository
1.BaseVariable	<u>vG.BaseVariablePath</u>	Initiation script and Global Variables storage
2.Locale	<u>vG.LocalePath</u>	Regional / language Locale
3.ConnString	<u>vG.ConnStringPath</u>	Connection string repository
4.Sub	<u>vG.SubPath</u>	Sub Function Library
5.ColorScheme	<u>vG.ColorSchemePath</u>	Color and Picture templates
6.Custom	<u>vG.CustomPath</u>	Repository to keep custom include scripts
4.Mart	<u>vG.MartPath</u>	Qlik mart repository
5.Config	<u>vG.ConfigPath</u>	Stores QlikView configuration files
6.Script	<u>vG.ScriptPath</u>	Folder to store special scripts used by publisher
7.Export	<u>vG.ExportPath</u>	Data Export repository, probably txt or <u>gvx</u>
8.Import	<u>vG.ImportPath</u>	Data Import from external systems
2.AcmeHR		
99.Shared_Folders		

Variable Naming Convention

It's important to have a framework variable naming convention, so that existing application variables doesn't collide with the framework variables.

- Variables used across the entire container are called Global and thereby have the name standard **vG.xxx (Variable Global)**. The Global variables are controlled and edited via the Variable Editor.
- Variables only used in a single application are called Local and named **vL.xxx (Variable Local)** or **v.**
- Variables used across all containers are called Universal and named **vU.xxx (Variable Universal)**. The Universal variables are stored in the Shared Folders container and are controlled and edited via the Variable Editor.

The Global and universal variables are modified by the Variable Editor application (read more in VariableEditor section) and stored in *\$(BaseVariablePath)\CustomVariables.csv* files in each container. Global variables should only be used when variables are shared by several applications in a Container. Universal variables should be used when variables are shared by several applications across all Containers.

By using Universal Variables that are stored in *\$(SharedBaseVariablePath)\CustomVariables.csv* files in the Shared Folders Container, we get "single point of truth" across all containers. Universal Variables are by default loaded during the framework initiation process, have the prefix **vU** and is also modified by the Variable Editor application.

Additional Variable standards

- Store often used expressions as Local variables
- Store reusable expressions as Global variables
- Extended name standard for Variables, example:
 - Local expressions variables starts with **vL.Calc_**
 - Global expressions variables starts with **vG.Calc_**
- Variables defining a path should always end with a '\'
- Reset local variables that are only used inside the script, not by the UI.
Enter the variable name and =; example: SET **vL.test** =;

Version Control

Revision control, also known as version control, source control or software configuration Management (SCM) is the management of changes to documents, programs, and other Information stored as computer files. It is most commonly used in software development where a team of people may change the same files. As the development team of a QlikView Application grows, the need for SCM grows as well. QlikView 11 Developer is integrated with the Microsoft Team Foundation Server (TFS) and the popular Open Source version control system Subversion (SVN) in combination with the SVN-client Tortoise SVN.

The separate document *QlikView_Source_Control_Management_QV11.pdf* will describe typical developer/multi developer scenarios and how one can take advantages out of SCM in these situations.

Project folders (-prj)

From QlikView version 10 all version control systems will work with QlikView (without use of TFS or SVN integration) by using QlikView project folders (-prj), this is a folder created beside each QlikView File that contains xml data describing objects and scripts in the qvw file. To create the prj folders in the development container use the 1.Create-prj script.

Create-prj tool

The Deployment Framework tool called Create-prj.cmd creates *xxx-prj* folders beside the *xxx.qvw* files in the *1.Application* folder structure. The -prj folders are used as object and script repository and is usually used for version control of QlikView Files. No configuration needed to use *Create-prj.cmd* just execute, works with both physical and UNC part. The *Create-prj* folder and script will be copied to new containers when running *CreateNewContainer.qvw*.

Development skill set

The development process can be split into two overall groups, *Front End* and *Back End* development. One notice, an individual developer seldom contains all the skill sets, best is to the developers skills in the best way based on the skill sets needed. The Deployment Guide sections are bases on the skill sets below.

Back End developers skill set

- Typically DBA knowledge like
 - Data source expert
 - QlikView data modeling
 - QlikView data model optimization
 - Good understanding in ETL process
 - QlikView Section Access models

Front End developers skill set

- Typically a BI developer
 - Business specific understanding
 - QlikView front end function and features
 - KPI and measurements
 - QlikView Front End optimization
- Typically a designer skill set
 - Design skills
 - Visualization
 - Usability

QlikView Development Teams

QlikView is an extremely flexible and easily adapted BI tool. As such, development teams can organize around several models for support, administration, back and front end development, training and management. It is recommended that the client consult its own IT standards for development, as they may drive this decision, or at least narrow the allowed choices. QlikTech does not expressly promote one of these scenarios over the others, but asks that clients determine for themselves which of these configurations might work best, given the nature of the QlikView development and the skills sets that exist.

Read more regarding QlikView Development Teams in DF Deployment Guide



Optimization strategy

QlikView is known for its wide user adoption. One of the main reasons for this is its capability to manage large data sets with short response time. Although a QlikView application most often is easy and fast to develop it's a very good idea to establish an optimization strategy as part of your QlikView development platform. As with most QlikView development, optimization is divided into a back-end and a front-end part. While back-end optimization focus on effective script and data modeling, the front end focus on user interface design with its charts, dimension and expressions. For long term success it is strongly recommended that you have an optimization focus in your application development, especially when you know that the application should hold a large data set and be distributed to a large number of users. A good idea is to have an optimization step connected to the validation/approval phase in your development process, this of course both for new applications as well as for changed/ improved applications.

You can read more detailed information, tips and tricks, about optimization in the back-end and front-end section of this document.

Document Security

Security in a QlikView document is handled in two different ways which can be combined:

- Physical split of a master file into number smaller files. The split is created on values in one or more fields e.g. create one file for each field value in field Country. This process is managed by QlikView Publisher
- Authentication and Authorization of data is dynamically reduced within the QlikView document by using of a mechanism called Section Access which is managed within the Script editor and further described later in this document.

Find more information regarding security in the *Operations Guide* and Section Access section later in this document.

Back End Development

Back end development involves the process that starts with extracting data from one or many data sources and ends up in creating a QlikView associative data model. This is managed in QlikView script editor.

Scripting basics

Scripting is the environment in which a QlikView Developer will automate the extract, transform and loading process of bringing data in the QlikView environment. Each QlikView document (application) contains a script editor through which this process is enabled.

Best practices dictate that using multiple tabs within a script will split out the various parts, enabling a simple view of the information for future development and support. Depending on the complexity of the application, you may have a variety of different script sections. The common parts of a script are below:

- Security (usually hidden script)
- Dates and Calendar information
- Tab per data source
- Tab per key measure/core table
- Tab per lookup table

Using Deployment Framework Containers when scripting

When using the Deployment Framework Container concept, the applications created need to have an initiation include script in the beginning of the script which sets base folder search path within the container. This initiation script is called **1.Init.qvs** and resides under **Include\1.basevariable** in every container.

Other framework scripts are automatic linked or optional and depend on application purpose.

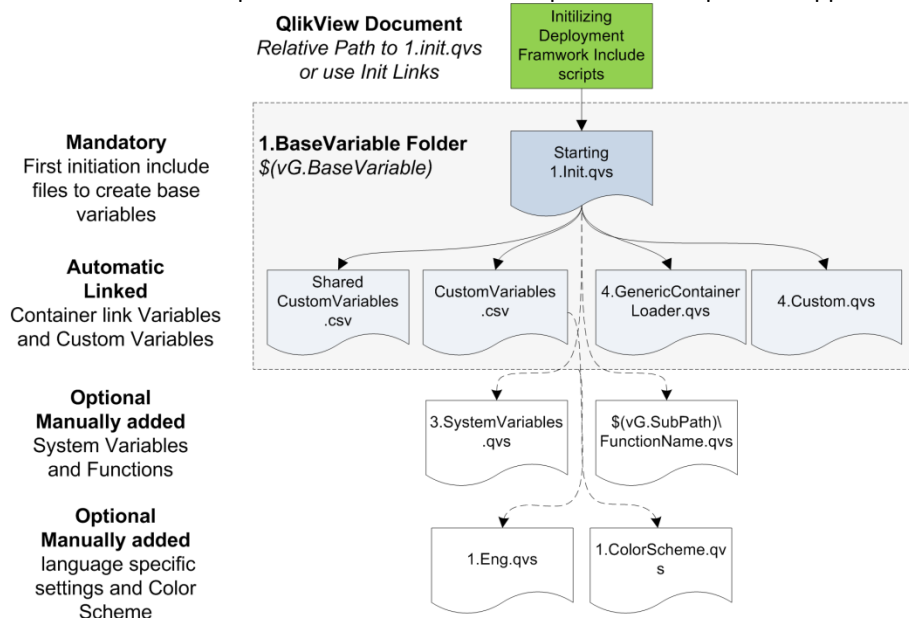


Diagram of deployment framework initiation process

The Diagram shows the **Mandatory** **1.init.qvs** script that will create Global Variables and Custom Global Variables. Some scripts are **Automatic linked** by 1.Init, these files and functions can be turned off in the 1.Init.qvs script. Some scripts are **Optional** and manually loaded into the QlikView Script.

Get started with QDF in QlikView

1. Create or save QlikView application (preferably in a subfolder) under 1.Application folder in the container.
2. Create a first Tab called **QlikView Deployment Framework** and Paste the script code down below in.
Use relative path for **1.init**, remember to change the relative path if the application path changes.

```
// First Base Variable Include use relative path to 1.init.qvs
// Contains base search path in this container
$(Include=..\3.include\1.basevariable\1.init.qvs);
$(Include=..\3.include\1.basevariable\1.init.qvs);
$(Include=..\3.include\1.basevariable\1.init.qvs);
$(Include=..\3.include\1.basevariable\1.init.qvs);
```
3. Reload and check in QlikView Variable Overview for new Global Variables called **vG.xxx**

QlikView

1.Init.qvs

1.Init populates global path variables for each container (example *vG.QVDPPath*). The initiation script is stored in *3.Include\1.BaseVariable\1.Init.qvs*.

Load the 1.init.qvs variables into QlikView Script in the beginning is mandatory, else the framework will break.

When entering *1.init* include statements into the QlikView script start, use relative path in QlikView.

Example: `$(Include=..\..\3.include\1.basevariable\1.init.qvs);`

Search for 1.Init.qvs

Here is an example of automatically identifying *1.Init.qvs* in the beginning of the script. The script is searching for a hidden *InitLink.qvs* script resided in the container base path (*vG.BasePath*).

```
// Advanced search for 1.Init.qvs only works with 0.95 or later
// Script to automatically identify and start Deployment Framework
// Based on InitLink.qvs stored in vG.BasePath
// Identifying Container based on InitLink.qvs
SET vG.BasePath = ;
SET vL.Path_tmp = ;
    for vL.x =1 to 10-1
        LET vL.Path='..\&'$(vL.Path_tmp)';
        LET vL.Path_tmp='$(vL.Path)';
        $(Include=$(vL.Path)InitLink.qvs);
        exit for when not '$(vG.BasePath)'= ''
    next
SET vL.Path = ;
SET vL.Path_tmp = ;
$(Include=$(vG.SubPath)\4.GenericContainerLoader.qvs);
```

Example how to identify that Deployment Framework is used, done by validating the *vG.BasePath* variable.

```
if not '$(vG.BasePath)'= '' then
$(Include=$(vG.BaseVariablePath)\3.SystemVariables.qvs);
endif
```

Adding additional scripts

Below are descriptions of optional scripts included after *1.init.qvs*.

System Variables

In this case *3.SystemVariables.qvs* that is used for loading System Variables. System Variables is technical variables like QlikView Server Log Path and is only needed for system monitoring. If QlikView Publisher and QlikView Server is not on the same server, the System Variables need to be changed so that they point to the correct servers and fold this is done by using Variable Editor. Read more in the Variable Editor section.

When needed the Global System Variables are loaded into QlikView with *3.SystemVariables.qvs* include script.

```
// System Variables (3.SystemVariables) Points to special system folders like QlikView Server Logs
$(Include=$(vG.BaseVariablePath)\3.SystemVariables.qvs);
```

Notice that *\$(vG.BaseVariablePath)* is used in the script to reach the include files, the applications home container variables are created by *1.init.qvs*.

Locale files

Contains include files with parameters that defines language, country and any special variant preferences that the user wants to see in their user interface. The locale global variable is *vG.localePath*.

Use this folder for custom Locale settings as well.

```
// Locale for English
$(Include=$(vG.LocalePath)\1.US.qvs);
```

Several additional scripts making life easier are available in the framework, read more in **Deployment Framework Core**.

Linking Containers together overview

By using the **LoadContainerGlobalVariables** function call it's possible to create Global Path Variable links to other containers in the *QlikView Deployment Framework* script tab.

Example: **call LoadContainerGlobalVariables ('AcmeT');** This function will create (based on the container Map) a complete list of Global Path Variables to 2.AcmeTravel (using the prefix AcmeT) container, the variables will have the same container name standard as inside a container but with an additional prefix, like *vG.AcmeTQVDPath* for the QVD path in AcmeT container.

Or this example: **call LoadContainerGlobalVariables ('Oracle','QVD;Include');**

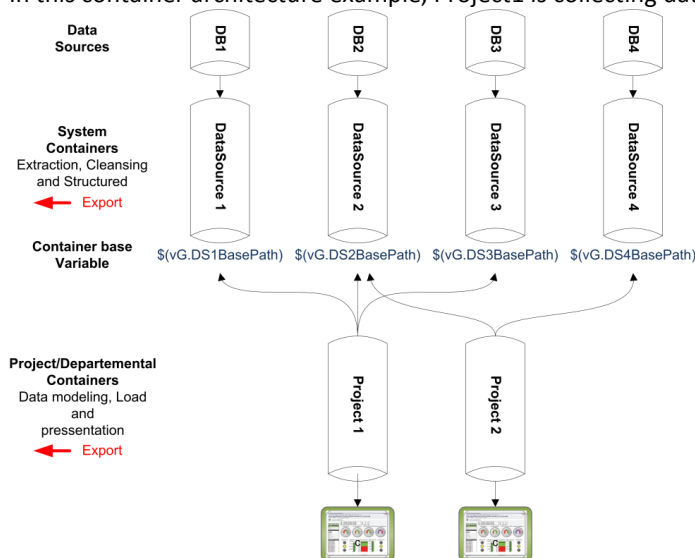
Will load selected Global Path Variables by use of additional switch and a ';' separator, in this case *vG.OracleQVDPath* and *vG.OracleIncludePath*

LoadContainerGlobalVariables uses the Container Map that is maintained by the Variable Editor (Tools section). Each Container have its own map copy, this approach makes it possible to restrict linking between containers by "hiding" containers in the map. **LoadContainerGlobalVariables** function will only work when the container map correlates to a physical container.

Read more under **Reuse of Script Code/Functions** section below in this document.

How to link Systems Containers into Project Containers

In this container architecture example, Project1 is collecting data from several system containers.



Here is an example how to create the Global Variable connections between containers.

1. Create new containers with the Administration Tool *VariableEditor.qvw* with Container Map view. Type the containers in *ContainerFolderName* table, in this example *98.System\1.DS*, *98.System\2.DS*, *98.System\3.DS*, *98.System\4.DS* this will eventually create four containers under a new System subfolder named *98.System*. (read more regarding Variable Editor in Tools section and in Operations Guide)
2. After enter prefix share variable names in *ContainerPathName* field, in this example *DS1*, *DS2*, *DS3*, *DS4*.
3. Then call the container (that should be linked) in the QlikView main script.
`call LoadContainerGlobalVariables ('DS1','QVD');`
4. Remember that *1.init.qvs* script always needs to be the first script to run.
5. Do the same with *2.DS*, *3.DS* and *4.DS* when needed.
6. After executing *LoadContainerGlobalVariables* function Global Path Variables that points to the selected container containers will be created, in this case the system containers: *\$(vG.DS1QVDPPath)*, *\$(vG.DS2QVDPPath)*, *\$(vG.DS3QVDPPath)*, *\$(vG.DS4QVDPPath)*

What in the framework are applications depended on?

The container structures are identical, as mentioned earlier but there could still be dependencies in the source container that needs to be copied into the destination container, like:

- Connection string, resided in *vG.ConnStringPath*
- Sub folders, example *\$(vG.QVDPPath)\1.Metadata*
- Custom global variables inside *\$(vG.BaseVariablePath)\CustomVariables.csv* that needs to be copied into the destination or be modified by **Variable Editor** in the new container. Example, copy the variable *\$(vG.MetadataQVDPPath)* (based on *\$(vG.QVDPPath)1.Metadata*)
- When running a *Binary* load statement, (if needed) change relative path in the script.
- Connections to Shared Folders container could need scripts/data from the original Shared Folders container
- Global Variable links (created by *LoadContainerGlobalVariables*) to other containers need to be added in the Container Map.

Deployment Framework Templates and Examples

The Administrative container contains several working templates and examples, this for easy start and to understand how to use the Framework in the best way. Examples and templates always reside under folders that start with 0, and are only stored in the 0.Administration container. These examples are:

- **Basic example how to Load Deployment Framework Include Init scripts.**
This is a basic example how the DF include scripts work, use this as a starting point when developing by using the Deployment Framework. Resides under *1.Application\0.Example\2.LoadIncludeExample*
- **QlikView Management Console (QMC) Table QVD Loader.**
This Template is used to extract the QMC Section Access Tables into QVD files.
This function is used as Section Access table storage but can be used as other data storage as well.
Resides under *1.Application\0.Example\3.QMC-TableQVDLoader*
- **QVD generator Template,** This template is loading data from Northwind and writing qvd's into *\$(vG.QVDPPath)\0.Example_Northwind* folder. Use this template as a quick start when developing qvd loaders. Resides under *1.Application\0.Example\1.QVD-Generator-example*
- **Data Export example,** how to export QlikView qvd data in txt or qvx format in *\$(vG.ExportPath)* folder. Resides under *1.Application\0.Example\5.Data-Export-example*
- **Calendar-Example** Master Calendar example that uses the CalendarGen sub function
Resides under *1.Application\0.Example\6.Calendar-Example*
- **QVD-Migration-example** QVD field copy example. Copy and scramble fields between Containers using QVDMigration sub function. Resides under *1.Application\0.Example\7.QVD-Migration-example*
- **Connection String Include example** to Nothwind Access database
\$(Must_Include=\$(vG.ConnStringPath)\0.example_access_northwind.qvs);
- **Qlik Mart example** based on Northwind and QVD generator qvd's from *\$(vG.QVDPPath)\0.Example_Northwind*
Resides under *\$(vG.MartPath)\0.Example_Northwind_Mart*
- **Northwind Access database,** Good to have when testing different scenarios.
Resides under *\$(vG.ConfigPath)\0.Example_Northwind*



QlikView Staging

QlikView staging is the process of intermediately storing data between the sources of information, most often in QVD files. Always use the template for QlikView Staging Application as starting point. In this application there are 3 tabs in the script as standard. These could be extended where needed.

The tabs should contain the following content:

- Main
 - Include statement for connection string ODBC/OLE DB stored in *vG.ConnStringPath*
 - Include statement for QVD-variables. An alternative to this is to use relative paths on each tab
 - Meta information about the application. I.E owner, purpose.
- Extract
 - Extracting the sources needed. Using an incremental approach when applicable. If there is no need for transformation the source could be stored directly to the presentation layer in the QVD-folder using this variable/relative path.
- Transform
 - When transformation is needed. For example creating new fields, cleansing information, aggregate and so on. This will be executed here.

Database Connection String

Connection strings are a security credential and should always reside in the architecture Back end, in the data tier. Well protected from unauthorized access. Remember that architecture front-end (QlikView Server and QlikView Web Server) does not have any open ports the back-end, these servers could be in different network zones and security boundaries. When distributing QlikView applications via the Publisher in Back end to the application tier, scripts and connection strings will automatically be removed.

Security considerations

By separating the connection string from the script reusability and higher security will be achieved.

There are two ways to separate and reuse the connection strings:

Include File

Best practice is to keep the connection strings in a separate Include file. This behavior is supported by Deployment Framework. Use the Global Variable *vG.ConnStringPath* to connect inside your container, example:

```
// Connection string to Northwind Access data source
$(Must_Include=$(vG.ConnStringPath)\0.example_access_northwind.qvs);
```

If the connection string is in another container, for example the Shared folders use the Global Variable

vG.SharedConnStringPath to connect, example:

```
// Connection string to Northwind Access data source
$(Must_Include Include=$(vG.SharedConnStringPath)\0.example_access_northwind.qvs);
```

Recommendation is to use **Must_Include** so that the QlikView script will fail if the connection string is missing.

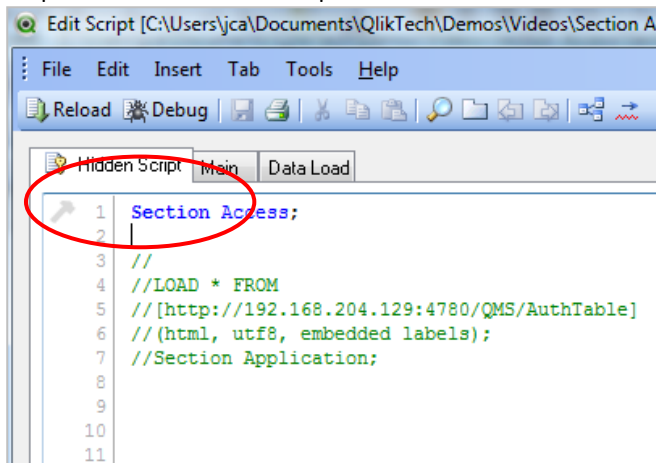
Global Variable

By using the Global Variable Editor there is the possibility to create a Global Variable representing the connection string. This method is not as secure as using an include file. Include files can be secured by different security groups this is not possible when using Global Variables that will be used by all the applications within a Container. But when a container is secured and dedicated for a source system (example Oracle container) connection strings as global variables could be used.

Security Tab (Hidden Script)

In QlikView it is possible to restrict the privileges of a document user from the **Document Properties: Security** and the **Sheet Properties: Security pages**. Any settings can be altered if the document user is logged in as ADMIN.

The user identity and password needed for opening a user restricted document are specified in the load script and will show up in the log file if you allow QlikView to generate one. However, by having the user access in the hidden script instead, the log file will not give away any login information. The Hidden Script button opening the hidden script is found in the Edit Script menu.



QlikView Security Table (Section Access)

In QlikView the security settings of the QlikView file is set in the script. Access rights and User Levels are defined in the Section Access part of the script. Section Access can be used to set access restrictions to data, sheets and sheet objects. All access control is managed via files, SQL databases or inline clauses in the same way as QlikView normally handles data. If an access section is defined in the script it must be followed by the statement Section Application in order to load normal data.

Section Access system fields

Access levels are assigned to users in one or several tables loaded within the Section Access. These tables can contain several different user-specific system fields, typically NTNAME and the field defining the access level, ACCESS. The full set of section access system fields are described below. Other fields like GROUP or ORGANISATION may be added to facilitate the administration, but QlikView does not treat these fields in any special way. None, all, or any combination of the security fields may be loaded in the access section. However, if the ACCESS field is not loaded, all the users will have ADMIN access to the document and the section access will not be meaningful.

Section access system fields are:

- **ACCESS** A field that defines what access the user should have.
- **NTNAME** A field that contains a string corresponding to a SSO user name or group name.
- **USERID** A field that contains a user ID that has the privilege specified in the field ACCESS.
- **PASSWORD** A field that contains an accepted password.
- **SERIAL** A field that contains a number corresponding to the QlikView serial number.
- **NTDOMAINSID** A field that contains a string corresponding to a Windows NT Domain SID
- **NTSID** A field that contains a Windows NT SID.

Section Access in Combination with Publisher

While QlikView Publisher can use its “loop and reduce” functionality to reduce a QVW by rows by user or group as it is being reloaded, you can also accomplish this in Section Access dynamically as the document is opened, either method will work and both have benefits. The Loop and reduce from Publisher will help you to reduce the memory footprint of the QVWs on your server(s), while the Section Access method is portable with the document. Another reason to use Section Access is the application of authentication in the QVW, through SSO or user ID and password. This is especially important if the QVW is going to be enabled for download from the AccessPoint or otherwise distributed to users.

Best Practices when using Section Access:

- In Section Access, always use the Upper() function when utilizing a load statement, use it on every column no matter what. (even when reading from .qvd)
- AD Groups for security if possible
- Security in include files for reuse
- Add the Publisher’s service account to the Section Access table
- Utilizing a ‘Star Schema’ design for the data model with NO LINK Tables.
Link tables hurt performance greatly!
- Best case is to have 1 fact table with the dimensions all directly connected to the fact.
In rare instances should additional ‘snowflake’ dimensions to be used.
- In the fact tables, have no more than 30 – 40 columns defined.(there can be a few more/less, but do not have 150 columns unless you fact is less than 10 Million records (with a decent server)
- Many times having too many columns are a situation brought on by utilizing ‘Role Playing Metrics’. While this may be helpful, too many of these metrics create a performance degradation on the server.

Reuse of script code

For easier manageability and faster development it's recommended to reuse script code as much as possible. By using Deployment Frameworks predefined structures and variables it's easy to reuse script code. There are two ways of reusing code in QlikView Script:

- Include files
- Use of functions

Include files

An include file is just a QlikView script (text file) that is included into the main script during execution. QlikView include scripts use the prefix *qvs*. The entire or parts of the script can thus be put in a file for reuse.

All Include files are stored in *6.Custom* folder, the global variable for *6.Custom* folder is *vG.CustomPath* and should always be used when accessing a custom script, meaning that it's not a part of the Deployment Framework initiation process. Example: `$(Include=$(vG.CustomPath)\1.xyz_Calculations.qvs);`

Sub Functions

QlikView have the possibility of reusing scripts and functions, a good way of reusing code is by using the [Sub](#) and [Call](#) commands. All sub functions are stored in *4.Sub* folder and are included in the script start right after the *1.Init* script. Use `Call function_name('Input parameters or variables')` command to execute the preloaded function.

The global variable for *4.Sub* folder is *vG.SubPath* and should always be used when accessing a function. Example:

- 1) Include *1.FileExist.qvs* containing Sub function in the script beginning: `$(Include=$(vG.SubPath)\1.FileExist.qvs);`
- 2) Call *FileExist* function when needed in the script: `call vL.FileExist ('$(vL.MetadataQVDPPath)\SOE.qvd')`

Instead of loading in sub functions when needed you can load them all with one single include (*99.LoadAll.qvs*).

Example first tab includes the script: `$(Must_Include=$(vG.SubPath)\99.LoadAll.qvs)`

Another function example is [LoadContainerGlobalVariables](#) that is used to create Global Variable links between containers.

The predefined Sub functions that exist in the *1.Sub* folder should not be deleted or modified the subs is used by Deployment Framework initiation process and tools like Variable Editor.

Hint. Use the QlikCommunity to find sub function examples, instead of coding everything from scratch

Pre-Defined Functions

Down below is a list of the predefined Sub functions available in Deployment Framework.

1. *FileExist.qvs*

SUB routine will check if a file exists, use before load to avoid errors during script load.

vL.FileExist returns *true* or *false*. First include the script: `$(Include=$(vG.SubPath)\1.FileExist.qvs);`

Syntax example in the script: `call vL.FileExist('$(vL.CVSTableName)'`

Example, action exit script after check:

```
call vL.FileExist ('$(vL.MetadataQVDPPath)\1.NorthWind\*');
```

Will Check if *1.NorthWind* folder exists and return *vL.FileExist = true* or *false*

```
call vL.FileExist ('$(vL.MetadataQVDPPath)\SOE.qvd');
```

```
if vL.FileExist = 'false' then; trace '### Did not find file, exit script'; exit script; endif;
```

2. *LoadVariableCSV.qvs*

SUB routine used for loading variables stored in csv files into the QlikView Script.

This file is used by *1.Init* to load Custom Global Variables.

First include the script in the beginning: `$(Include=$(vG.SubPath)\2.LoadVariableCSV.qvs);`

Execute (Call) the Sub inside the script:

```
call LoadVariableCSV('My Variable File.csv', ['Optional Search Tag', 'Optional Container Map Mode' 'Optional Comments as variables]);
```

- **My Variable File** Is the VariableFile name to load, usually ends with Variable.csv, the function will try to find the first variable file in `$(vG.BaseVariablePath)` your container and second in `$(vG.SharedBaseVariablePath)` in the shared container
- **Variable Tag** will load variables based on Search Tag's created in the variable editor
- **Container Map Mode** is a special mode to create variables based on the Container Map, this is used by the Variable Editor.
- **Comments as variables** will create a `_comment` variable for every real variable (if comments exist), this is nice way to add meta-data into expressions. *Comments as variables* can also be activated by setting the variable `SET vL.CommentsAsVariables=True;` before the *1.Init.qvs* Initiation script. Use this to get comments from *CustomVariables.csv* file loaded by default.

Examples:

Load variables from a CSV file stored in `$(vG.BaseVariablePath)`: `call LoadVariableCSV('MyVariables.csv')`

Load variables with HR tag from a CSV file: `call LoadVariableCSV('MyVariables.csv','HR')`

Load variables and Variable Comments `call LoadVariableCSV('MyVariables.csv','','True')`

Set Container Map as Base Variables from a CSV file (Advanced):

```
call LoadVariableCSV('$(vG.BaseVariablePath)\ContainerMap.csv','','true')
```

3. *LoadContainerMap.qvs*

SUB routine used for loading a Container Map csv file. This routine will return information for a specific container.

These parameters are used when creating Global Variables to Link a container.

The *LoadContainerMap* function is used by the *4.GenericContainerLoader.qvs* script and by the *LoadContainerGlobalVariables* function.

First include the script in the beginning: `$(Include=$(vG.SubPath)\3.LoadContainerMap.qvs);`

Execute (Call) the Sub inside the script, example:

```
SUB LoadContainerMap ('Container Map file', 'Container name', [' Optional $(vG.BasePath)']);
```

Load Container Map returns these variables:

vL.ContainerFolderName This is the Container folder name

vL.ContainerPathName This is the Container prefix name

vL.RootPath This is container path *vG.RootPath* or alternative Path

vL.Comment Comments regarding the container

vL.LoadContainerMapCount Returns a result (Variable prefix name) only if Variable prefix duplication found. This so that Variable Editor can alert operator to remove duplication.

The third switch *\$(vG.BasePath)* is optional and specially designed to identify Root Path (*vG.RootPath*) during initiation (1.Init). This is done by opening the container map and checking where I am and where the Root Path is in relation to my container? The value must be global variable base path (*vG.BasePath*). If this process fails the Root Path will be set to one folder above your container. When using this switch the *Container name* value is not needed.

4.GenericContainerLoader.qvs

The [LoadContainerGlobalVariables](#) function creates Container Global Variable links to other containers based on the Container Map. SUB routine loading Container link Global Variables into QlikView Script. This routine is intended to be used inside the QlikView scripts and is designed for easy use. 4.GenericContainerLoader.qvs is a SUB that is loaded during 1.init initiation phase but is not used until the [LoadContainerGlobalVariables](#) function is called.

The function will exit without mapping if the physical container is missing. *Container Path Name* created and maintained by the Variable Editor is a mandatory value.

There is also a short name for [LoadContainerGlobalVariables](#) available named [LCGV](#) that will work the same

Execute (call) the Sub function inside the script,

Call LoadContainerGlobalVariables ('Container Path Name', [Optional Single Folder [;Additional folders separated by ;]] [Optional Use Shared Folder Container Map]);

Example 1, loading all Container link Global Variables to 2.AcmeTravel:

```
call LoadContainerGlobalVariables ('AcmeTravel'); or
call LCGV ('AcmeTravel');
```

Example2, loading a single Global Container Variable, in this case Acme Travel QVD path (*vG.AcmeTravelQVDPath*).

```
call LoadContainerGlobalVariables ('AcmeTravel','QVD'); or
call LCGV ('AcmeTravel','QVD');
```

Example 3, load several Global Container Variables by use of ';' as separator,

in this case *vG.OracleQVDPath*, *vG.OracleIncludePath* and *vG.OracleApplicationPath*

```
call LoadContainerGlobalVariables ('Oracle','QVD;Include;Application'); or
call LCGV ('Oracle','QVD;Include;Application');
```

Example 4, loading Container link Global Variables to 2.AcmeTravel based on Container Map in Shared Folders:

```
call LoadContainerGlobalVariables ('AcmeTravel','','true'); or
call LCGV ('AcmeTravel','','true');
```

5.DoDir.qvs

DoDir is a simple to use but powerful function that will index selected folder/file structure and return a Table containing file name and path under selected file system. First include the script: *\$(Include=\$(vG.SubPath)\5.DoDir.qvs);*

After execute (call) the Sub DoDir inside the script.

Call DoDir (Scan Path, Table Name, [Folders Only], [Single Folder], [Qualified Felds])

Scan Path It the folder path to scan

Table Name Is the Table name, optional default name is *DoDirFileList*

Folders Only Is an optional switch if set to 'true' only folders will be returned

Single Folder Is an optional switch if set to 'true' only one single folder will be indexed

Qualified Felds Is an optional switch if set to 'true' all field named will be Qualified *based on the Table Name*

Examples:

- *call DoDir ('\$(vG.IncludePath)'); //Simple Example list files in vG.IncludePath*
- *call DoDir ('\$(vG.IncludePath)*.qvs'); //Will only return only files with file type qvs under vG.IncludePath*

QlikView

- `call DoDir ('$(vG.IncludePath', 'IncludeFileTable'); //Change Table name to IncludeFileTable`
- `call DoDir ('$(vG.IncludePath', '', 'true'); //Returns only folder names under vG.IncludePath`
- `call DoDir ('$(vG.QVDPPath\HR.qvd'); //Returns a table for a single file only`

Table and fields that is returned:

FullyQualifiedName	DoDirFileSize	DoDirFileTime	DoDirFileName	DoDirContainerPath	DoDirFileExtension...
C:\Users\mbg\Docume	46	2012-11-13 19:34:36	desktop.ini	desktop.ini	INI
C:\Users\mbg\Docume	36870	2012-11-13 22:05:38	Folder.ico	Folder.ico	ICO
C:\Users\mbg\Docume	418	2013-03-18 16:08:20	Info.txt	Info.txt	TXT
C:\Users\mbg\Docume	767	2012-10-08 15:42:34	InitLink.qvs	InitLink.qvs	QVS
C:\Users\mbg\Docume	10548	2013-08-21 16:46:55	Version1.3.txt	Version1.3.txt	TXT
C:\Users\mbg\Docume	420	2013-03-18 16:07:40	Info.txt	0.Template\Info.txt	TXT

- **FullyQualifiedName** is the file name and complete path
- **DoDirFileSize** is the file size
- **DoDirFileTime** is file date and time
- **DoDirFileName** is the File Name without path
- **DoDirContainerPath** lists the files in relationship with the current container
- **DoDirFileExtension** Contains the File Extension in upper case, perfect to use when searching for types

6.CreateFolder.qvs

Create Folder function will -as the name says- create a folder (if non existing) or a folder structure.

First include the script: `$(Include=$(vG.SubPath)\6.CreateFolder.qvs);`

After execute (call) the Sub CreateFolder inside the script.

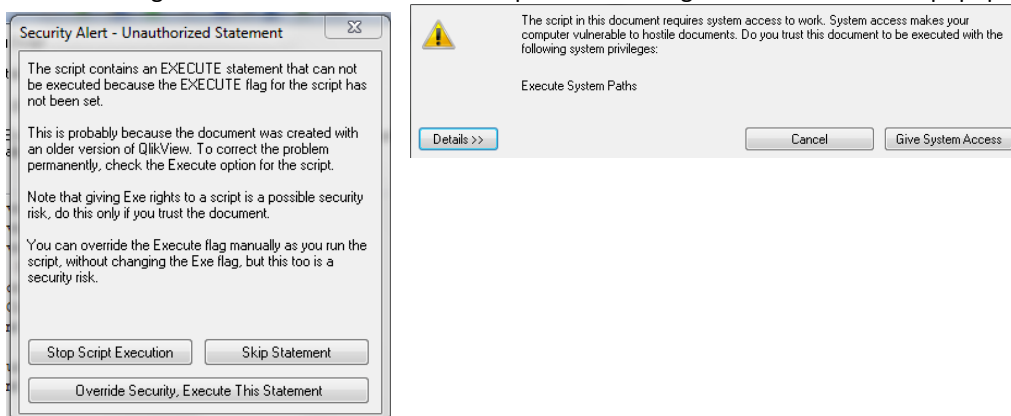
`sub CreateFolder (vL.FolderName)`

`vL.FolderName` Is the folder name or folder structure to create

Examples:

- `call CreateFolder (' $(vG.QVDPPath)\NorthWind'); // Will create NorthWind folder under vG.QVDPPath`

When executing this function in QlikView Developer and creating a folder one of these popup boxes will appear:



Press Override Security to execute the folder creation, next run the folders are already created and the box will not return.

7.CalendarGen.qvs

Calendar Generation Function created by Jonas Valleskog and functions added by QlikTech.

Generic calendar generation script that enables scalable handling of creating and navigating multiple date fields

Implementation instructions

In the script editor, Deployment framework tab include the sub: `$(Include=$(vG.SubPath)\7.CalendarGen.qvs);`

Call the SUB function (once per date field) after table load statements.

`CALL CalendarGen('Date Field', 'Calendar Table' [, 'Months Left Fiscal Date'] [, 'Min Date', 'Max Date'] [, 'Link Table'];`

- **Date Field** is the date field to link calendar. Generated Calendar is based on this field
- **Calendar Table** (Optional) is the master calendar table name default is the same name as *Date Field*
- **Months Left Fiscal Date** (Optional) to activate Fiscal Dates, set no of months left of the Calendar year the month the Fiscal year begins. E g '3' if the first month of the Fiscal year is October.
- **Min Date Optional** Set hard Minimum calendar date ex. '11/7/1996' (depending on locale settings)
- **Max Date Optional** Set hard Maximum calendar date ex. '8/13/1999' (depending on locale settings)
- **Link Table** (Optional) By default link table is identified based on *Date Field* use this setting if need to override

Examples:

`CALL CalendarGen('OrderDate');`

`CALL CalendarGen('OrderDate', 'OrderDateCalendar', '3'); // Fiscal Dates`

`CALL CalendarGen('OrderDate', 'OrderDateCalendar', '', '11/7/1996', '8/13/1999'); // Min and Max date`

The sub function will return table with the standard fields below:

- **Table Name** – The *Date Field* table name used as key field to data model
- **Table Name Week** – Week number field Ex. 32,33,34
- **Table Name Year** – Year field Ex. 2001, 2002
- **Table Name Month** – Month field Ex. Jul, Aug
- **Table Name Day** – Day number field Ex. 1,2,3,4
- **Table Name WeekDay** – Weekday short name field Ex. Mon, Tue, Wen
- **Table Name Quarter** – Quarter field Ex Q1, Q2, Q3, Q4
- **Table Name MonthYear** – Concatenated month and year field Ex. 08-2002, 09-2002
- **Table Name QuarterYear** – Concatenated quarter year field Ex. Q3-2002, Q4-2002
- **Table Name WeekYear** – Concatenated week year field Ex. 32-2002, 33-2002
- **Table Name YTD Flag** – Year to Date Flag field shows 1 if current year
- **Table Name PYTD Flag** – Past Year to Date flag field shows 1 if last year
- **Table Name CurrentMonth Flag** – Current Month flag shows 1 if historical month is same as current month
- **Table Name LastMonth Flag** – Last Month flag shows 1 if historical month is same as last month
- **num Table Name** – Autounumber field based on rows ex. 1,2,3,4,5,6...700,701,702
- **Table Name numMonthYear** – Autounumber field based on MonthYear field ex. 2, 28, 59, 89
- **Table Name numQuarterYear** – Autounumber field based on QuaterYear field ex. 2, 89, 181
- **Table Name numWeekYear** – Autounumber field based on WeekYear field ex. 2, 4, 11, 18, 25

Tips and tricks:

- Check out 6.Calendar-Example to get inspiration. Copy or re-create the calendar objects (time related list boxes) laid out in the front-end of the example QVW file.
- Use *DateFormat* variable when formatting date, this creates flexibility when changing locale.
ex. `Date(OrderDate, '$(DateFormat)') AS OrderDate`
- To avoid potentially slow queries against large in-memory tables, contemplate storing out the date field to QVD first and use the QVD store as the input source to the MinMax: table creation.
- If gaps in calendars for missing dates are not an issue, consider replacing AUTOGENERATE() logic for generating the calendar table with a distinct list of each date seen in the source table instead.

8.QVFileInfo.qvs

QVFileInfo sub function returns information (in table format) regarding QlikView files that stores metadata (QVW and QVD). First include the script: `$(Include=$(vG.SubPath)\8.QVFileInfo.qvs);`
After execute (call) the Sub *QVFileInfo* inside the script.

Call *QVFileInfo*('Fully Qualified file Name', ['Table Name'])

Fully Qualified file Name is the path and name of qvd or qvw file. *Table Name* (Optional) is name of the table returning the result default table name is *QVFileInfo* linked with *QVFileInfo_field* (field details table)

Examples:

```
call QVFileInfo('$(vG.QVDPATH)\Customer.qvd') // Will get MetaData regarding Customer.qvd
call QVFileInfo('$(vG.QVDPATH)\Customer.qvd','QVFileTable')
```

Table *QVFileInfo* contains table and file information regarding QVD and QVW files:

- **FullyQualifiedName** is the file name and complete path, use as link to *DoDir* Table
- **QVTablesKey** Table link key to *QVFileInfo_Fields* table
- **QVTableName** Name of tables in an QVW file or name of Table in a QVD file
- **QVFileTime** Data reload date
- **QVTableNbrRows** Total number of rows in *QVTableName*
- **QVTableNbrFields** Total number of fields in *QVTableName*
- **QVTableNbrKeyFields** Total number of Key fields in *QVTableName* only used by QVW files
- **QVTableComment** Table Comments, only used by QVW files

QVFileInfo_Fields is a help table, containing Field information regarding QVD and QVW files:

- **QVTablesKey** Table link key to *QVFileInfo* table
- **QVFieldName** Name of Fields in a Table
- **QVFieldComment** Field Comments, only used by QVW files

It's best used in combination with *DoDir* function that will index the QlikView files and use *FullyQualifiedName* field as link to the *QVFileInfoTable*. This is an example of *DoDir* and *QVFileInfo* functions working together:

```
$(Include=$(vG.SubPath)\8.QVFileInfo.qvs);
$(Include=$(vG.SubPath)\5.DoDir.qvs);
call DoDir('$(vG.BasePath)');
for vL.LoopDoDirRows = 1 to NoOfRows('DoDirFileList')
    LET vL.FullyQualifiedName = peek('FullyQualifiedName', $(vL.LoopDoDirRows), 'DoDirFileList');
    call QVFileInfo ('$(vL.FullyQualifiedName)');
next
```


9.QVDMigration.qvs

QVDMigration sub function migrates and consolidates qvd data between containers, using fixed file names or wild-card (*) migrating a qvd folder in one single statement. QVDMigration can optionally migrate selected fields and scramble fields if needed. The sub function is primarily designed for data migration into a self-service (sandbox) environment. Needed subfolders in destination path will automatically be created by use of *CreateFolders* function.

First include the script: `$(Include=$(vG.SubPath)\9.QVDMigration.qvs);`

Execute (call) the Sub function inside the script,

Call QVDMigration (QVD Source File, QVD Destination File, [Select specific fields (, separator) leave blank for all fields], [Scrambled fields (, separator)], [Table Name Suffix], [Include Subfolders]);

QVD Source File is the QVD source file or folder

QVD Destination File is QVD destination path. Optionally, to rename file add filename

Fields to select (Optional) used when selecting specific fields from the Source QVD. Multiple fields are separated with (,).

Scrambled fields (Optional) used when scrambling fields from the Source QVD. Multiple fields are separated with (,). Scramble overrides **Fields to select** parameter if dual entries found. Scrambling will have performance impact so carefully select fields to scramble.

Table Name Suffix (Optional) primarily used as meta-data separator between source and destination this by adding a suffix on the destination qvd table names. The difference will be exposed in Governance Dashboard as shown below.

TableName	QVD\QVX
Customer	C:\QV-Docs\SourceDocs\1.Production\0.Administration\2.QVD\Customer.qvd
	C:\QV-Docs\SourceDocs\1.Production\1.AcmeSales\2.QVD\Customer.qvd
Customer-Shared	C:\QV-Docs\SourceDocs\1.Production\99.Shared_folders\2.QVD\Customer.qvd

*Separating Table Name (Meta Data) by using **Table Name Suffix**, shown in Governance Dashboard*

Include Subfolders (Optional) If set to true subfolders under Source Files will also be migrated, needed subfolders in destination path will automatically be created by use of *CreateFolders* function

Examples:

Migrate Customer.qvd to shared QVD folder without any manipulation

```
call QVDMigration ('$(vG.QVDPPath)\Customer.qvd','$(vG.SharedQVDPPath)');
```

Migrate Customer.qvd to shared QVD folder and changing name to Customer_new.qvd

```
call QVDMigration ('$(vG.QVDPPath)\Customer.qvd','$(vG.SharedQVDPPath\Customer_new.qvd)');
```

Migrate fields CustomerID and CompanyName in all Customer*.qvd files to shared QVD folder

```
call QVDMigration ('$(vG.QVDPPath)\Customer*.qvd','$(vG.SharedQVDPPath)','CustomerID,CompanyName');
```

Migrate fields CustomerID and CompanyName in Customer.qvd to shared QVD folder scramble CustomerID field

```
call QVDMigration ('$(vG.QVDPPath)\Customer.qvd','$(vG.SharedQVDPPath)', 'CustomerID,CompanyName','CustomerID');
```

Migrate all Customer qvd files to shared QVD folder, scrambling CustomerID field in all the qvd's

```
call QVDMigration ('$(vG.QVDPPath)\Customer*.qvd','$(vG.SharedQVDPPath)\Customer.qvd','', 'CustomerID');
```


10.QVDLoad.qvs

QVDLoad will load up qvd files into a data model based on the meta-data headers in the qvd files. Also qvd files stored in subfolders can optional be loaded. QVDLoad is based on *QVDMigration* and have the same code and switches except for destination path.

First include the script: `$(Include=$(vG.SubPath)\10.QVDLoad.qvs);`

Execute (call) the Sub function inside the script,

Call QVDLoad(QVD Repository, [Select specific fields (, separator) leave blank for all fields], [Scrambled fields (, separator)], [Table Name Suffix], [Include Subfolders]);

QVD Repository is the QVD source file or folder storage

Fields to select (Optional) used when selecting specific fields from Repository. Multiple fields are separated with (,).

Scrambled fields (Optional) used when scrambling fields from Repository into the application. Multiple fields are separated with (,). Scramble overrides **Fields to select** parameter if dual entries found. Scrambling will have performance impact so carefully select fields to scramble.

Table Name Suffix (Optional) will add a suffix on all tables in the data model

Include Subfolders (Optional) If set to true qvd files in subfolders will also be loaded

Examples:

Load in all qvd files in vG.QVDPATH folder and create a data-model based on table headers

```
call QVDLoad('$(vG.QVDPATH)');
```

Load in all qvd files stored in every subfolder under vG.QVDPATH

```
call QVDLoad('$(vG.QVDPATH)',",",",",true');
```

Load in fields CustomeID and CompanyName in all qvd files.

```
call QVDLoad('$(vG.QVDPATH)','CustomeID,CompanyName');
```

Loads fields CustomeID and CompanyName and scramble CustomerID field from Customer.qvd

```
call QVDLoad('$(vG.QVDPATH)\Customer.qvd','CustomerID,CompanyName','CustomerID');
```

Migrate all Customer qvd files to shared QVD folder, scrambling CustomerID field in all the qvd's

```
call QVDMigration('$(vG.QVDPATH)\Customer*.qvd','$(vG.SharedQVDPATH)\Customer.qvd','',CustomerID');
```

99.LoadAll.qvs

Simple include scrip that will load in all available sub functions in one single go. Recommendation is to use

Must_Include so that the script breaks if *99.LoadAll.qvs* is missing.

In first tab include the script: `$(Must_Include=$(vG.SubPath)\99.LoadAll.qvs)`

To simplify the script initiations even more `$(Must_Include=$(vG.SubPath)\99.LoadAll.qvs)` can be put inside

4.Custom.qvs file (in Shared Folders Container) so that it will be run during *1.Init* initiation for all applications.

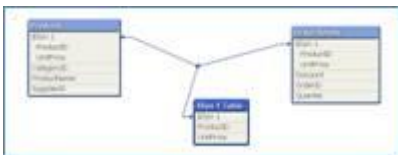
Data Modeling

Understanding

The cornerstone of QlikView is the associative in-memory search technology.

There are some very specific characteristics with this technology that you have to keep in mind.

- Two fields in different tables with exactly the same name, case sensitive, will automatically be connected to each other and fields with exactly the same field value, case sensitive, will be associated with each other.
- If two tables have more than one field in common, QlikView will automatically create a synthetic key a kind of link table. The easiest way to detect a synthetic key is by opening the table viewer (Ctrl-T):



Synthetic key

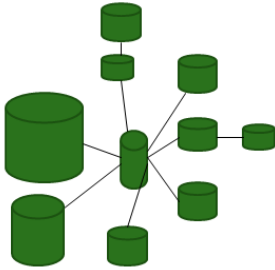
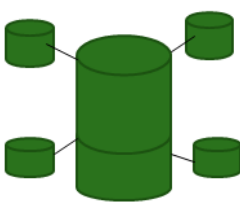
















Another characteristic with the associative database is that the number of distinct (unique) values in a table is more important than the number records. By delimit the number of distinct values in a table the performance of an application can be significantly improved.

Example: Let's say you have a fact table with 1 billion recs, one of the fields is a timestamp field containing date and time (measured down to fraction of seconds) with almost 800 million distinct values. Two alternative actions will both improve the performance:

- If you don't need to analyze on time level, simply transfer the field to a date field (use makedate function) and there will not be more than 365 distinct values for one year.
- If you need to analyze on time level, determine on what time level you need to analyze (hour, minute) and create a new field, Time. Depending on what level you decide to analyze, hour will give you 24 distinct values and minute will give maximum 1440 distinct values)

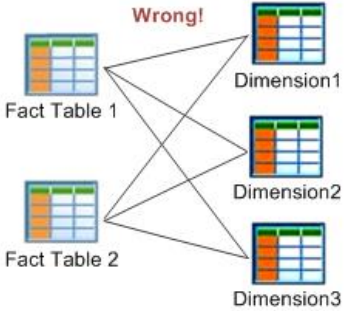
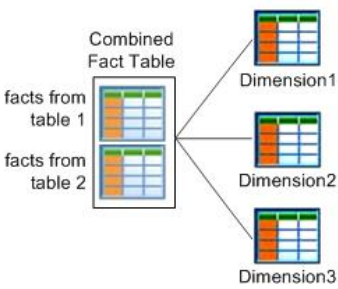
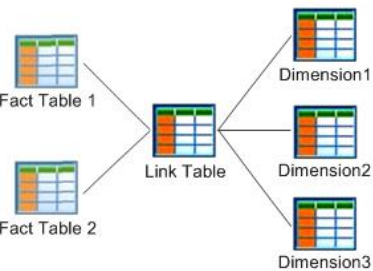
Data models

Represented below are diagrams of 3 basic data models that can be built in QlikView (along with many other combinations). Using these 3 examples we can demonstrate some of the differences in performance, complexity and flexibility between them.

Option 1 Snowflake	Option 2 Star Schema	Option 3 Single Table
		
Response Time 		
RAM consumption 		
Script run time 		
Flexibility Model 		
Complexity Script 		

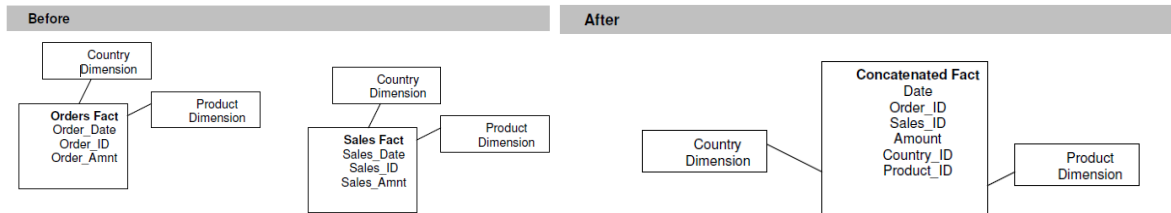
Multiple fact tables

While star schemas are generally the best solution for fast, flexible QlikView applications, there are times when multiple fact tables are needed. Here are the wrong and right ways to join them:

Option 1, Wrong Does not Work!	Option 2, Right Concatenated Fact table Recommended by QlikTech	Option 3, Right Link Table Works on small data sets
		

Further examples of how to build and use link tables are contained in QlikCommunity on line (<http://community.qlikview.com/>)

To show how this could be accomplished, the section below takes us through a scenario of two facts tables to be combined into one fact table.



Script Example:

```
Load OrdersFact
    Order_Date as Date
    Order_ID
    Order_Amount as Amount
    Country_ID
    Product_ID
    'Order' as TransactionType
```

```
CONCATENATE
Load SalesFact
    Sales_Date as Date
    Sales_ID
    Sales_Amount as Amount
    Country_ID
    Product_ID
    'Sale' as TransactionType
```

Placing the 'Sale' and 'Order' text types in the script will provide you with a column to determine the transaction type.

Sales

Region	Product	Date	Sales
RegionA	P1	2009-01-31	100
RegionA	P1	2009-02-28	120
RegionA	P1	2009-03-31	140
RegionA	P2	2009-01-31	500
RegionA	P2	2009-02-28	550
RegionA	P2	2009-03-31	600
RegionB	P1	2009-01-31	50
RegionB	P1	2009-02-28	55
RegionB	P1	2009-03-31	60
RegionB	P2	2009-01-31	200
RegionB	P2	2009-02-28	180
RegionB	P2	2009-03-31	160

Plan Yearly

Region	Date	Plan
RegionA	2009-01-1	8000
RegionB	2009-01-1	10000

Procurement Cost

Product	Date	Cost
P1	2009-01-31	130
P1	2009-02-28	1400
P1	2009-03-31	1600
P2	2009-01-31	500
P2	2009-02-28	650
P2	2009-03-31	600

Concatenated Facts

Region	Product	Date	Sales	Plan	Cost
RegionA	P1	2009-01-31	100		
RegionA	P1	2009-02-28	120		
RegionA	P1	2009-03-31	140		
RegionA	P2	2009-01-31	500		
RegionA	P2	2009-02-28	550		
RegionA	P2	2009-03-31	600		
RegionB	P1	2009-01-31	50		
RegionB	P1	2009-02-28	55		
RegionB	P1	2009-03-31	60		
RegionB	P2	2009-01-31	200		
RegionB	P2	2009-02-28	180		
RegionB	P2	2009-03-31	160		
RegionA		2009-01-1		8000	
RegionB		2009-01-1		10000	
	P1	2009-01-31			130
	P1	2009-02-28			1400
	P1	2009-03-31			1600
	P2	2009-01-31			500
	P2	2009-02-28			650
	P2	2009-03-31			600

A concatenation of fact tables example.



Preceding Loads

The use of preceding load statements can simplify your script and make it easier to understand. See the code below for an example of this.

Table1:

```
LOAD CustNbr as [Customer Number],
      ProdID as [Product ID],
      floor(EventTime) as [Event Date],
      month(EventTime) as [Event Month],
      year(EventTime) as [Event Year],
      hour(EventTime) as [Event Hour];
```

```
SQL SELECT
      CustNbr,
      ProdID,
      EventTime
FROM MyDB;
```

This will simplify the SQL SELECT statement so that the developer can continue to test/augment the statement using other tools, without the complexity of the QlikView transformations embedded in the same SQL statement.

For more information on the Preceding LOAD feature, see the QlikView Reference Manual.

Large Data Sets

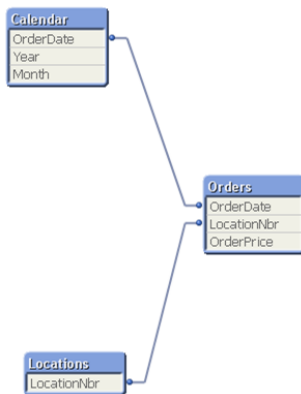
QlikView can handle very large data sets and routinely does so. However, to optimize the user experience and hardware needed, you have options.

Consider the following scenario: You have a large orders data set (1 billion rows). You need to provide high level summary metrics for your executives, trending analysis for your Business Analysts, and detail tables and values for your Orders Processing team. You have many data design options with QlikView, but for demonstration purposes let's explore just 3 of them below:

Detailed fact table only – allow QlikView to do all of the work to display the details and summarize metrics from the lowest level of detail to the highest summary needed.

Advantages – simplicity. This is the easiest solution to code. You simply connect the Orders at a detailed level (perhaps SKU level) to the data model and design all of the high level metrics, trending charts and detailed tables and selections into the QVW.

Disadvantages – QlikView will need to aggregate up to 1 billion rows of detail with every selection made. While QlikView is probably the only BI tool that can do this with acceptable performance, it will still result in a slower user experience than it needs to.



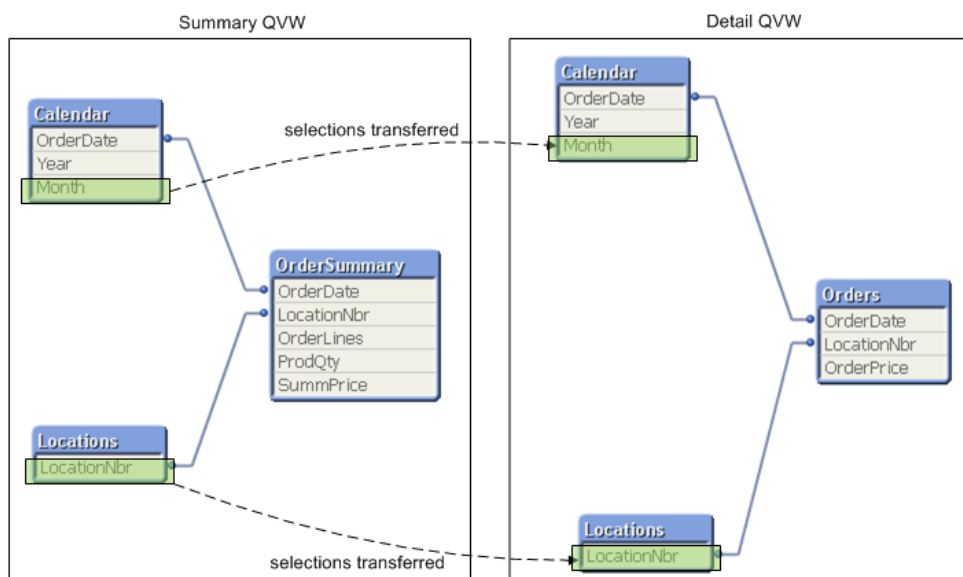
Document Chaining – 2 (or more) versions of the QVW are built. One of them has the detailed Orders table as the primary fact table the others have pre-aggregated versions of the Orders table as their primary fact tables. Let's assume just 2 QVWs for this case. You have a diagram below showing the data model from the "summary" QVW and a data model from the "detail" QVW. Note that the dimension values are largely the same between the two models. The main distinction is the fact table in the data model. The users can start from the summary application, showing high level metrics and charts.

If they want to drill into details you can use the Document Chaining feature in QlikView to transfer selections from one QVW to another QVW and open that second QVW. The user will see new charts and tabs show up and (if you design it as such) doesn't even need to know they have transferred from one QVW to another. This means you will only be using the 1 billion row fact table **when your users need it**. The rest of the processing will take place on the pre-aggregated version of the Orders table, which might be smaller than 100 million rows, for example. Document Chaining is discussed in detail in the QlikView Reference Manual and in several QlikView documents.

Advantages – optimizes hardware and speed of response for QlikView navigation and charting. Because the users' selections and navigation are specific to their needs, you don't waste CPU and RAM processing 1 billion rows of detail when the user didn't need things processed at that level.

Disadvantages – tables (QVDs) need to be pre-aggregated and maintained for this approach. While this is a one-time development effort, it is slightly more complex than option 1, where only one version of the Orders table is needed.

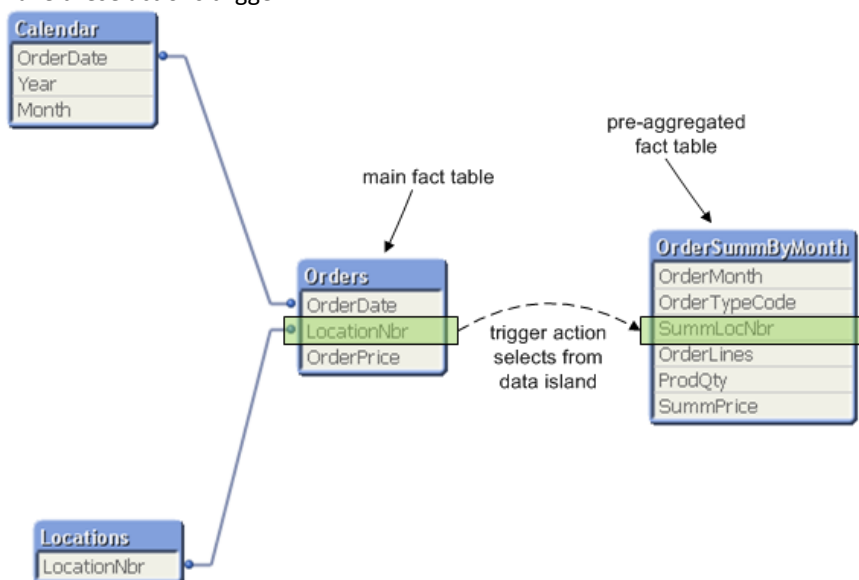
QlikView



The 3rd option (and by no means the last) is to use a pre-aggregated summary table **in addition to** the detailed table in a single QVW data model. The diagram shown below is one way to use a pre-aggregated table in the same data model as the detailed version of the table. You would load the pre-aggregated table as a data island (not connected to the other tables in the data model). Then, as relevant selections in the detailed fact table are made you can transfer those selections to the pre-aggregated table using a triggered Action (QlikView version 9 and above).

Advantages – this option doesn't require a second QVW and document chaining in order to use both detailed and summary versions of a large table.

Disadvantages – this option will require some settings to be made in the QVW to trigger the actions that transfer selections from one table to another. As the QVW changes over time, you will need to keep track of where/when to make these actions trigger.



Please note: these are many more ways you could meet the needs described in the above scenario. These are just 3 methods that call out the features and capabilities of QlikView to manage very large data sets. Please see the Architecture Best Practices Guide for more examples of ways to manage large data sets and large deployments of QlikView in an optimal way.

Key factors that affect the model:

Distinct column data.

Distinct key field information.

Both can affect the memory size of the Data Model and the user experience. By having many tables, the links can become a memory hog. It has been known that you can reduce your memory foot print by fifty percent when modifying the data structure; and thus, additionally increasing the UI response.

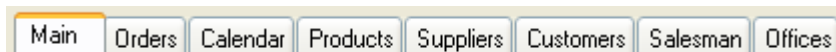
Optimization Tips and Tricks

- Please keep in mind that what really counts when it comes to optimization of a QlikView data model is the number of records.
- Don't normalize data too much. Plan for 6 – 10 total tables in a typical QlikView application. This is just a guideline, but there is a balance to be struck with QlikView data models. See the Data Model section of this document for more details.
- Eliminate small "leaf" tables by using Mapping Load to roll code values into other dimensions or fact tables.
- Store any possible field as a number instead of a string
- De-normalize tables with small numbers of field
- Use integers to join tables together
- Only allow 1 level of snow flaked dimensions from the fact record.(fact, dimension, snowflake, none)
- Use Autonumber when appropriate, will reduce application size
- Split timestamp into date and time fields when date and time is needed
- Remove time from date by floor() or by date(date#(..)) when time is not needed
- Reduce wide concatenated key fields via Autonumber, when all related tables are processed in one script (There is no advantage when transforming alphanumeric fields, when string and the resulting numeric field have the same length)
- Use numeric fields in logical functions (string comparisons are slower)
- Is the granularity of the source data needed for analysis? If not aggregate by using aggregating function like "sum() group by"
- Create numeric flags (e.g. with 1 or 0)
- Reduce the amount of open chart objects
- Calculate measures within the script (model size <> online performance)
- Limit the amount of expressions within chart/pivot objects, distribute them in multiple objects (use auto minimize)

Additional scripting best practice

Other scripting best practices include:

- Use Autonumber only after development debugging is done. It's easier to debug values with a number in it instead of only being able to use surrogates. See the QlikView Reference Manual if you are not sure how/when to use Autonumber.
- Put subject areas on different tabs so you don't confuse the developers with too much complexity



- Name the concatenate/join statements
- When adding script to a QVW, it is best to do a binary load on large data sets then extend the script. Later merge the script after development is near complete. This doesn't functionally change anything, but it saves time during development.
- Use *HidePrefix=%;* to allow the enterprise developer to hide key fields and other fields which are seldom used by the designer (this is only relevant when co-development is being done).
- When using the *Applymap()* function, fill in the default value with something standard like 'Unknown' & Value which is unknown so users know which value is unknown and can go fill it in on the source system without the administrators having to get involved. See the QlikView Reference Manual if you are not sure how/when to use Applymap().

```
StateMapping:
mapping load * inline [
St,State
Tx,TX
Te,TX
Tex,TX];

LOAD
ApplyMap( 'StateMapping' , St, 'Other')
```

- Never use Underscores or slashes (or anything 'techie') in the field names. Instead use code user friendly names, with spaces.
- Instead of: "mnth_end_tx_ct" use: "Month End Transaction Count"
- Only use Qualify * when absolutely necessary. Some developers use Qualify * at the beginning of the script, and only unqualify the keys. This causes a lot of trouble scripting with left join statements, etc. It's more work than it's worth in the long run. See the QlikView Reference Manual if you are not sure how/when to use Qualify and Unqualify.
- Use "Include" files or hidden script for all ODBC/OLEDB database connections.
- Use variables for path name instead of hard-coding them throughout your script. This reduces maintenance and also provides a simple way to find paths (assuming you put them in the first tab to make it easy to find).
- All file references should use Container naming convention.
- Always have the Log file option turned on if you need to capture load-time information for debugging purpose
- Comment script headings for each tab. See example below:

```
//=====
// App Name:    Wireframe
// Author:      Matt Stephens, QlikTech
// Created:     June, 2010
// Purpose:     This app is a template app demonstrating the use of
//              wireframe backgrounds to organize QlikView screens into
//              logical and effective presentation themes. There is also
//              a zip file called Wireframe Images.zip that accompanies
//              this QVW. It holds dozens of pre-built wireframe images
//              in various color schemes.
// Modified:    July 18, 2010 BPN - added Intro tab comments
//=====
```

- Comment script sections within a tab with short descriptions. See example below:

```
// -----
// Load the Sessions table first
// -----
Sessions:
LOAD
    MakeDate(LEFT(Timestamp,4), MID(
        Date(Timestamp, 'YYYYMMDD') & '_'
        Time(Timestamp)      as SessionsTi
        Timestamp            as Timestamp,
```

- Add change date comments where appropriate. See example below:

```
Looptable:
LOAD FileName as QVDName
//FROM $(MetaPath)FileList.qvd(qvd)
resident FileList //changed 2010-09-06
WHERE UPPER(Extension) = 'QVD';
```

- Use indentation to make script more readable by developers. See example below:

```
// -----
// Main loop though all QVDs found above
// -----
for X = 1 to fieldvaluecount('QVDName');
    let QVDName = fieldvalue('QVDName', $(X));

    Load *,
        upper('$ (QVDName)') & ' _' & Date(Today(), 'YYYY-MM-DD') as LoadDateKey,
        lower('$ (QVDName)') as FieldQVDFileName,
        Upper('$ (QVDName)' & '-' & date(Today(), 'YYYY-MM-DD')) as FieldHeaderKey;

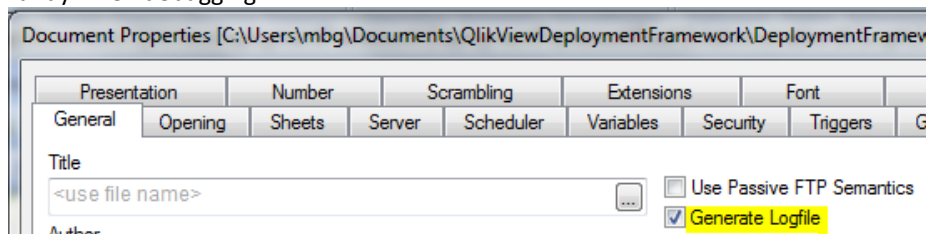
    QvdFieldHeader:
    LOAD
        //lower('$ (QVDName)') as QVDFileName,
        date(Today(), 'YYYY-MM-DD') as FieldHeaderDate,
        FieldName as QVDFieldName,
```

- Never use LOAD * in a load statement. Instead list the columns to load explicitly so that you know what fields will be loaded and this won't change as new columns are added or deleted from source tables. This also helps developers to identify the loaded fields in the script. See example below:

```
// =====
// Locations Data from a QVD
// =====
Locations:
LOAD LocationNbr      as [Location Nbr],
    AddressLine1      as [Address Line 1],
    AddressLine2      as [Address Line 2],
    City              as City,
    Country            as Country,
    CountryRegionCode as [Region Code],
    PostalCode         as [Postal Code],
    [State / Province] as [State Code]
FROM [$(QvdPath)Locations.qvd] (qvd);
```

Application logging

It is best practice to turn Document logging on under Document Properties and General Tab in the QlikView Application. These logs can be used to monitor the system by use of the Governance Dashboard. These logs are also very handy when debugging.



Deployment Framework log tracing and debugging

When the log is activated it's easy to find where in the DF initiation scripts the problem has accrued. Search for the log trace that starts with **### DF** alt **### DF Error** and after the section/include file name.

If error in the script is not generated in Deployment Framework section a good idea is to comment the initiation scripts and thereby using old Global Variables. The advantages of this is that the application log and debug sequence is shorter thereby easier to debug. Remember to activate DF initiation after the debugging.

If having problems with Section Access, Input Fields or other faults making application access impossible, use the initiation script (*1.Init.qvs*) as your escape. The command **Exit script;** in the beginning of *1.Init.qvs* will exit before the faulty script part executes.

Using binary load with Deployment Framework

To load from a QlikView mart the binary load statement need to be used in the QlikView scripts. Binary load can only be put as the **first** statement of a script. Best practice is thereby to use relative search path to the qvw mart in the binary section, instead of the framework global variables. Example:

```
Binary [..\..\4.mart\0.example_northwind_mart\example_northwind_mart.qvw];
```

The Deployment framework *1.Init* include sections will follow the Binary load section.

Front End Development

When creating a new user application it should always take the starting point from a Template Application. The document template should include the standard structure in the script and the companies visual guidelines implemented. The data source should primarily be QVD-files created in Back-End development phase.

UI Design

Design matters. It impacts user adoption rates, utilization rates, speed of analysis and usage patterns. All of these things impact how effective your QlikView document can be. The principles of good interface design promoted by Stephen Few and Edward Tufte are the basis for the best practices QlikTech recommends when designing and building a QlikView document. The outline below shows (at a high level) some of those tenants of good design. QlikTech makes many QlikView examples, documents, slide decks and other materials available to help demonstrate these principles.

QlikView Developer Toolkit

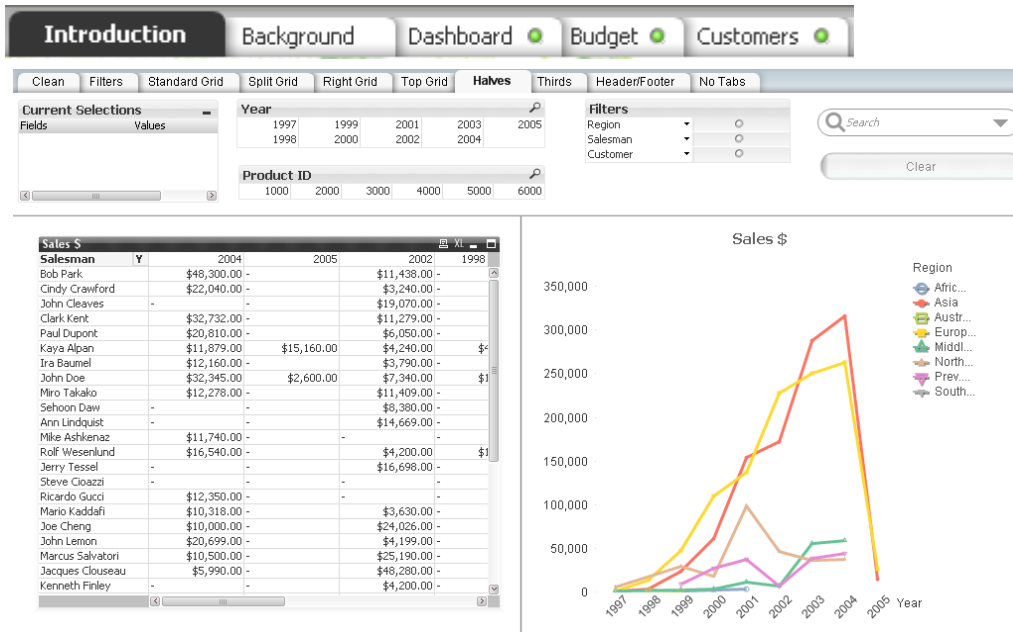
The number one tool for a QlikView Designer is the *QlikView Developer Toolkit* which is available with the installation of QlikView 11. The purpose of the Developer Toolkit is to help QlikView developers make more attractive & useable applications. There are a variety of backgrounds, guides, and panels that can be incorporated into your design to get you started

Developer Toolkit is divided up into several folders of assets

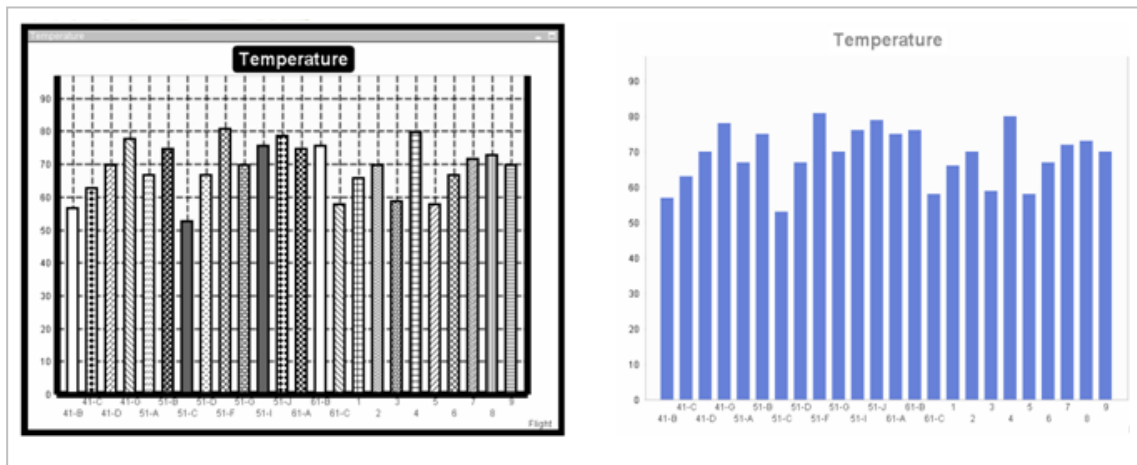
- Backgrounds: help define space to place objects on
- Buttons: images to use as buttons
- Guides & Rulers: help you align objects within QlikView
- Icons: useful images for common tasks
- Panels: can be used to define spaces when using a background you have found
- Qlik: QlikView branded images
- Rules: are simple line styles to divide up regions of space
- Shadows: are more graduated ways of dividing space

UI Best Practice

Use of supplied or developed templates and tabs for consistency and simplicity:



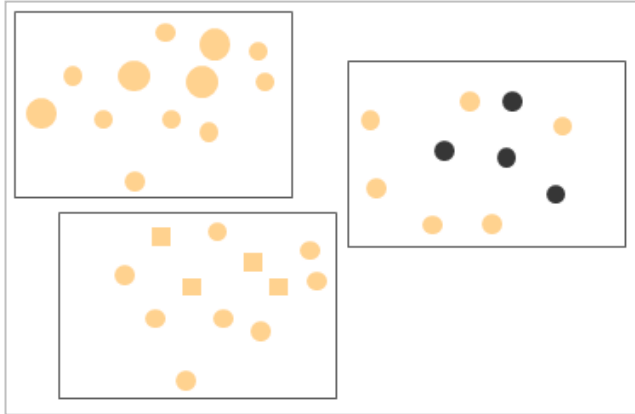
Use of implied closure to limit non-data ink space:



Use of neutral and muted colors and use of contrast: Muted and neutral colors are much less strenuous on the eyes and increase user adoption. Use of contrast helps the eyes quickly identify interest points or exceptions. These concepts go together, since the use of contrast with primary colors is difficult to do. Consider a combination of muted colors and the use of contrast in all charts, especially where exceptions or outliers are meant to be highlighted.



Use of size, shapes and intensity to call attention to data points: Shapes are another rapid identification point for the eyes. They can be used to segment data points into groups. Color intensities work well for ranges of values or outliers.



Additional UI Best Practice

- Put a current selections box on every sheet in the same location
- Make list boxes appear in the same locations on every sheet
- Organize list boxes and multi-boxes first in the frequency of use (most used on the top, least used on the bottom). Then, sub-sort the list boxes into groups in hierarchical order (largest group on the top, smallest group on the bottom).
- Put dropdown select properties on every straight/pivot table
- Use Variables as expressions instead of defining the expressions directly in the expression editor
- When Creating a Drill group, add an expression for the label of the field in the drill group. The expression should be equal to Only(All Higher fields) & '>' & 'current field name', so that it equates to Sales-RepA>Product.SalesRepA is the item which was drilled into, Product is the values which are represented in the chart
- Instead of defining exceptions in straight/pivot tables, instead use charts which show the exceptions quickly
- Always include a Help / How-To tab and/or a link to a help site on our website. Examples of Help/How-To tabs are included in the Getting Started section in QlikView. Consider copying one of the interactive How-To pages into a template that you can use across applications.
- Name each sheet and object with descriptive headers
- Black & White charts are best when considering color blindness and simplicity
- Red & Green - Many people are red/green color-blind - consider this e.g. when using visual cues
- Red and green are also associated with good and bad indicators / performance. Only use red and green when you mean to indicate good and bad.
- Design for a fixed resolution that applies to your organizations desktops (e.g. 1024 x 768)
- Always consider sort order and whether to present frequency (# or %) in list boxes (sometimes very useful but definitely not always)
- Repeated objects (clear buttons) at the same position in every sheet
- Multi boxes can be good for people that are used to working with QV but they are not very intuitive. List boxes take more space but are better (you can e.g. see the gray areas better).
- Clean layout in charts – line up axis titles, chart title, text, etc...
- Hierarchy dimensions placed in order
- Time and Dates are crucial elements of most apps and they must be highly intuitive to search and use
- Table columns should always be searchable (display totals in tables whenever it makes sense)

QlikView

QlikTech strongly recommends the incorporation of design best practices for all QlikView developers and designers when starting a QlikView deployment. Good interface design leads to high adoption rates and effective interfaces. QlikView's rich UI layer allows for world class visualization and design in all QlikView applications.

For new QlikView deployments and new designers it is strongly recommended that QlikView Designer training be attended by all developers and designers. The Designer courses are structured to reinforce good design and to learn the QlikView techniques that help deliver that design in a simple, elegant way. They are also a great opportunity to practice good design and apply that design to your QlikView applications in a lab setting.

Many of the design best practices are displayed in the demo applications that are publicly available at <http://www.demo.qlikview.com>. Also visit QlikCommunity for more tips.

UI Design References

- QlikView Developer Toolkit
- QlikView Demo <http://www.demo.qlikview.com>
- Information Dashboard Design, by Stephen Few
- Show Me the Numbers, by Stephen Few
- The Visual Display of Quantitative Information, Edward R. Tufte
- Visual Explanations, by Edward R. Tufte

Color Scheme Variables

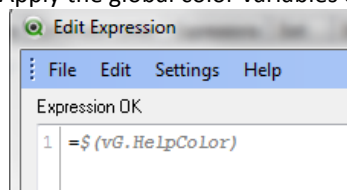
Use Global Variables to reuse company color schemes. It is easier more consistent to develop the GUI when using pre defined color variables. Use the Color Scheme global variable `vG.ColorSchemePath` and store schemas in include files. Example, include Color scheme script in Deployment Framework tab after init section:

```
$(Include=${vG.ColorScheme})\0.Example_ColorScheme.qvs);
```

In the scheme include file add global color variables by using the company RGB codes

```
SET vG.HelpColor = RGB(234,94,13);
```

Apply the global color variables on the objects



Variable expressions

Global Variables is a good way of reusing expressions, edit the expressions in Variable Editor.

Reasons for holding expressions in variables:

- To achieve reuse: the formula for a measure such as Sales usually remains the same across a QlikView document, so it doesn't make sense to write it on every chart.
- To enforce consistency in the formulas: by avoiding the risk of having different formulas that calculate the same measure.
- To provide a single point to apply changes: if and when a formula needs to be changed, you only need to change one variable and all the charts and other objects that refer to that variable will follow.

- To allow the end user to make changes through an input box, when needed. This could be the case of targets for KPIs or general parameters.

Expression Optimization Tips

- Eliminate Count(Distinct x)'s **They are very slow**
- Eliminate Count Numbers, or Count Texts, they are almost as slow as Count(Distinct)
- `date(max(SDATE,'DD.MM.YYYY'))` is factor xxx faster than `max(date(SDATE,'DD.MM.YYYY'))`
- Use numeric flags (e.g. with 1 or 0) which are pre-calculated in the script
- `sum(Flag * Amount)` and `sum(if(Flag, Amount))` **use instead** `sum({Flag=1} Flag * Amount)`
- Reduce the amount of open chart objects
- Limit the amount of expressions within chart/pivot objects, distribute them in multiple objects (use auto minimize)

Macros

The following are some reflections you should be aware of when you start including macro statements in your application. There are also a number of reasons why to avoid macros

Running a macro could result in deletion of the QlikView Server cache. undo-layout buffers and undo logical operation buffers and this in general has a very large negative impact on performance as experienced by the clients. The reason for deleting the caches etc. is that it is possible to modify properties, selections from the macros, thus opening up for conflicts between the cached state and the state that was modified from a macro and these conflicts will practically always crash or hang the clients (and in worst case; hang or crash the server as well).

The macros themselves are executed at VBS level while QlikView in general is executed at assembler level which is thousands of times faster by de-fault. Furthermore, the macros are single threaded synchronous as opposed to QlikView that is asynchronous and heavily threaded and this causes the macros to effectively interrupt all calculations in QlikView until finished and thereafter QlikView has to resume all interrupted calculations which is a delicate process and very much a source (at least historically) for deadlocks (i.e. QlikView freezes while the macro is still running, without any possibility that the macro will be finished).

While QlikView is increasingly optimized in terms of performance and stability, the macros will always maintain their poor performance and the gap between genuine QlikView functionality and the macros will continue to increase, making macros less and less desirable from a performance point of view. This fact combined with the above fact that the macros tend to under-mine all optimizations made in QlikView calls for severe negative tradeoffs as soon as macros become an integral part of any larger application.

The macros are of secondary nature when it comes to QlikView functionality - first all internal basic QlikView functions are run and tested and thereafter the macros are run and tested which effectively means that macros will never have the same status or priority as basic QlikView functionality - always consider macros as a last resort but nothing much else. Since the automation API reflects the basic QlikView in terms of object properties etc., the macro content may actually change between versions making this a very common area for migration issues. Once a macro is incorporated in an application, this application has to be revisited with each new version in order to make sure that the macros were not affected by any structural changes in QlikView and this makes macros extremely heavy in terms of maintenance.

Only a subset of macros will work in a server environment with thin clients (Java, Ajax) since local operations (copy to clipboard, export, print etc.) are not supported, though some of these have a server-side equivalent (e.g. Server-



SideExport etc.) that is very expensive in terms of performance with each client effectively affecting the server performance in a negative way.

In conclusion: what we are striving for is a heightened awareness when it comes to macros and what may work with a few thousand records does not necessarily scale very well when macros are involved and the problems tends to manifest themselves and become more serious when larger datasets are involved. It is also important to note that certain events can only be captured through the use of macros and for this reason it may be difficult to avoid macros altogether. The R&D department always strives to incorporate as much of this functionality as possible as basic QlikView functionality, thus limiting the use of macros in the long run – however as previously stated: certain events are difficult to catch except from an outside macro...

Given all of the above, macros cannot be part of any recommended QlikView design pattern!

Actions

Action has been around since QlikView 9. They are derived from the old button shortcuts, which they also replace. Apart from offering a much wider range of operations than the old shortcuts (including most common operations on sheets, sheet objects, fields and variables), you may also define a series of operations within a single action. The introduction of actions should greatly reduce the need for macros, which is good since macros are never efficient from a performance point-of-view.

Actions can not only be used on buttons. Also text objects, line/arrow objects and gauge charts can be given actions, which are executed when clicking on the sheet object in question.

The trigger macros of previous versions of QlikView have been replaced by trigger actions. This gives you the possibility to build quite elaborate triggers without the use of macros. Trigger macros from previous versions will be automatically translated to a Run Macro action when loaded into QlikView.

Read more about Triggers in the QlikView Reference Manual.

Tools

Variable Editor

Variable Editor is a QlikView application that graphically controls Deployment Framework. *System* and *Custom Global Variables* can be added and edit within Variable Editor and all containers are plotted in a Container Map (master is stored in Administration container) this map can also be edited with Variable Editor.

Variable Editor is found under *6.Script\2.VariableEditor\VariableEditor.qvw* in the *0.Administration* container.

Do not execute Variable Editor on a remote computer with high latency network access to the container folder structures, Variable Editor is not optimized to run on slow networks.

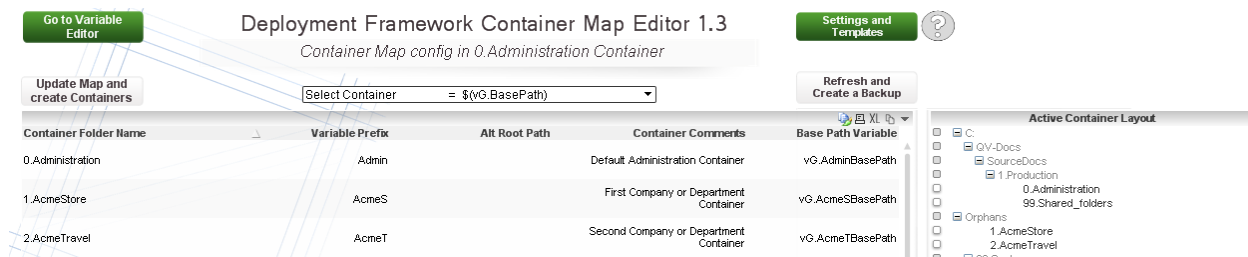
Help ?

There is a help button in the VariableEditor available when needed.

Container Map Editor

Is used to administrate and populate containers within the framework. Press **Go to Container Map** to change to container view. Container Map is used by Deployment framework to find the containers and to create container connection variables. These variables are created when using the *LoadContainerGlobalVariables* function, example: **call LoadContainerGlobalVariables ('HR') ;** Will connect to 1.AcmeHR container (if exist).

To edit container map the Variable Editor application **MUST** be started within the *0.Administration* container.



Edit or modify container map in the table, remember that it's only the container Map that is changing not the physical container structure. *Selected container* box must be set to *vG.BasePath* or *vG.AdminBasePath* to be in the *0.Administration* container and create new containers.

Container Input Fields

- *ContainerFolderName* contains the Container folder Name. To create or add in a sub container structure type *folder name\container name*. Example 1: *1.Oracle* to create a container in the same level as 0.Administration
Example 2: *98.System\1.Oracle* to create a container under a system folder. To add a container in another file system.
- *ContainerPathName* , enter prefix share variable names in *ContainerPathName* field, example *Oracle*.
- *Alt root path*, Edit an optional container path in *alt root path* field. A container could also be copied in a sub-folder structure the subfolder name will be created automatically.

Select Container

Use this dropdown to select witch container that we should view add or modify. Default (vG.BasePath container) is the same container as the Variable Editor application is stored. All the other containers are found based on the container map and if the container physically exists.

Refresh Create a Backup

Will refresh the view without changing container settings and also create a Container Map backup.

Retrieve Backup

Use Retrieve Container Map Backup to get back to the backup stage.

Update Container Map

Use this button to apply the new Container map after adding and/or modifying the container layout.

Update Map and
create Containers

Create New Containers option

Create New Containers will create containers based on the current container Map. This button is only shown after Update Container Map is applied and accepted. New Containers can only be created from the 0.Administration container this means that the selected and applied container either is *vG.BasePath* or *vG.AdminBasePath*.

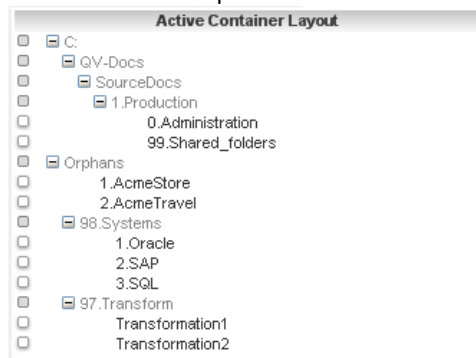
Create Containers based on the new Container Map?

Create New
Containers

Cancel

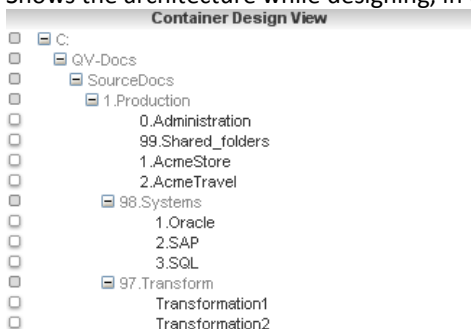
Active Container Layout

Shows physical containers that exist within the Container Map Container that exists in the Map and not in real life will be shown as Orphans as shown in the example below:



Container Design View

Shows the architecture while designing, in this view no Orphans is shown and no reload/refresh is needed.



Variable Editor Tab

Go to Container Map Editor

Settings and Templates

Deployment Framework Global Variable Editor 1.3

HR_KPIVariables file in 0.Administration Container

Update Variables

Variable Files = HR_KPIVariables

Select Container = \$(vG.BasePath)

Refresh and Create a Backup

Add Variable File =

Variable Name	Variable Value	Variable Comments	Search Tag

Retrieve Backup

Search

Help

Variable Name

Type SET or LET in front of your variable.
Use vG. as global variable prefix.
Example SET vG.statistics

VariableValue

Do not combine numbers and letters
when using LET function use the
SET function instead

Variable Files

Variable files other than CustomVariables
will not be loaded by into the applications by default
Add Sub Function below into the application script:
`call LoadVariableCSV('${vG.BasePath}\HR_KPIVariables.csv',
[Specific variable Optional])`

Global Variables

The Global variables are modified by the Variable Editor and I stored in `$(BaseVariablePath)\CustomVariables.csv` files in each container. Global variables (with the prefix `vG.`) are loaded by default into QlikView during the framework initiation process in the beginning of the script (read more in using Deployment Framework Containers). Global variables should only be used when a variable is shared by several applications in a Container.

Universal Variables

By using Universal Variables that are stored in `$(SharedBaseVariablePath)\CustomVariables.csv` files in the Shared Folders Container, we get “single point of truth” across all containers. Universal Variables are by default loaded during the framework initiation process, have the prefix `vU` and is also modified by the Variable Editor application.

System Variables

System Variables are actually also Global Variables that start with (vG.), the difference is that System Variables are predefined variables used to store system settings like QlikView Server log path. System Variables are also not pre-loaded, *3.SystemVariables.qvs* include script needs to be run to load in the System Variables into QlikView. System Variables are modified by the Variable Editor and I stored in *\$(BaseVariablePath)\SystemVariables.csv*. There is usually only need for one System Variable version, the main is stored in 0.Administration container and is by default replicated out to the other containers.

Variable Input Fields

- *VariableName* Type *SET* or *LET* in front of your variable name. Use *vG.* or *vU.* as Global or Container Global Variable prefix. Example1 *SET vG.statistics*. Example2 *SET vU.statistics*.
- *VariableValue* Type value or text, when entering text do not use brackets (") this is done automatically. Do not combine numbers and letters when using *LET* function, use the *SET* function instead for this.
- *Comments* Used for comments like author and creation date
- *Priority* Used only for easy search

Variable Files, Custom Global Variables

Custom Global Variables will automatically be loaded into QlikView applications when using Deployment Framework. Each Container has its own Custom Global Variable file that the applications use.

For Global Variables that need to be used across containers modify Shared Custom Variable file with Variable editor.

Refresh Create a Backup

Will refresh the view without updating Variable files and at the same time create a backup.

Retrieve Backup

Use Retrieve Backup to get back to the backup stage created by *Change Variable File and Create a Backup* button.

Update Variables

Use this button to apply the new variables after adding and/or modifying.

Add and Remove Variable Files

Variable Editor has the possibility to add variable files into the selected container in addition to the default *Custom Global Variables*. Type the variable filename into the *Add Variable File* input box and press enter like example below:

Add Variable File = HR_KPI

When running the QlikView script (*Refresh and Create a Backup* will execute the script) the new empty file will be created as *HR_KPIVariables.csv* and stored under selected container *3.Include\1.BaseVariable*.

To remove a Variable File add the command **del** before the filename and run the script (*Refresh and Create a Backup* will execute the script) like example below:

Add Variable File = del HR_KPI

Variable files other than Custom Variables will not be loaded by *1.Init.qvs* into the applications by default.

Add Sub Function below into the application script instead:

```
$(Include=$(vG.SubPath)\2.LoadVariableCSV.qvs);
```

```
call LoadVariableCSV('$(vG.BaseVariablePath)\HR_KPIVariables.csv ' , '[Specific variable Optional]')
```

Variable Files, System Variables

System Variables setting are hardware and system folder settings, example log locations needed to monitor the platform. QlikView System monitor uses these settings.

To execute System Variables inside a QlikView application include:

```
$(Include=$(vG.BaseVariablePath)\3.SystemVariables.qvs);
```

These are the default System Variables, change so that these settings represent your QlikView environment.

- *vG.ServerLogPath* QlikView Server logs path.
Default to C:\ProgramData\QlikTech\QlikViewServer\
- *vG.UserDocumentPath1* is QlikView Sever User Document path. If having more User Document folders use *vG.UserDocumentPath2*, *vG.UserDocumentPath3*...
Default to C:\ProgramData\QlikTech\QlikViewServer\QlikView\
- *vG.QMSPath* QlikView Management Service ProgramData folder path.
Default to C:\ProgramData\QlikTech\ManagementService\
- *vG.QVPRPath* Publisher QVPR data base path (usually the same as *\$(vG.QMSPath)\QVPR*).
Default to *\$(vG.QMSPath)QVPR*
- *vG.QDSPath* QlikView Publisher Distribution Service ProgramData folder path.
Default to C:\ProgramData\QlikTech\DistributionService\
- *vG.DSCPath* Directory Service Connector ProgramData path
Default to C:\ProgramData\QlikTech\DirectoryServiceConnector\
- *vG.QVWSPath* Path to QlikView Web Service ProgramData
Default to C:\ProgramData\QlikTech\WebServer\
- *vG.SAPPath* SAP Connector ProgramData path.
Default to C:\ProgramData\QlikTech\Custom Data\QvSAPConnector\

- *vG.SFPPath* Sales Force Connector ProgramData path.
Default to C:\ProgramData\QlikTech\Custom Data\SalesForce\

Optimization tools

According to the concept “QlikView on QlikView” a number of optimization tools are available

QlikView Optimizer application

Analyze the QlikView mem-file and detects “expensive” fields and objects in an application

Unused Fields application

Detects fields in the data model that are not in use in the application

Complexity Analyzer (included in Governance Dashboard)

Set a complexity index on each application depending on parameters like:

- usage
- number of records
- cardinality (distinct values)
- number of user objects
- calculated dimensions
- long expressions
- etc



Troubleshooting & Support

Support Types

Supporting QlikView applications and environments can be done in several ways.

As a best practice, QlikTech recommends that support levels and services be identified for the following areas:

- QlikView Applications (QVWs)
- QlikView Interface (end user support)
- QlikView Server/Publisher
- QlikView Data Architecture (QVDs and QlikView data, in general)

Many QlikView clients utilize certified QVWs for application support of high importance apps. This can help especially when business teams are creating their own QVWs and your support team is only responsible for supporting the certified applications that it had a chance to code/interface/data review. See the section called Testing & Certification in this document for more details on the certification process.

Appendix A, Checklists

Development Checklists

QlikTech recommends the use of a developer checklist to highlight and reinforce development best practices. Most enterprise clients develop this from a template or sample of best practices. Consult your Account Executive or Regional Services Director for a sample from QlikTech. One way to help promote the visibility and presence of the checklist is to limit it to one page and laminate it for each developer. This will make it easier to post the checklist and refer to it often. Some clients will use the checklist in code reviews to ensure that best practices were followed before releasing a QVW to Test or Production environments.

Data Model Performance

- ☐ Synthetic keys removed from data model
- ☐ Ambiguous loops removed from data model
- ☐ Correct granularity of data
- ☐ Use of QVDs where possible
- ☐ Use integers to join tables where possible
- ☐ Remove system keys/timestamps from data model
- ☐ Unused fields removed from data model
- ☐ Remove link tables from very large data models
- ☐ Remove unneeded snowflake tables (consolidate)
- ☐ Break concatenated dim. fields into distinct fields
- ☐ All QVD reads optimized
- ☐ Use Autonumber to replace large concatenated keys

Interface Performance

- ☐ Run QlikView Optimizer to test memory usage
- ☐ Minimize count distinct functions
- ☐ Minimize nested ifs
- ☐ Minimize string comparisons
- ☐ Macros minimized or eliminated
- ☐ Minimize Show Frequency feature
- ☐ Minimize open objects on sheet
- ☐ Minimize set analysis against large fact tables
- ☐ Minimize pivot charts in very large apps
- ☐ Avoid "Show Frequency" feature on large data
- ☐ Avoid AGGR function when possible
- ☐ Avoid IF statements in calculated chart dimensions
- ☐ Avoid built-in time functions in GUI (inmonth, etc...)

Development checklist example

Design Best Practices

- ☐ Use of colors for contrast/focus only
- ☐ Use of neutral and muted colors
- ☐ Use of templates/themes where available
- ☐ Display optimized for user screen resolutions
- ☐ Design consistency across tabs
- ☐ Formatting consistency across objects
- ☐ Most used selections at top - least at bottom
- ☐ Drop-down selections on all straight/pivot table columns
- ☐ Developer QV version matches production
- ☐ Test client types for rendering
- ☐ Use of Common Variables for expressions
- ☐ Use calculation conditions on large charts

Script Best Practices

- ☐ Naming standards used for columns, tables, variables
- ☐ Script is well commented - changes date flagged
- ☐ First tab holds information section
- ☐ Subject areas each have tab in script
- ☐ Use of Include files or hidden script for all ODBC connections
- ☐ All code blocks with comment sections
- ☐ All file references using Global Variables naming
- ☐ Business names for UI fields
- ☐ Connection strings in Include file
- ☐ Turn Generate Log file option on
- ☐ UPPER() function used on Section Access fields
- ☐ Publisher Service Acct added to Section Access
- ☐ Use numeric flags where possible

Optimization Checklist

Hardware and Windows		
Bios Settings	Status	Comments
Latest Bios version?		The initial Bios version is often buggy
NUMA		Disable (Node Interleaving=Enable)
Energy Saving/Power Profile		Maximum Performance
Hyper threading		Disable
Hardware pre-fetch		Disable
Turbo Boost		Optimized for Performance
Memory	Status	Comments
Is memory alignment correct?		Align memory according to specs
Same memory size in all slots?		Do not mix memory size
Max memory speed		Use only fast memory, Do not mix memory speed
Hemisphere mode achieved? (half full or full memory slots)		Hemisphere mode will increase memory speed
Windows	Status	Comments
Power Options/Power Plan to High Performance		Even when energy saving mode is disable in bios this still need to be set
Automatic page file disable, set fixed to 20GB		Could drain resources
Backup running during office hours		Backup system could file lock resources that QlikView needs
Antivirus Services		Antivirus could file lock resources that QlikView needs
QlikView Infrastructure		
AccessPoint IIS and client	Status	Comments
Is QVWS disabled?		QVWS will disable IIS AccessPoint
QlikView AJAX app pool, Rapid-Fail Protection: False		Rapid-Fail Protection could disable application pool
Is QVP protocol via Tunnel?		Will reduce performance on the Plug-In client.
QlikView Server settings	Status	Comments
Working Set High and Low set properly?		Working set should be high enough to utilize all free memory (reserve 5 - 12 GB for Windows)
Is Document Timeout set properly?		If several documents are used this setting should usually be set between 240 -30 minutes (not default 480min)
QlikView Publisher (QDS)	Status	Comments
Advanced! Extend heap size in registry (Tab hive size)		To run more parallel tasks than 10

QlikView Developer		
Data model	Status	Comments
Is the model simple?		Keep the data model as simple as possible. Star scheme is preferred.
Are there apps with more than one fact table?		Hops between tables in calculations will reduce speed
Are there link tables?		Speed will be reduced when using link tables
Is the data model optimized?		Less decimals gives less distinct values, is it possible to make field numeric only
No of Hops between tables		The more hops in a calculation the slower it gets
Interface	Status	Comments
Calculation or Show conditions used?		Show condition use less CPU
Are unused objects minimized?		Minimize objects do not use any resources
Using circular groups?		All objects in group are calculated
Variables for repetitive calculations		A variable will calculate only once
Count(distinct)		Separate flag + sum much faster
Try to avoid calculated dimensions		Most often calculated single threaded
Have you identified single threaded operations?		
Are there lots of complicated set analysis expressions?		
QlikView Optimizer application	Status	Comments
Biggest (Bytes) fields (ID)		Could these be optimized?
Biggest (Size) fields (ID)		Could these be optimized?
Big fields with many unique values (count)		Could these be optimized? Like removing decimals?
Biggest key fields		Could auto number be used?
Number fields typed as text		Long number fields like security no could be typed as text, double byte size for every record.
Unused Fields application	Status	Comments
Could unused fields be removed?		
Complexity Analyzer (Governance Dashboard)	Status	Comments