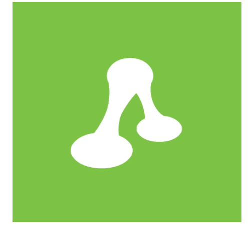

KnowledgeKube

BEST PRACTICE GUIDE

mercato | USER EDUCATION

KnowledgeKube Best Practice Guide

KnowledgeKube is a product of Mercato Solutions. www.knowledgekube.co.uk

The information in this document is provided for information only and is subject to change without notice. While we make strenuous efforts to ensure that the content of this document is accurate and up to date when published, Mercato Solutions Limited gives no warranty on its accuracy and assumes no responsibility or liability for any errors or inaccuracies whatsoever (to the extent such exclusion is permitted by applicable law).

KnowledgeKube® is a registered trademark of Mercato Solutions®. © Copyright Mercato Solutions Limited. All Rights Reserved.

First Published: May 2016

Published by Mercato Solutions Limited.
45-55 Camden Street
Birmingham, B1 3BP, UK.

www.mercatosolutions.co.uk

A version of this documentation can be found online at **http://documentation.knowledgekube.co.uk/best-practice-guide**

[THIS PAGE INTENTIONALLY LEFT BLANK]

Table of Contents

Introduction i
Common Terms i
Starting A New Project 1
Planning Your Application 1
Before Building Your Application 1
Establish an Admin Screen 1
Build a Question Set in a Spreadsheet 2
Build a Status and Role Matrix 2
Establish a Location For Shared Documents 2
Design a Process Flow Diagram 2
Build a Wire-frame Mock-up of Your Finished Site 2
Give Project Managers and Team Leaders Access to Private Models 2
Project Implementation Cycle 2
Choosing a Suitable Data Source 4
Working with Online and Offline Data Sources 6
Naming Conventions 7
Group Dependencies 9
Client Browser Best Practices 9
Building an Application 11
Saving Your Model 11
Refreshing Your Model 12
Source Control 12
Using Expression Comments 13
Using the RedirectToUrl Function 13
Using Question Groups 14
Writing Question Text 15
Add to Expression Parser and Auto Post Back 15
Using Identifiers 15
Using Identifiers in Expressions 15
Managing Variable Memory 16
Using Range and Matrix Definitions 19

Using Actions 20
Using Attributes 20
Planning the Application's Appearance 20
Using Style Classes 21
Using Read-only Text Decoratively 22
Using Document Images 25
Using Expressions and Functions 27
The Expression Engine 27
Writing Expressions 28
Multi-line Expressions 29
Optional Parameters 30
Using If-statements 31
Compound and Labelled Statements 31
Including Semicolons in Expressions 32
When to Use 'If' and 'Or' 33
When and How to Use Ampersands 34
Repeated Expressions 34
Error Handling 37
How KnowledgeKube Handles Errors 37
Exceptions and The Stack 37
Model-Level Error Handling 38
Error Expression Code Block 39
Error Steps 40
Using External Data Sources 43
Connect Conditions 44
Using T-SQL Data Sources 44
Creating an SQL Database 46
Date/Time Format Conventions 47
Data Sources and Variables 47
Using Filters 48
Primary and Secondary Level Filtering 49
Refreshing Filters 49
CSV Data Sources 50
Using WithDBTrans 51

Introduction

The KnowledgeKube Best Practice Guide will help you avoid common errors and use time-saving techniques, whether you're starting a new project or working on an existing one. It is designed to be read alongside the **KnowledgeKube** guide and **Expression Cookbook** in order to give you a full understanding of the program and how it can be used. Because of this, the Best Practice Guide will make frequent references to terms and features described in those documents without necessarily explaining them.

Common Terms

Here are some common terms used within KnowledgeKube and in this guide:

Term	Context	Meaning
Application	KnowledgeKube	A KnowledgeKube project as it appears on the site, consisting of one or multiple models.
Execute	Expressions	When an expression is triggered, either by a button or other command, it is said to 'execute'. Some expressions can execute themselves automatically, others only by specific instruction, all depending on type and conditions.
Form Trace	Expressions, Variables	The form trace records stack information in real-time and is returned by expressions or displayed in the Form Trace window of the Preview tool.
Front-end	KnowledgeKube	The site/GUI that displays your model.
Model	KnowledgeKube	The largest element in the KnowledgeKube interface that holds all other content, such as question groups, actions and data sources.
Pass, Assign	Expressions, Data drawn from data sources	The transfer of information from one point to another, such as from a question to a variable. When an expression is successfully executed it will generate a value that must then be stored somewhere. This could be its own keyword (if added to the expression parser) or a variable. The value is automatically transferred upon being generated, and the process is referred to as "passing" or "assigning" the value to the variable.
Return	Expressions, Data sources	To generate a brand-new value, such as the outcome of a calculation, or to locate an existing one, such as a value stored in a variable or data source row.

Term	Context	Meaning
Run	Actions	When activated using an expression, an action will "run", thereby executing all expressions contained within it.
Stack	Expressions	The part of KnowledgeKube that stores all data relevant to all calculations, executions and errors. Information can be returned from the stack using certain expressions.

1

Starting A New Project

When starting a new project, you should clearly establish the desired outcomes for the application. From there you can work out the required elements before you start building it and distribute work accordingly. This will give you a good idea of how the application will develop and reduce future modification and corrections. This is particularly useful for group projects as it helps everyone focus on their own work and allows for easy sharing of ideas before anything substantial is done to a application.

Planning Your Application

This topic contains some key points to consider before and during the creation of your application. In addition to completing these steps, here are some additional aspects to consider when planning your project:

- **"Project Implementation Cycle"** on the next page.
- **"Choosing a Suitable Data Source"** on page 4.
- **"Working with Online and Offline Data Sources"** on page 6.
- **"Naming Conventions"** on page 7.
- **"Group Dependencies"** on page 9.
- **"Client Browser Best Practices"** on page 9.

Before Building Your Application

The following steps should be considered before you start building your application:

Establish an Admin Screen

This will require a separate application to allow you to view and edit the tables in your data sources without needing to go into the database itself. Providing a **GUI** (Graphical User Interface) for modifying your database helps to ensure data can be maintained easily and with less chance of errors. Keeping this process separate will mean that you don't have to worry about users of the application gaining access to your database. You may wish to leave this step until you have started building your application but ideally should be in place by the time the application is completed. This can also be achieved using the *ShowModel* function to allow you to access the admin model from within the main model.

If you do implement an Admin model, make sure you limit which user roles can access it, in order to prevent confusion and unintended changes.

Build a Question Set in a Spreadsheet

This helps create a clear understanding of what the application is required to do. If you are building a basic insurance application, for example, then you need to know which order the question groups will appear in, what they each need to contain, where navigation buttons are placed and where they will direct the user, where expressions will be executed and so on. These requirements can then be laid out in a spreadsheet as they would appear in KnowledgeKube to give you a good idea of what the application will require, how work can be allocated and so on. This can be used as part of an agreed planning process, letting interested parties see an approximation of the finished application, and agree aims from the outset.

Build a Status and Role Matrix

Use a spreadsheet to plan which elements of the application will be affected by the roles assigned to the logged-in user, or by the current status of the application. This includes marking which areas of the application will be restricted to certain users, and which areas will be hidden under specific circumstances. Keeping track of conditional behaviour like this can be very useful when your application becomes larger and more complex, as it will let you see how the application should behave without having to examine it in detail.

Establish a Location For Shared Documents

When a user uploads a document to KnowledgeKube, the original location of the document is recorded. If a different user attempts to view or download one of these documents, KnowledgeKube will attempt to access it from the original location. If the original location was the user's local machine, nobody else will be able to access the documents. As such, you should ensure that files are uploaded from a network location that is accessible to everyone who needs to access the documents while making changes to the application.

Design a Process Flow Diagram

With the initial planning complete, you should be able to design a complete flow diagram of your application in a program such as Visio. This will let people see a simple, visual representation of the various processes that constitute your application, helping them to check whether each process behaves as it should. It will also make testing the application much easier, as you already know how it is supposed to perform. Use annotations to clearly explain any complex sections to help you.

Build a Wire-frame Mock-up of Your Finished Site

This is an optional step and may not be necessary if you feel confident enough to proceed. By this point you will know what is included at every stage, and should be able to create detailed diagrams of what each page of your site will contain. You may also want to incorporate role-based behaviour, to show how the application changes depending on user. This will also serve as final confirmation of your plans to proceed, as you will be easily able to correct any issues and make any necessary additions. Gather as much feedback as you can and make any required changes as necessary.

Give Project Managers and Team Leaders Access to Private Models

Private models cannot be searched for and do not appear in the models list. If there are any issues with your model in your absence, nobody else will be able to resolve them. Ensure you add key individuals to your model so that they can resolve errors on your behalf if necessary.

Project Implementation Cycle

Generally speaking, a model in a project will have one of four states:

- **Initial** - The model is being built and has not been tested or signed off.
- **Test** - A copy of a completed model used to implement and test future updates.

- **Release** - The model has been signed off by all parties and made available to the client.
- **Historic** - An expired copy kept for reference and testing purposes.

An individual model in a project has the following life cycle:

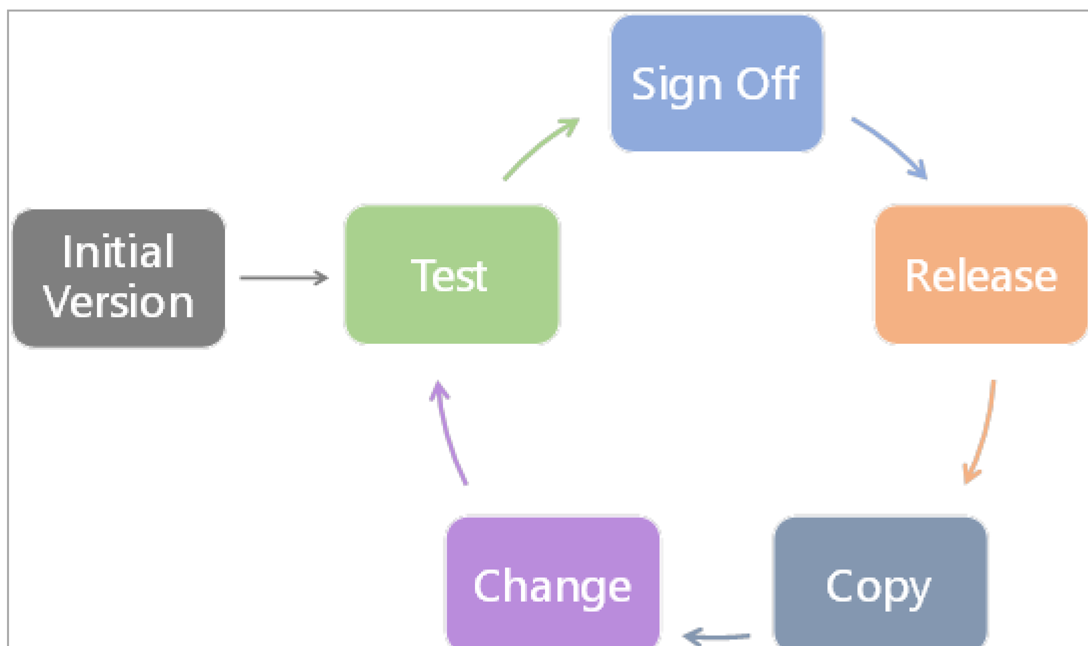

Figure 1-1: Project implementation cycle for a model.

- **Initial Version** - The first version of the model, from when it is first created to the point at which is it finished and ready for testing.
- **Test** - The model is thoroughly tested for errors and any corrections are made.
- **Sign Off** - The finished model is shown to the customer for approval. When the customer is satisfied the Operations team sign off the model and it is **Released**.
- **Release** - The model is published to the website and made publicly available. Changes cannot be made to this model - very minor updates can be made where required, but more substantial content must be added using a **Copy**.
- **Copy** - Each released model is copied, so that a backup is available and new content can be safely implemented and tested.
- **Change** - Any required updates are applied to the copy model, which then goes through the standard **Test** - **Sign Off** - **Release** process, eventually replacing the current released model.

When working on a model, team members must follow the below process to ensure that the development cycle is correctly adhered to:

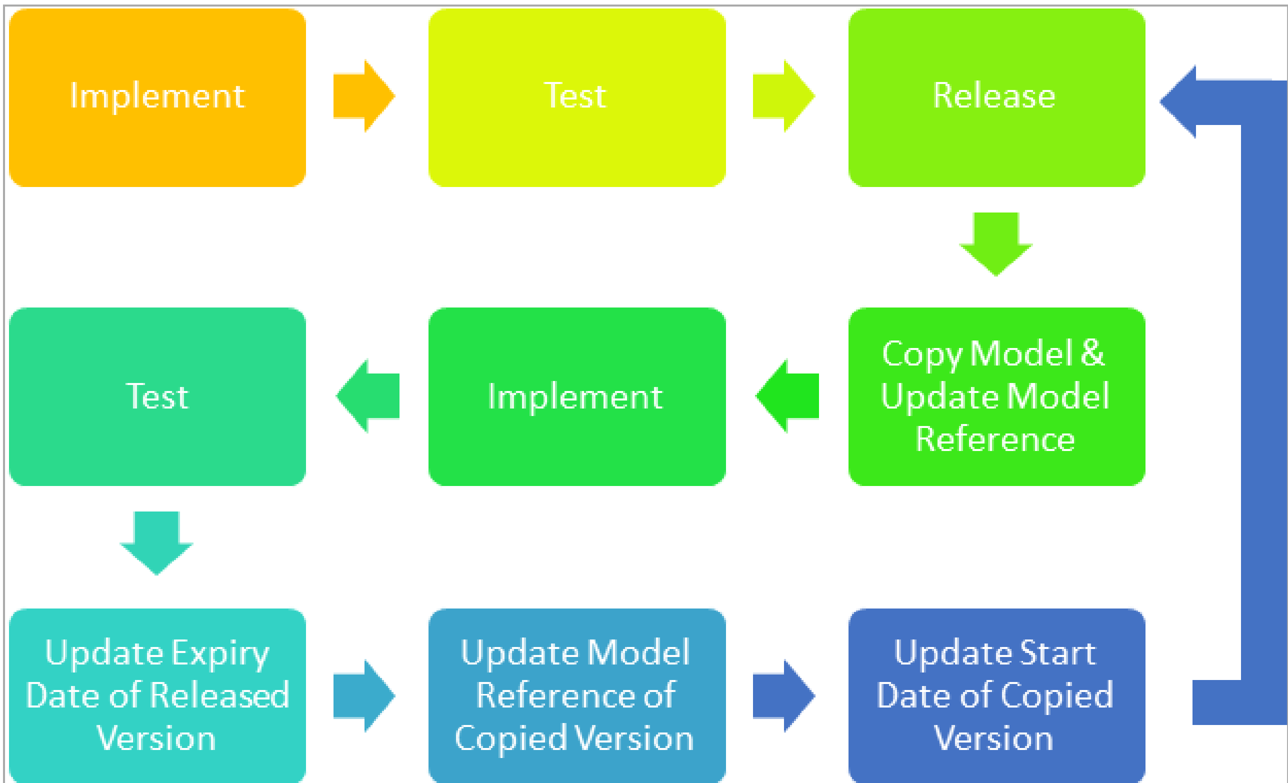

Figure 1-2: Implementation cycle for an Operations team member.

1. **Implement** - Working on a non-released model, such as an intial or copy model.
2. **Test** - Test the model when all work is complete - either in the **Initial** model or the **Copy** of the currently released model.
3. **Release** - When the model has been reviewed and signed off by the customer and the Operations team it is then made available. This is also known as making the model 'Live'.
4. **Copy Model and Update Reference** - When a model is released, it is copied. The copy is given a suffix such as 'Test' or a version number to differentiate it. Copied models can be hosted on their own dedicated CDS page, and made available to a customer for review if required.
5. **Update Expiry Date of Released Version** - Set a suitable expiry date for the released model if it is due to be replaced by a newer version. The model must be replaced by a suitable copy with an equivalent start date.
6. **Update Model Reference of Copied Version** - The copy model must have its reference changed to be the same as the released model, to effectively replace it when it expires.
7. **Update Start Date of Copied Version** - The copy model's start date is changed to the day after the release version is set to expire, so it replaces it immediately. The now-released model is copied and all previous steps are repeated.

Choosing a Suitable Data Source

KnowledgeKube is capable of managing complex data operations and storing the resulting data locally. Alternatively, it can write data to and receive data from an external data source, allowing a model to handle much larger quantities of data and become much greater in scope. The requirements of your project will largely determine whether or not you need to use a data source and which method of data management suits you best.

When working with a client you may be required to use their data structure, in which case the project's data requirements are largely out of your control. You may still be able to utilise other KnowledgeKube data tools, such as the **Data Designer**, to manage non-specific data, for example. In these circumstances there are important considerations in managing the client's data. Refer to **"Working with Online and Offline Data Sources"** on the next page for more information.

Please bear in mind that an external data source could be one that you operate and maintain within your own company. An "external" data source is simply any data infrastructure that operates outside of KnowledgeKube.

You should always consider that as your requirements grow, can your current data source adapt accordingly? As more users interact with the data and as more data is being read and written, will your infrastructure be able to keep up? Whilst an application may start small it could grow to become very substantial. Appropriate planning will help to avoid costly alterations in the future.

The key consideration is what data you will use, and how, if at all, it needs to be stored or made accessible to others. If data is only required within the scope of a single user session, it is more efficient to store it locally using string functions or matrices and then discard it when the session ends. If data needs to be recorded or accessed by multiple users at once then saving the data to an external data source will be more useful. Data maintained on a database can be set to automatically update itself, whereas short-term data will need constant human maintenance.

Some further points to consider:

- Databases have implementation and hardware requirements, as well as associated installation and upkeep costs.
- Security considerations for storing and moving data externally.

Each type of data source has its own inherent complexities to set up and maintain; you should use the one that is best-suited to you and the goals of your application.

The following list gives a general idea of the complexity of various KnowledgeKube concepts, ordered from least complex to most complex:

- Keywords, functions, tools such as matrices, and the *SaveForm* and *RedirectToUrl* functions. (Local).
- CSV data sources. (Local). This type of data source has natural limits to the amount of data it can process and ideally should not exceed 250 rows. Beyond this, your site may experience significant performance issues. If you need to be able to manipulate more data, consider using the **Data Designer**.
- Data Designer. (Repository). KnowledgeKube's in-built table creation and data entry system enables you to quickly and easily construct SQL-structured tables for use in any model on the repository. The tool is intuitive and requires no prior understanding of SQL to use. Data Designer is not as flexible as SQL Server and is comparatively limited in the quantity of data it can manage. If you expect that your data requirements may be excessive, or you require SQL Server-specific functionality, consider using a dedicated SQL database.
- SQL databases. (External). This type of data source is best for transactional data operations and can manipulate much greater volumes of data than any other method.
- RESTful APIs such as Google or Facebook (External). KnowledgeKube can make service requests to an API to interact with and return JSON and XML-format data.
- T-SQL databases. (External).

Working with Online and Offline Data Sources

You will typically have very little control and visibility of a client's data structure, so a client is usually asked to provide two versions of their data - an 'offline' and 'online' version. Online data is used by the client in their business, and offline data is typically a duplicate or backup that is safe to use for testing. Models in the implementation and testing stages, or that have expired, must use offline data, and released models must use online data.

Offline data is also known as 'Test' or 'Sandbox' data. Online data is also known as 'Active' or 'Live' data.

Because models are repeatedly copied in the implementation cycle, you cannot simply instruct a model to connect to one particular data type because you will only have to change it again when the model is released or expires. You will need to implement a method of selectively switching between offline and online data connections automatically according to the state of the model. That way when a model is released it will automatically use online data, and offline data when it expires. This is done using a **Constant** keyword and conditional expressions.

Neglecting to change the connection type can lead to significant problems, especially with respect to security and data integrity, and potentially compromise a whole project.

When a model is created it should have two connections created as standard - connections to the online and offline data, respectively. A constant keyword is created that will store either the value **1** - for non-released models - or **2** - for released models. The constant is then referenced in conditional expressions anywhere the model needs to change its behaviour, such as establishing data source connections. You only have to change the value of the constant to change the behaviour of the model, allowing you to quickly and easily transition models as they are released and expire.

For example, the following expression is used to choose a connection type depending on the value of the constant *CONModelStatusID*:

```
If:(CONModelStatusID = 1)
{
	SetConnectionProperty("OfflineDataConnectionName", "Credentials");
}
Else
{
	SetConnectionProperty("OnlineDataConnectionName", "Credentials");
};
```

These principles also apply when using the **Data Designer**. You should not maintain completely separate databases for online and offline data, as maintaining them independently is laborious and an inefficient use of time. Instead, you should create a table containing status values - using **1** for offline data and **2** for online data - and use the status value column as a foreign key to define which data is to be used for online or offline purposes. This enables you to manage all of your data in one place and filter it according to the status value, ensuring the model always uses the appropriate data.

For example, the model status table would be structured as follows:

ModelStatus	Status
1	Offline for Testing
2	Online for Release

The *ModelStatus* column is then used as a foreign key in a data table:

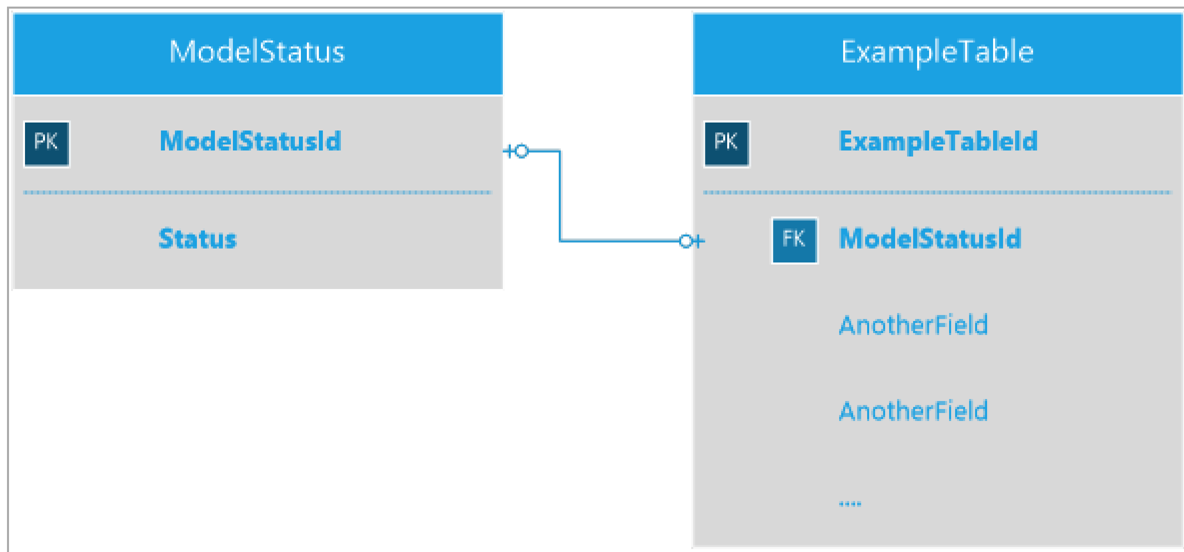

Figure 1-3: ModelStatus column used as foreign key.

Naming Conventions

As part of planning your application you should decide upon a consistent naming convention. Implementing a clear and consistent language for questions, expressions, variables etc. from the start will avoid mistakes caused by ambiguity, especially as a application grows larger and its functionality becomes more complex. This is especially useful in group projects, as it allows each member of the project team to be able to discern the purpose of content based on its name.

It is recommended that you give each element a descriptive name that clearly defines its purpose, and include an abbreviation specific to the type. This also lets you assign similar names to related elements - such as an expression and the variable that receives its value from that expression - to help make the relationship between those elements and make their purpose clear.

An example of a naming convention is shown in the following table:

Item	Context	Example
Action	Actions, also referencing its purpose.	*ACTApproveClaim*

Item	Context	Example
Alias	Data source field aliases.	*InsuredAddressAlias*
Axis	Matrix axes, referencing the question or variable that provides the value.	*TotalCostAxis*
Button	Buttons, to differentiate between navigation buttons and buttons used to execute other commands.	*BTNAddClaim*
Column	Data grid column keywords.	*COLClaimDate*
Data Source	Data sources, this is mainly to help identify it when referenced in an expression. The name could also refer to the name of the table in the data source for further clarity.	*ClaimsHistoryDataSource*
Element	Axis elements, referencing the name of their parent axis. Be sure to add other identifying details to differentiate individual elements if you have a lot per axis.	*TotalCostElement*
Filter	Data source filters, including a basic description of what the filter does.	*ApprovedClaimsOnlyFilter*
Get	Expressions (or actions containing expressions) that return a value	*GetTotalCost* *GetTotalCostAction*
Grid	Data grid questions.	*GRDClaimsHistory*
Matrix Range	Matrix and Range rating definitions respectively.	*MTXPolicyRating* *RNGPriceDiscount*
Place Holder	Place holder questions, referencing the name of the child question group.	*PHAdditionalClaim*
Show	Read-Only Text/Narrative questions that display the value of another question or variable.	*ShowTotalCost*
Var	Variables, referencing a related function if applicable.	*VARTotalCost*

For example, if you had an expression to calculate today's date and then pass the result to a variable, you could call the expression *GetTodaysDate* and the variable *VARTodaysDate*. The names given to these elements makes them easy to distinguish; even if someone doesn't have a thorough understanding of the application, they should still be able to work out how the different parts work together. Furthermore, a clear naming convention will assist in searches using the **Query Model** feature of the **Model Inspector**, which in turn will save time when working with very large models.

You can adapt this convention or create your own; the only important thing is to be clear and consistent throughout all of your applications. You may need to rename elements as your application grows, if new functionality or processes may necessitate new identifying techniques. Adapt as necessary and always agree upon a technique before using it.

Group Dependencies

You should always try to avoid circumstances where work on a group project can be delayed because they are waiting on someone else to complete their own work. When planning a group project you should therefore clearly establish who will be working on each section of the application.

It is also important that you remember to make use of KnowledgeKube's Source Control feature and remind yourself to check your completed work back in to allow others to work on it. Plan to work through key sections methodically, complete them before moving on and adjust your plans as necessary if problems are encountered.

Client Browser Best Practices

When building a application you must always consider how it will appear on your site. Browsers all have different compatibility requirements and what works for one may cause issues in another. Avoid tailoring your application to specific browsers, instead ensure your application is compatible with all browsers. This way, your work will be consistent and require little to no alteration if the compatibility requirements change.

Some key points to bear in mind are:

- Test your application in each of the browsers as you build it - do not leave this until the very end!
- If you want to clearly separate terms in a filename, use underscores (_) instead.
- When starting an application make sure the site is up-to-date and that the application is at the same level of compatibility.
- When starting a new project, aim to have as much of the actual source data as possible. This will help avoid errors caused by altering earlier sections to account for the new data.
- You should also have a method in place for your end-users to report an issue to your team, to allow you to address it as soon as possible.

[THIS PAGE INTENTIONALLY LEFT BLANK]

2

Building an Application

With a good plan in place you can start building your application. It is important to understand how all of the elements in KnowledgeKube interact, so you can reduce the chance of errors or conflicts, and make efficient use of your time. The following chapters will explore the key areas of KnowledgeKube and give you helpful advice as you work on your application.

Saving Your Model

When working in KnowledgeKube it is essential that you save your work, and save it often. It goes without saying that losing hours or even days of work due to an error can be very frustrating. The **Save** button will post everything in the current model back to the database server, saving your work and making your changes available on the front-end.

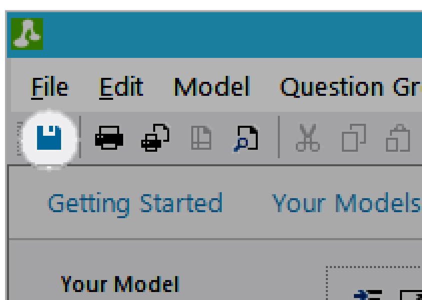

Figure 2-1: The KnowledgeKube toolbar, with the Save button highlighted.

You should save your work any time you make a significant change to avoid potentially losing data.

Equally important are the **Store Question** and **Save** features, visible in the bottom right of the main KnowledgeKube window whenever a question is selected.

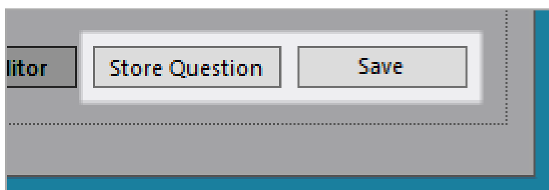

Figure 2-2: Store and Save Question options highlighted.

These buttons are used to perform the following actions:

- **Store Question** - Selecting this will store your changes in temporary memory for the duration of the session, instead of saving them. This allows you to safely test the question without putting the application at risk; if you are not satisfied with the outcome, simply refresh the model or close it to discard the changes.

- **Save** - When creating or editing a question you must select this option to save your changes. For example, if you add a question and all other relevant details then move on without selecting this option then all of your changes will be lost.

Refreshing Your Model

The **Refresh** function will clear all temporary data and revert your model to its last saved state, and as such it is critical that you save your changes before doing so as unsaved data will be lost.

This allows you to quickly discard unwanted or outdated changes and, if working on a group project, update the model to include changes made and saved by other users. For example, if another editor has added a new question group you will need to refresh your model before you can see it.

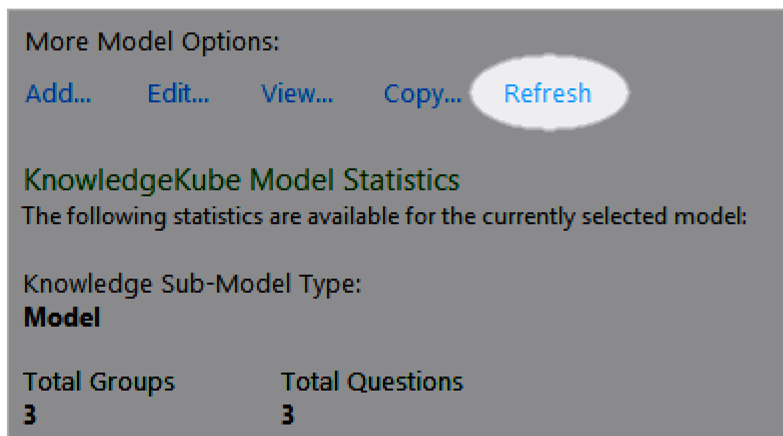

Figure 2-3: Refresh option in the Your Model tab.

Source Control

Making changes to many elements of a application requires that you first **Check Out** the elements for editing. This locks the element to your user profile meaning that no other user can edit or save changes to the element until you are finished and **Check In** the element again. Each element can only be checked out by one person at a time so don't forget which elements you have checked out!

The following areas of KnowledgeKube use source control:

- Question Groups
- Actions
- Rating Definitions
- Documents
- Data Sources
- Cargo

When working as part of a group project you should clearly establish a protocol for managing content that is subject to source control, such as when to check items back in and what to do when someone absent has left something checked out.

Furthermore, ensure you add clear notes every time you check something back in so that others are clearly able to see the change history of the item.

You can implement a bug cataloguing system using the Check In functionality. If you encounter a problem whilst working on a particular question group, when you check it in you can enter a code in the 'Reference' field of the **Confirm Check In** window. These references can then be viewed by clicking the **View History** button.

Using Expression Comments

The **Model Inspector** lets you view comments within your model. Different categories of comment are used to identify specific parts of your application that need attention or explanation. Use comments wherever possible, to provide information and structure.

- **CR** - Change Requests. This comment is applied to content that was added or modified as part of a task. Write a description of the change and include the reference number of the task in the comment. This is very useful for keeping track of your model as it grows, and helps other users to understand areas they haven't worked on.
- **TODO** - This comment is applied to unfinished areas or as a reminder where further information or resources are required. All TODO comments need to be resolved before the model can be released.
- **NOTE** - This comment is used to provide specific, important information about a particular area of a model. For example, explain complex expressions, give an overview of a process, summarise an action, and so on.
- **BUGFIX** - This comment is used if a workaround for a bug has been used, or if a previously bugged area now works properly. Explain the circumstances around the bug, including a bug reference if possible, and carefully review the comments for future fixes and follow-up work.
- **INFO** - This comment is used to provide generic information at certain points of a model. For example, if part of the model is limited by role, for example.

!	Type	Description
	BUGFIX	Site error prevents expression working correctly - connection details entered manually
	CR	Submit button added as requested
	TODO	Get updated database name for connection
	NOTE	Expression will establish connection to user database, update values, and refresh data source
!	INFO	This form is used to collect customer information. Users of all roles can see this page

Figure 2-4: Examples of expression comments in Task List.

Using the RedirectToUrl Function

If your application is particularly large or complex, it might be a good idea to split the content across multiple models. Doing this will allow you to develop and maintain the key parts of your site separately. This reduces the overall risk to your site, errors will be confined to their own model and as such much easier to identify and resolve.

You can direct a user from one model to another by using the *RedirectToUrl* function to call a Content Delivery **Page Link**. You should always use page links with this function, instead of literal model references or URL strings, as page links are much easier to maintain if the linked address changes.

You can also use a variable in place of a Page Link. For example, you could pass a URL or model reference from a data source table to the variable, or use a model response value.

In order to carry information from one model to another you can create a Cargo connection and a set of variables to hold the session data that you want to transport.

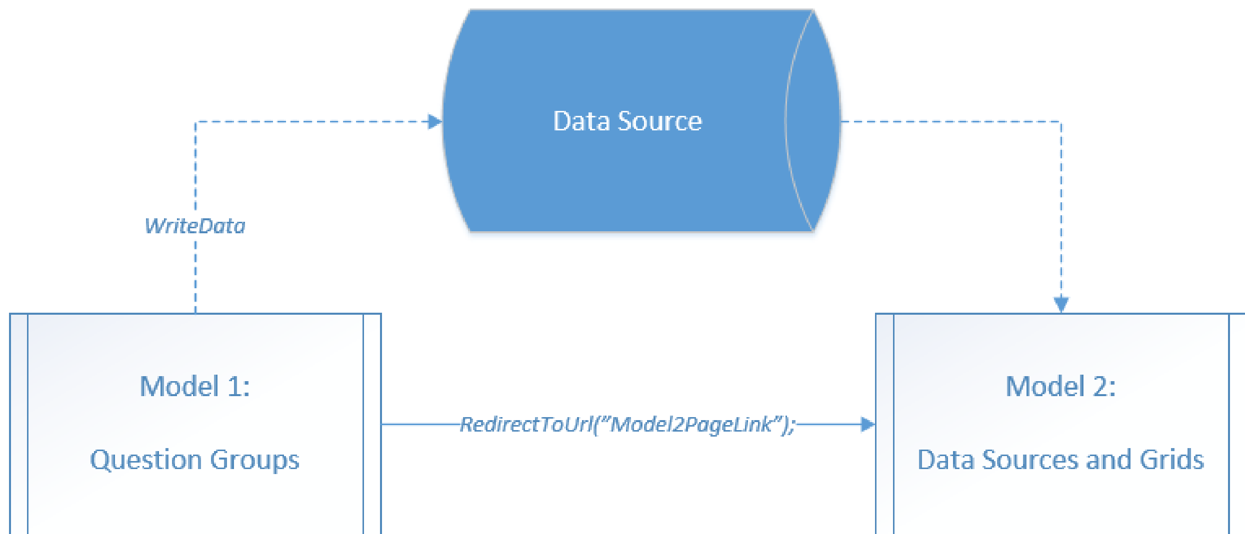

Figure 2-5: Example of a site built using two models.

The example above represents a site built using two models, the first of which collects information to pass to a data source, while the second model displays the contents of that data source.

Using Question Groups

Question groups are used to store the majority your application's content. Each type of group performs a different function or purpose. Refer to the main **KnowledgeKube** guide for complete information on the different group types, but a brief summary is as follows:

- **Form** - The standard question group type capable of containing all question types. You should add your questions in a linear, easy to follow order that is as close to how the questions will flow in the front-end as possible. This will help you to build it and allow others to easily understand the content and purpose of the group. Forms can consist of up to four columns through which the group's contents can be distributed.

- **Progress Indicator** - Progress Indicators render at the top of the page and are used to help the user navigate through the form by displaying a sequence of links—or 'breadcrumbs'—to previous stages. A model can only contain one of these groups and the group can only contain **Navigation Button** questions. You will need to carefully consider how many buttons to include and how they will direct the user, as well as the use of visibility and enable attributes to hide or deactivate the buttons as appropriate.

The position of a question group in the model is not important as the only way to move between them is to add buttons that call the *ShowForm* function. Similarly, a Progress Indicator group will behave the same no matter where it is placed in the model. As such, the placement of question groups is entirely up to you, but you should still assemble your groups in an order which makes it easy to follow and understand the general structure of the model. For example, whilst you could place the first and second question groups at opposite ends of your model it may not be easy for someone else to

understand that they're designed to follow each other. Doing this also makes it easier to assign navigation buttons as you can follow the order in the model.

The first question group you create will automatically be set as your application's **Startup Group**. If you wish to set a different question group as your application's startup group you can do so, but only if it contains at least one question.

Always bear in mind that for Form groups, the text in the 'Description' field is displayed as a header for that page (unless the group is contained in a placeholder) so ensure it is clear and descriptive.

Writing Question Text

When writing question text, remember that whatever you write will be displayed to the end user, and to ensure your application looks professional you should use a consistent case. When writing questions ending in a colon, use title case throughout (each principal word has a capitalised first letter). If a question ends in a question mark then it only needs sentence case (capitalise the first letter of the first word). For example:

- Number of Properties:
- Are you satisfied with your care?

When working with question types that have no visible question text—such as expressions or place holders—you can use the Question Text field to provide a description of the question's purpose or other important notes for future reference. In situations like this, the case isn't important as long as other administrators can easily understand what you've written.

Add to Expression Parser and Auto Post Back

When creating questions you have the option to add them to the expression engine, and the option to cause them to automatically post back to the data repository. Both of these options should only be used when necessary.

- Adding questions to the parser when they aren't needed will cause your application to allocate more memory to your application than it needs to, which will cause it to run less efficiently.
- A question should only post back to the repository when you need data to be updated in real time. For example, If post-back can wait until the user has completed all questions and confirmed the form by clicking a button, there's no need to have one occur every time they answer a single question. Doing so will cause the performance of your application to suffer.

Using Identifiers

Always ensure that your variables are being passed values in a format they can accept and that those values are also returned the same way. For example, a date passed to a variable as a string cannot be returned as a date. Similarly, some processes resolve *True* and *False* outcomes as 1 and 0, and vice versa. This can cause issues if you do not design dependant parts of your application accordingly. This is most important when using *WriteData* expressions - if the expression cannot write to a column because the source data is an incompatible format, the entire expression will fail.

Using Identifiers in Expressions

You can substitute any string value in an expression with the contents of a variable. For example, if you want to use the *WriteData* function to update a specific row in a data table, instead of putting a literal row ID into the expression, you could have a variable called *VARUpdatedRowID* that contains a number determined by the end user's selections.

By placing the variable's keyword inside the expression instead of the literal string value, your expression becomes far more useful. For example, the following *WriteData* expression uses the keyword of a variable in the *Data* argument rather than a manually specified CSV string:

```
WriteData(ColumnKeywordsVar, MyDataSource, False, "");
```

You can use a similar process if you need to use a semicolon in an expression. Because semicolons are read by expressions as being the end of a statement, you cannot use a semicolon for any other reason without the parser considering it a line break. To get around this, you can create a **Constant** whose value is simply a semicolon, then place the keyword of that constant anywhere you want the expression to read a semicolon.

For example, you could tell an expression to treat a semicolon as a CSV list separator by putting the constant's keyword - e.g. *CONSemiColon* - in place of ";". This process is explained in more detail in **"Writing Expressions"** on page 28.

Managing Variable Memory

There are two ways of initialising variables in an application:

- **Design-time** - Variables created via the **Model Identifiers** interface are initialised with a default value as soon as the application loads.
- **Run-time** - Variables created within the application using assignment expressions.

In both cases, the variable will use a portion of system memory from the moment it is initialised until the session ends or the variable is **Disposed** of, which clears any data it contained and removes it from memory. To dispose of a variable, assign it a **Null** value:

```
MyVariable:=Null;
```

To use variables, and therefore application memory, effectively, you must consider exactly when you need to use your variable and at what point it is no longer needed. Run-time assignments let you control when memory is required, and Null assignments let you clear memory when a variable is no longer needed. When your application is small this may not have much of an effect, but it will have significant impact as your application grows larger, as you add more variables, and as more memory is taken up by other processes.

We will look at an example of an application that uses both design-time and run-time variables. An insurance company sells a number of products through their website and use an in-built quoting system to provide customers with a premium. As a user progresses through a quote, their current premium needs to be displayed on every single page, and will be modified by certain selections the user makes. As such every page from beginning to end will have to refer to the variable containing the premium - *VARPremium*. It would be inefficient to repeat the calculations every time the value is required, so we will use a design-time variable and leave it until the session ends.

In the application there are also expressions that are specific to a certain page or product that also require a variable to store a value. That value will not be needed on any of the following pages, and some may not even be needed at all depending on the options the user selects. For example, the first page of each product section has an expression that calculates an additional fee for that cover type and adds it to the total premium.

One way of implementing this would be to create variables to store the premium and product-specific values:

Variables
Constants
Key Value Pair

Variable Name	Initial Value
V	
VARPremium	0
VARInsuranceModifierPets	2.5
VARInsuranceModifierHouse	1.5
VARInsuranceModifierCar	3.5

Figure 2-6: Variables created in the Model Identifiers interface.

Then use a series of *If* statements on the first page of each product section to modify the customers premium accordingly:

```
7  VARPremium:=if(VARProductType="Pets", (VARPremium * VARInsuranceModifierPets),
8      if(VARProductType="House", (VARPremium * VARInsuranceModifierHouse),
9          if(VARProductType="Car", (VARPremium * VARInsuranceModifierCar),0)));
```

Figure 2-7: If statements to modify premium.

This method is inherently inefficient - if the customer only selects one product, then the other two variables are wasted memory. The product variables are only used once - if the user continues to the next page, or selects another product entirely, that value is no longer required but the variable will remain in memory until the session ends. We should aim to only use the value we need, use it once, and dispose of it afterwards.

Furthermore, the expression engine will have to process every line of the expression before returning a value. A more efficient way of implementing this process would be to use run-time assignments inside compound *If* statements:

```
4   if:(VARProductType="Pets")
5   {
6       VARPremiumDifference:=
7           VARPremium*1.5;
8
9       VARPremium:=
10          VARPremium+VARPremiumDifference;
11
12      VARPremiumDifference:=
13          NULL;
14  }
15  if:(VARProductType="House")
16  {
17      VARPremiumDifference:=
18          VARPremium*0.5;
19
20      VARPremium:=
21          VARPremium+VARPremiumDifference;
22
23      VARPremiumDifference:=
24          NULL;
25  }
26  if:(VARProductType="House")
27  {
28      VARPremiumDifference:=
29          VARPremium*2.5;
30
31      VARPremium:=
32          VARPremium+VARPremiumDifference;
33
34      VARPremiumDifference:=
35          NULL;
36  }
```

Figure 2-8: Example of how to use a run-time variables.

We haven't used any design-time variables for the product modifiers, so their values are not in memory. Using a series of compound expressions, we assign and use their value when required and then immediately dispose of them again. This saves the application from having to initialise those variables when the application loads, and it also makes each individual part of the application run faster as there isn't any redundant information being maintained from page to page.

Furthermore, as we have used compound expressions, we are also reducing the amount of memory required as the contents of each expression will only be parsed if the main *If* statement resolves as true - if not, the contents will be skipped entirely.

If you are using a run-time variable in a conditional statement then there is the possibility that the variable will be created but never used. As such, you should incorporate the assignment expression into the conditional statement, so that the variable is only created when it is definitely required. Similarly, you will need to implement the assignment of the Null value so that you dispose of the variables once they are finished being used.

Please note, that if you include an assignment expression in a *While* statement, every iteration of the expression will pass a new value to the variable - even if the value is the same - effectively performing a continuous series of assignments until

the main expression is resolved. This will have an impact on the performance of your application. As such, in these circumstances you need to ensure that your assignment is performed separately before initiating a *While* statement.

Using Range and Matrix Definitions

Matrix definitions are used to store data in variables that is then returned by being compared against two or more inputs from the application. This same effect could be achieved by using nested expressions but as you start to include alternative conditions, or conditions with several parts, you will find your expressions growing larger. This will make your application more susceptible to error and difficult to maintain.

For example, consider the following expression used to assign a numeric value depending on the response to a multiple choice question:

```
If(Question1="Response_A", 1, 0);
```

In this example, if the user selects *Response_A*, the value returned will be "1", and if another response is selected the value returned will instead be "0". This is fine if there is a small set of potential responses, but as you add more elements for the expression to consider, you will need to expand by nesting additional statements inside the first, as shown below:

```
If(Question1="Response_A", 1, If(Question1="Response_B", 2, If
(Question1="Response_C", 3, 0)));
```

As you can see, expressions will grow quickly as you add further conditions. Including a second question with only two potential responses will require an even larger expression:

```
If(Question1="Response_A" & Question2="Response_D", 1, If(Question1="Response_
B" & Question2="Response_D", 2, If(Question1="Response_C" &
Question2="Response_D", 3, If(Question1="Response_A" & Question2="Response_E",
4, If(Question1="Response_B" & Question2="Response_E", 5, If
(Question1="Response_C" & Question2="Response_E",6,0))))));
```

Now, despite a relatively modest selection of responses, the expression has grown into something complex and potentially difficult to maintain. When expressions grow beyond a certain size, the possibility of an error occurring due to factual inaccuracy or incorrect spelling grows significantly. Adding another question with two possible responses will result in the following:

```
If(Question1="Response_A" & Question2="Response_D" & Question3="Response_F", 1,
If(Question1="Response_B" & Question2="Response_D" & Question3="Response_F", 2,
If(Question1="Response_C" & Question2="Response_D" & Question3="Response_F", 3,
If(Question1="Response_A" & Question2="Response_E" & Question3="Response_F", 4,
If(Question1="Response_B" & Question2="Response_E" & Question3="Response_F", 5,
If(Question1="Response_C" & Question2="Response_E" & Question3="Response_F", 6,
If(Question1="Response_A" & Question2="Response_D" & Question3="Response_F", 1,
If(Question1="Response_B" & Question2="Response_D" & Question3="Response_F", 2,
If(Question1="Response_C" & Question2="Response_D" & Question3="Response_F", 3,
If(Question1="Response_A" & Question2="Response_E" & Question3="Response_F", 4,
```

```
If(Question1="Response_B" & Question2="Response_E" & Question3="Response_F", 5,
If(Question1="Response_C" & Question2="Response_E" & Question3="Response_F", 6,
0)))))))))))));
```

As you might imagine, creating an expression in order to analyse an application that has dozens or even hundreds of response combinations will result in prohibitively complicated expressions. If the terms of the matrix altered and you had to update one or more outcomes, locating and changing that exact expression could be very difficult and likewise an error might not be easy to spot.

Similarly, you must also be careful when adding more axes to a matrix as it will become exponentially more complex to design. For example, a matrix with four axes, each receiving ten responses, would result in 10x10x10x10 (10,000) outcomes to manage. In these situations it might be more practical to pass the value from one matrix into another or to use an external data source.

Using Actions

There may be circumstances where you need to re-use a particular expression or series of expressions. Rather than write the same expression in multiple places, you can add them to an action and simply call the action using a *RunAction* function. This can be especially useful in nested expressions, as you can condense a potentially complex series of expressions into a single argument.

Expressions are covered in **"Using Expressions and Functions"** on page 27.

Using Attributes

Attributes allow you to modify the behaviour or appearance of questions and other content in your application. Certain types of content have attributes added to them automatically upon creation, while others you will need to add yourself.

You should limit, or preferably avoid, using attributes with expression-type questions and try to include as many commands within the expression itself. Similarly, try to avoid using attributes on buttons if they do not control the look or visibility of the button. As many commands as possible should be included within the button itself rather than as attributes, in order to make the expressions easier to execute and reduce processing requirements.

Planning the Application's Appearance

From the beginning you should have a good idea of what you want each page of a application to look like, from composition to final appearance. This is especially important if you are making a mobile version of your site, as it will be easier to encounter issues with size and spacing. You should also consider that certain styles, like the data grid tiles, can dominate a page and be more visually striking than normal.

Question groups can have 1-4 columns in which to place their contents. All new questions added to a group appear in the first column. By default, each column will have the same width, so the contents of those columns will be resized to fit the space provided. Similarly, by default each type of question will occupy a pre-set vertical space in the chosen column. This

can be overridden by applying special styles, so each question will be spaced equally to ensure content in different columns lines up appropriately.

Using Style Classes

You can use **Style Class** attributes to change the appearance of certain types of KnowledgeKube content. The class determined by these attributes must exist in the CSS (Cascading Style Sheet) of the site that hosts your application. The maintenance of these style sheets should only be undertaken by web designers or other individuals who are comfortable editing the CSS mark-up. The full technical guide to styling is on the **KnowledgeKube Extended Library** site at referencestyles.onknowledgekubesandbox.co.uk/home.

The majority of style classes provide minor aesthetic changes, such as sizing, colour, placement etc. but some may significantly alter the appearance and functionality of an element. For example, a data source grid can be converted from a table to a series of interactive tiles, such as the one below:

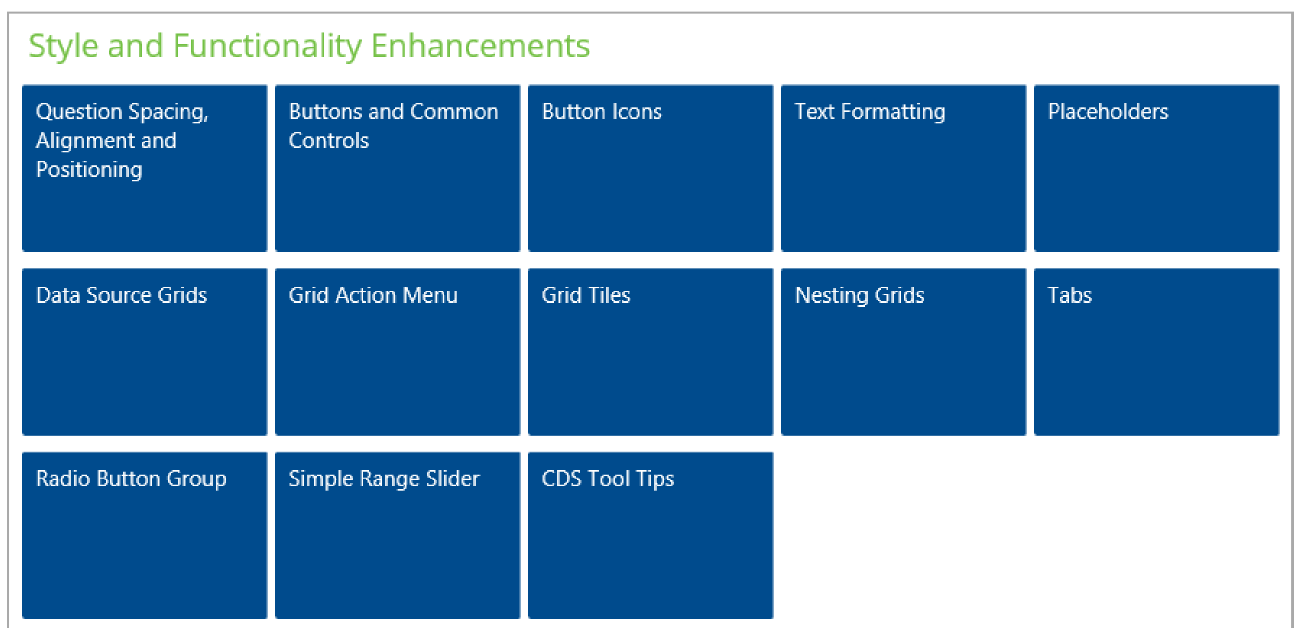

Figure 2-9: Data grid tiles.

Multiple choice questions can be redesigned so that instead of appearing as drop-down menus, they present the user with a slider for selecting one of several potential responses:

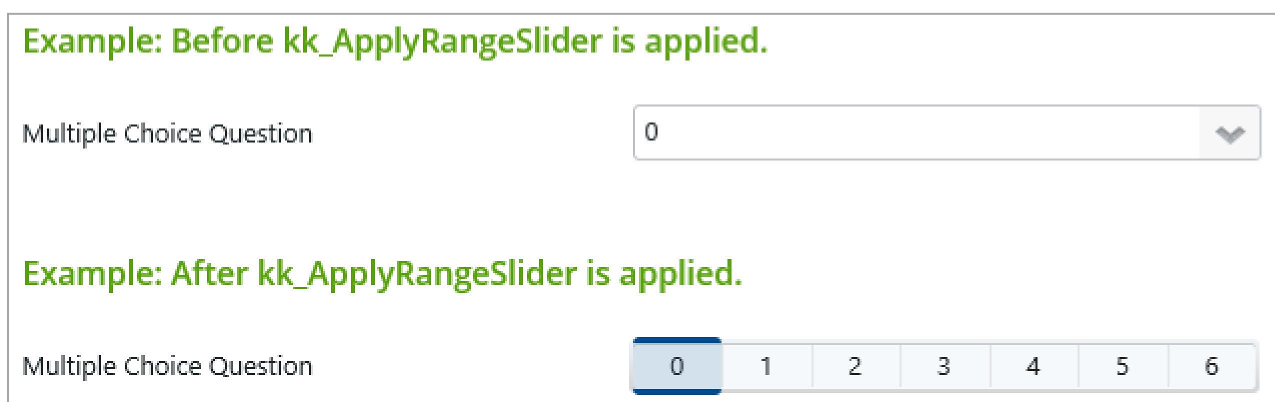

Figure 2-10: Multiple choice question becomes a slider.

If a multiple choice question contains a large number of responses it is recommended that you keep using the default drop-down menu appearance to ensure all of your responses are clearly shown on the page.

You should try to remain consistent with the style classes you apply throughout the application. For example, if you apply a style to multiple choice questions in one question group, apply the same style to every question of the same type within your application.

Although style classes can be combined to produce multiple effects, you should always consider potential side-effects of doing this.

4 Column Layout

Back

Four Column Multiple Choice	Four Column Multiple Choice	Four Column Multiple Choice	Four Column Multiple Choice
0 25 50 75 100	0 25 50 75 100	0 25 50 75 100	0 1 2 3 4 5 6 7 8 9
Tool Tip	Tool Tip	Tool Tip	Tool Tip
Watermark (tool tip title).			
Button	Button	Button	Button
Free Text	Free Text	Free Text	Free Text
Checkbox	Checkbox	Checkbox	Checkbox
Yes/No	Yes/No	Yes/No	Yes/No
Yes No	Yes No	Yes No	Yes No
Radio Button Group	Multiple Choice	Multiple Choice	Multiple Choice
Grapefruit, Pear, Orange, Apple	--Please Select--	--Please Select--	--Please Select--
	Read Only Text	Read Only Text	Read Only Text

Figure 2-11: A four-column page with numerous question types and styling options.

Using Read-only Text Decoratively

Read Only Text questions are used to deliver plain text, optionally paired with a response value. When rendered, this type of question consists of two parts:

- **Question Text** - The **Question Text** specified in the question's **Properties Panel**.
- **Question Control** - A label determined either by the **Fixed Response** field in the question's Properties panel, or by a mapping attribute assigned to the question.

Special **Style Class** attributes can be assigned to read-only text questions, altering their appearance to some extent. Certain classes only affect the question text, while others can affect one or both parts of the question. In the latter case, the style class attribute must be modified to include square brackets containing a reference to the part you want to style.

For example, the *kk_DisplayLargeSizeText* class can be applied to either the question text or the question control. To affect the question text, you'd need to write the attribute's value as follows:

```
kk_DisplayLargeSizeText[QuestionText]
```

To apply the same style to the control, add a style class attribute with the following value:

```
kk_DisplayLargeSizeText[QuestionControl]
```

If you want both parts of the question to be affected by this class, you'd need to assign both of the attributes described above.

Example: No styles applied.

Read Only Question Mapped Variable Text

Example: After kk_DisplayLargeSizeText[QuestionText] is applied.

Read Only Question Mapped Variable Text

Example: After kk_DisplayLargeSizeText[QuestionControl] is applied.

Read Only Question Mapped Variable Text

Example: After kk_DisplayLargeSizeText[QuestionText] and kk_DisplayLargeSizeText[QuestionControl] is applied.

Read Only Question Mapped Variable Text

Figure 2-12: Examples of how the question control and text is affected by style classes.

You may want to use this combination of question and style attributes to create headers within your application. Whether you do this instead of deploying markup-based headers inside **Content Block** questions is dependent on the intended structure of your application, and how comfortable you are writing HTML. If you have a dedicated design team to create style classes for you, and you would prefer to simply assign those classes to your content using attributes, read-only text questions are the better option.

By default, the question text of a read-only question will leave enough page width for the question's control, even if no value has been assigned to that control.

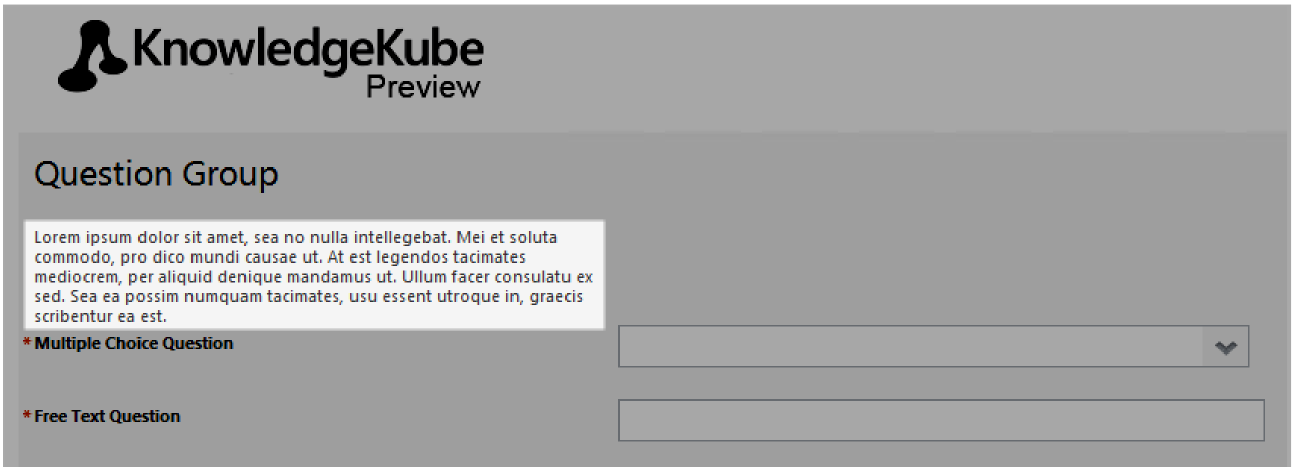

Figure 2-13: A Read Only Text question with no style applied, and no value mapped to its control.

In situations where no control value is needed, apply the following class to stop the empty control from being rendered:

```
kk_HideQuestionControl
```

If you would prefer the question text to span the entire width of its group column, apply the following class:

```
kk_QuestionTextToFillColumn
```

If you do this without also using *kk_HideQuestionControl*, the control will appear below the question text instead of alongside it. If *kk_HideQuestionControl* is used, the question text will appear on its own, as though it were a single block of text.

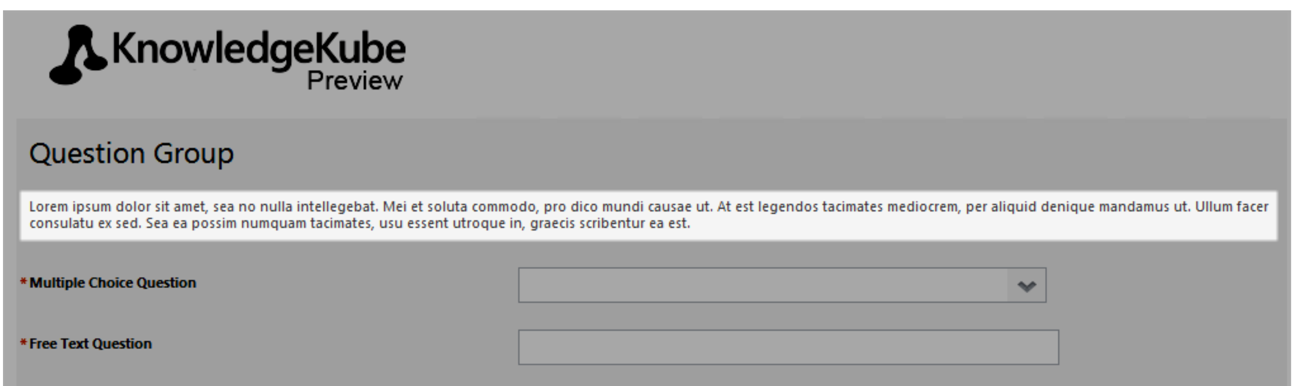

Figure 2-14: A Read Only Text question with both kk_QuestionTextToFillColumn and kk_HideQuestionControl applied.

For this reason, if you intend to use your read-only text question as a heading, you should assign attributes containing the *kk_HideQuestionControl* and *kk_QuestionTextToFillColumn* classes, and one or more attributes such as *kk_DisplayLargeSizeText[QuestionText]* to change its format.

Using Document Images

When you upload a dynamic template to KnowledgeKube, any embedded images will also be uploaded. When the document is saved, the images are automatically extracted and saved to a new folder so they can be accessed every time the document is generated.

You can set a location for KnowledgeKube to save these images, which should be a globally accessible folder on your network to store the images, allowing all users to have access to them.

An embedded image will be duplicated in the specified location and given a new name, which is derived from the documents keyword and the number of images previously imported from it.

You should not change the name of an image after the document has been uploaded, as the document's XML structure will be set with the specific image names and file paths. Changing the name of an image will cause this link to break, which could lead to the image not appearing in the document and possibly being overwritten or lost.

If you do change the name of an image you may need to remove and re-add a document in order to allow the XML structure to map correctly. Only then will the image appear in the designated location.

If you want to replace an image with another, you can delete the original image and upload another with the exact same file name and file path. This will place the new image in the exact same spot as the old one.

[THIS PAGE INTENTIONALLY LEFT BLANK]

3

Using Expressions and Functions

You will use expressions throughout your application so it is important to understand how they are used, how best to write them and other useful information.

The different types of expression and their roles are covered fully in the **KnowledgeKube User Guide** and **Expression Cookbook**.

Always aim to write your expressions as efficiently as possible, while maintaining a neat layout that is easy for other administrators to read. If you need to execute a series of expressions simultaneously, try to keep them all in the same place, such as within a button or action, as this will make it quicker and easier to execute them. Each command - or **Statement** - within an expression requires processing power, so calling expressions spread throughout an application will increase processing time. In small applications this will not be noticeable but it can become a serious problem as your application gets larger.

Use the correct expression type for what you want to achieve. For example:

- A **Form Load Expression** question will be triggered each time the host question group is loaded. As such they must **never** be placed in an Expression Group, as this will cause an endless loop where the expression executes itself, eventually causing KnowledgeKube to crash. For the same reason, do not place a *RunExpressionGroup* expression inside the group it executes.
- The *SaveFormData* function will save the current model's response data to a database within KnowledgeKube. Use this expression at every key point in your application, such as after an important question group or on a 'Submit' button at the end, whichever suits you best.

The Expression Engine

The expression engine is the core logic processor within KnowledgeKube. It is capable of executing complex commands ranging from simple arithmetic to complex, multi-line nested operations that execute hundreds of functions to generate and return data from throughout a model. The engine provides dozens of functions for constructing your expressions, a complete list of which can be found using the IntelliSense feature in the **Expression Editor**. KnowledgeKube expressions must be written with due care, because mistakes will cause an expression to behave unexpectedly or otherwise fail. You should always take the time to plan how your expression will work together with the rest of your application - efficient expressions mean better overall performance.

Unless deliberately designed to the contrary, expressions are executed from top to bottom. This means the first statement in an individual expression will run first, followed by the one below it and so on until the expression is complete. The same goes for expressions inside question groups - when they are triggered, the one nearest to the top of the question group order will go first. Most importantly, an expression only returns its value once it has executed successfully and if any

expression in a sequence fails, then all subsequent expressions will not be executed and all prior ones will not return their values.

Examples of why an expression may not execute correctly:

- Incorrect use of brackets.
- Incorrect or misspelt Keyword.
- Missing semi-colon at the end of a statement.
- Semi-colon within a statement.
- Incorrect formatting (wrong operators, misspelled functions etc.).
- A preceding expression, or a command in the same expression, not executing successfully.
- A dependency on values that are not present, either because they haven't been generated yet or another operation has failed.

You can use the Error Handling functionality and the **Form Trace** feature in the Preview tool, to highlight any expressions that fail and how. For more information about the preview tool, refer to the KnowledgeKube guide. Error Handling is covered in more detail in **"Error Handling"** on page 37.

You can use *Stop* and *Exit* statements to specify conditions in which a compound expression can fail, without affecting the rest of the sequence. Statements like these are very context-sensitive and should only be used once you understand them properly. Refer to the **Advanced Expression Cookbook** and **"Error Handling"** on page 37 for more information about using these expressions.

Writing Expressions

Here are some key concepts to consider while writing expressions:

- The best way to write expressions is using the **Expression Editor**. While expressions can be entered directly into the **Expression to Evaluate** field, using the editor allows you to check the outcome in real-time and to use Intellisense. To help you further, you can separate the **Expression** and **Output** tabs so they are both open in the same window, and you can update the Output tab simply by clicking anywhere within it.

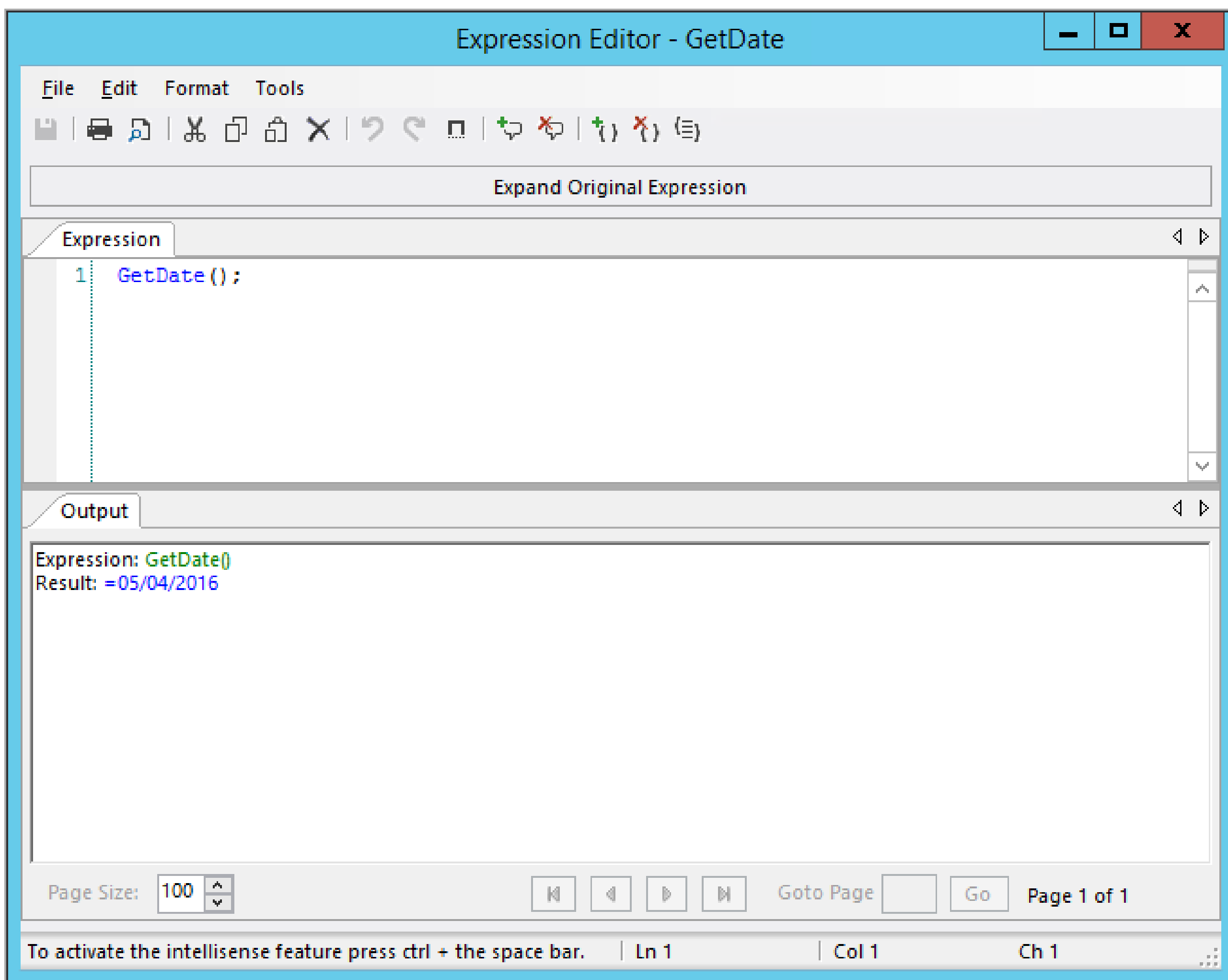

Figure 3-1: Expression Editor with Expression and Output tabs open in the same window.

- If your main expression is dependant on certain values being available before it executes, aim to process those calculations before the main expression. This will help to prevent errors with the expression itself and also allow the main expression to execute quicker.
- If you find yourself repeating large expression blocks, use the **Snippets** feature to store and re-use expressions.
- Parts of an expression can be commented out using double slashes (/) or selecting the option at the top of the window. Content marked this way is not processed by the expression engine and can be used to temporarily remove lines from the sequence or add notes for reference. Please note that commented-out content still needs to be appended by a semicolon.

```
 6  //Role-based authentication;
 7  if(IsUserInRole(GetAuthenticatedUser(),"Approver"),1,0);
 8  ShowForm("ApproverGrids");
 9
10  //This group disabled for testing;
11  //ShowForm("SummaryPage");
```

Figure 3-2: Expression Editor showing a commented out note and expression.

Multi-line Expressions

The editor makes it easier to write very large multi-line expressions. There may be times when expressions like this are required and a consistent writing technique will avoid mistakes and also help others working on your application

understand and edit them if necessary. Take the following example:

```
Expression | Output
1  PolicyInstallmentAmountVar:=if(PolicyLengthVar = 3, PaymentPlanOneVar,
   if(PolicyLengthVar = 4, PaymentPlanTwoVar, if(PolicyLengthVar = 6,
   PaymentPlanThreeVar, if(PolicyLengthVar = 12, PaymentPlanFourVar,0))));
```

Figure 3-3: Sample expression as viewed in the editor.

This expression consists of a series of *If* statements to determine which value to assign to the *PolicyInstallmentAmountVar* variable, based on the value currently assigned to another variable called *PolicyLengthVar*. This is more efficient than writing four individual statements, but when written as it is above, it may be a little hard for other people to understand.

Another way of writing this sequence is:

```
12  PolicyInstallmentAmountVar:=
13          if(PolicyLengthVar = 3, PaymentPlanOneVar,
14                  if(PolicyLengthVar = 4, PaymentPlanTwoVar,
15                          if(PolicyLengthVar = 6, PaymentPlanThreeVar,
16                                  if(PolicyLengthVar = 12, PaymentPlanFourVar,0))));
17
```

Figure 3-4: Expression with separated, indented arguments.

In this version each individual argument within the expression has been separated and indented. This will not affect the operation of the expression in any way. The expression parser does not treat spaces, carriage returns, tabs etc. as line breaks, so expressions like the one above will be read as one continuous line. Written this way the sequence is easier to follow and each argument can be viewed individually to easily identify errors. Furthermore, closing the brackets at the end of the expression is now easier, by simply counting the open brackets at each indent.

Optional Parameters

In expressions with multiple optional parameters, you must not omit parameters you don't need before using ones that you do. For example, if a function has one mandatory and three optional parameters, and you omit the first two optional parameters completely, it will interpret the third optional parameter as actually being the first, and the second parameter in the function overall.

```
Function(Parameter1, OptionalParameter4);
```

In these cases, you will need to use an acceptable replacement, such as an empty string, in the parameter, to allow the parser to read the order and content of the arguments correctly.

```
Function(Parameter1, "", "", OptionalParameter4);
```

Bear in mind that some optional parameters may reference others and that some functions may require default values for optional arguments rather than empty strings.

Using If-statements

If statements can be written as a normal function but they can also be used like this:

```
if:(InsuredPropertiesVar>1)
{
ShowForm("SecondHomeQuestions");
}
```

This method is referred to as a compound statement, which is covered in **"Compound and Labelled Statements"** below. Please note that when written this way there is no need for a 'false' condition; the expression will only ever carry out the successful command or do nothing at all.

If you need to include alternative outcomes you can use the *Else* statement. For example:

```
if:(PremiumVar>250)
{
ShowForm("SummaryPageGroup");
}

Else

{
ShowForm("AdditionalOptionsGroup");
}
```

Using *Else* statements you can add more alternative outcomes than an in-line expression is able to, which can only use a maximum of two. However, always bear in mind that the expression will become increasingly complex the more alternatives that you include. Use in-line expressions where a basic true/false outcome is desired, and *Else* statements in a compound expression where multiple possible outcomes are desired. If your expression becomes too complicated, consider using *Or* statements instead or even implementing matrix definitions.

The composition of expressions with multiple possible outcomes and knowing which to use is very important. Remember to use the correct type of *If* statement wherever necessary - a compound expression cannot be used to pass a value to a variable, nor can an in-line expression mimic the behaviour of a compound statement without substantial and overly-complex nesting.

Furthermore, an in-line expression will exit as soon as the result has been satisfied. Even if it were hundreds of lines long, it would resolve quickly if the required condition is met in the first few lines. A compound expression will always have to process every single line to ensure that no other conditions could resolve the expression.

Compound and Labelled Statements

Compound and labelled statements allow you to extend the ability to control the logical flow of expressions, allowing for more flexible and intuitive multi-line expressions to be used within applications.

When using compound and labelled statements there are certain things that are important to bear in mind:

- *JumpTo* should not be used for initiating loops. *While* should be used instead.
- *While* loops should be as specific as possible to ensure they start and stop exactly when required, without looping more or fewer times than required.
- *Stop* statements will terminate all expression execution, not just the compound statement they are in, so be careful when using them as you may prevent necessary parts of an expression from executing.
- Similarly, *Exit* statements may terminate expression flow before all the necessary parts of the expression have been executed.
- Never loop a statement without including a condition to stop/control it, such as *Stop* or *Exit*. Otherwise the expression will execute constantly and slowdown the application, eventually causing it to crash.
- Compound expressions cannot pass a value to a variable as they do not generate a result of their own, assignment expressions will have to be included within the compound expression itself.

A labelled statement is an isolated expression, known as a **block**, that is called within a compound statement using a *JumpTo* statement. For example:

```
Expression | Output
1 JumpTo:TestExpression;
2
3 #TestExpression:
4
5 if:GetAuthenticatedUser="Administrator"
6 {
7   ShowForm:="AdministratorPage";
8 }
```

Figure 3-5: Example of a labelled statement.

In the above example, *TestExpression* is a labelled statement. When the block is called using the *JumpTo* statement, if *GetAuthenticatedUser* equals Administrator the user will be taken to the Administrator Page, and the expression will terminate.

A block name is always contained between a '#' and ':' and cannot contain spaces or other inappropriate characters. The block can contain as many expressions as necessary - the block could even call other labelled statements - and is only executed when called by the *JumpTo* command.

Including Semicolons in Expressions

Ordinarily, you cannot include semi-colons anywhere within an expression as the expression parser uses semi-colons to denote line-breaks. If you are writing an expression that requires a semi-colon within one of its arguments, you can use a constant with a semi-colon as its value in the argument instead. The expression parser will not interpret this as a line-break and the semi-colon can be used correctly. For example, the following pair of functions are used to build a list of email addresses. First, an email address is entered by a user and passed to a variable called *RecipientCCAdd*:

```
RecipientCCAdd:=
Concat(" ",RecipientCCAdd);
```

An empty space has been added to the beginning of each email address string using the *Concat* function. As multiple users enter their address, a string is built up out of the individual addresses:

```
CCChainVar:=
CSVListAddRow(CCChainVar,RecipientCCAdd,",",CONSemiColon);
```

Because email addresses need to be separated by semi-colons in order to work correctly in an e-mail client, a semi-colon will need to be included in the string. This is achieved by creating a constant called 'CONSemiColon' with a semi-colon as its value and using that as the *RowSeparator* argument in the *CSVListAddRow* expression. After multiple iterations of these functions, a string will be compiled in the *CCChainVar* variable as follows:

```
CCChainVar = " emailaddress; emailaddress; emailaddress; ... "
```

Please note that the *ListSeparator* argument has still been specified in the *CSVListAddRow* expression for clarity. Because each row only has a single item the list separator does not actually appear, if the argument was omitted it may be unclear to other users what effect this is having.

When to Use 'If' and 'Or'

There are cases when it is preferable to use an *Or* expression instead of a nested *If* expression. Which expression to use depends largely on whether or not the result remains the same depending on the condition. A simple rule for when to use each expression is:

- **Nested If** - Different qualifying conditions that produce different outcomes.
- **Or** - Different qualifying conditions that produce the same outcome.

For example, the following *If* expression returns a different result depending on the values of three variables:

```
if(Variable=1,1,if(Variable=2,2,if(Variable=3,3,0)));
```

If the conditions change but you want the outcome to remain the same, you would write the expression like this:

```
if(Variable=1,1,if(Variable=2,1,if(Variable=3,1,0)));
```

This is repetitive and inefficient. It will be simpler to rewrite the expression as an *Or* expression instead:

```
if(Variable=1|Variable=2|Variable=3,0,1);
```

The same three variables are used but only one outcome is needed.

When and How to Use Ampersands

Ampersands (&) are used in expressions in two different ways:

- To combine two or more string values together.
- As an *And* operator.

The following example shows how three strings can be concatenated into one; a first name, an empty space and a surname:

```
"Joe" & " " & "Bloggs"
```

When an ampersand is used in this way, it acts as a connector, adding the different strings together to form a single new one:

```
Joe Bloggs
```

Using an ampersand as a logical *And* operator allows you to check the logical value of two expressions and return a value of True if both expressions also return a True value. For example, the following expression will only evaluate as True if the *FirstLetter* variable holds the value "A" and *SecondLetter* holds the value "B":

```
(FirstLetter="A") & (SecondLetter="B");
```

When you use an ampersand in this way you must use parentheses to separate the individual expressions you are evaluating. Otherwise, the parser will not consider the expressions as logical units, and the result will be incorrect.

Repeated Expressions

If you are using an expression repeatedly throughout your application, and it's value is never updated, then you should execute the expression once and pass the outcome to a variable. You can then use the variable in any dependant expressions instead of copying and repeating the expression each time. If the value needs to change, you will have to re-evaluate with a new expression wherever necessary and pass the new value to the variable.

For example, the following expression is used to compare a customer's total premium against a very large series of smaller, month-by-month comparisons in order to determine if they are eligible for a free upgrade to their policy:

```
if(TotalPremiumVar > 5500,
if((JanPremiumVar > 1500) & (NoClaimsVar < 3))|
((JanPremiumVar > 2500) & (NoClaimsVar < 2))|
((JanPremiumVar > 3500) & (NoClaimsVar < 1))|
((FebPremiumVar > 1500) & (NoClaimsVar < 3))|
...
((DecPremiumVar > 3500) & (NoClaimsVar < 1)), ShowForm("FreeUpgrade"));
```

The expression is executed in every group of the model where the 'Total Premium' value is referenced. As the model is designed to accommodate every type of customer it will be used in groups that certain customers will never see. This will be laborious and increases the risk of errors. Instead, rewrite the expression as follows:

```
FreeUpgradeVar:= if(TotalPremiumVar > 5500,
if((JanPremiumVar > 1500) & (NoClaimsVar < 3))|
((JanPremiumVar > 2500) & (NoClaimsVar < 2))|
((JanPremiumVar > 3500) & (NoClaimsVar < 1))|
((FebPremiumVar > 1500) & (NoClaimsVar < 3))|
 ...
((DecPremiumVar > 3500) & (NoClaimsVar < 1)), 0);
```

Store the expression in a **Form Load Expression** question so it is executed when the model first opens. Then use the following expression whenever you need to use the outcome:

```
if(FreeUpgradeVar = 1, ShowForm("FreeUpgrade"),0);
```

It is now a lot easier to determine the purpose of each expression. You will now be able to tell if *FreeUpgradeVar* received a value and if it didn't then you know the error is in the form load expression. You will only have to correct it once rather than checking every iteration of the expression in the model.

[THIS PAGE INTENTIONALLY LEFT BLANK]

4

Error Handling

Every system, such as a KnowledgeKube application, must have good **Fault Tolerance**. This is the term given to the degree to which a system is affected in the event of an error (not how easy it is for an error to occur). If an error prevents any further action, or even crashes the program, this is referred to as **Failing Fast** and puts data at risk of loss or damage. Not only that, it makes your system appear vulnerable and unprofessional.

A good understanding of the potential weak points of your application and effective management of errors will allow a application to continue operating while the error is being managed. This is referred to as **Recovery**, and a application's ability to recover from errors is crucial to its integrity and security.

Detecting and managing errors in your application, both as you build it and after it is published, is known as **Error Handling**.

How KnowledgeKube Handles Errors

The default level of error handling built into KnowledgeKube is known as **Platform-Level** error handling. This is configured by a developer during the installation of KnowledgeKube and gives users no ability to directly manage errors, and a limited capability to fix them. You can see one aspect of error handling in the form trace of the **Preview** tool.

There are two elements to platform-level error handling:

- **Error Log** - Errors are identified and the details are emailed to the Mercato development team. If there is no developer environment for Mercato to use, this is the only way of reporting and resolving errors. If there is no email functionality in place, or the email system is the cause of the error, this will make resolving issues much more difficult.
- **Error Table** - Errors are identified and written to a database within KnowledgeKube, that can only be accessed and managed by a Mercato developer. This is still useful if email functionality is not present as the errors can still be identified.

As you can see, platform-level error handling is very limited and almost entirely reliant on external assistance. You can design an error-handling system within the application that gives you a lot more control over detecting and managing errors. This is referred to as **Model-Level** error handling, and is driven by a series of expressions that give you complete control over how errors are detected and managed.

Exceptions and The Stack

The expression engine will automatically detect and report errors (exceptions) as they occur in the **Stack**, which is the data layer within KnowledgeKube where all calculations take place. You can see the information the stack produces in the following places:

- The **Output** tab of the **Expression Editor**.

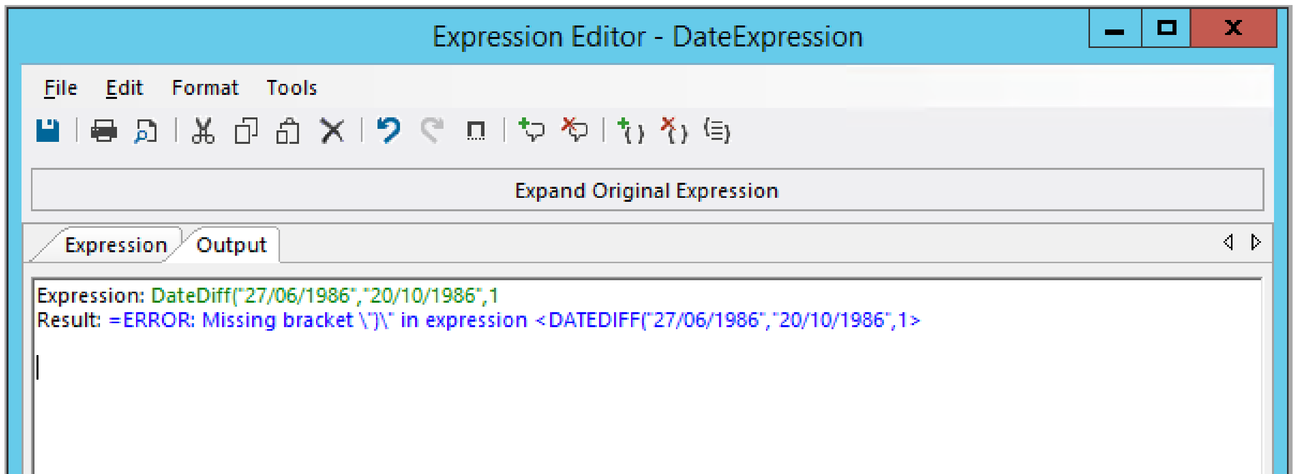

Figure 4-1: The Output tab showing an expression error.

- The **Form Trace** window in the **Preview** tool.

Figure 4-2: The Form Trace in the Preview showing an expression error.

Error messages are often quite detailed and this can lead to security concerns, as deliberately prompting errors to produce these messages is one way that malicious users can attack and compromise your site.

When you write error messages, consider that a front-end user only needs to know that an error has occurred, and not any specific details of what caused the issue. Make sure that messages are generic enough to simplify errors for front-end users and prevent that potentially detailed information regarding expressions or data processes are revealed. For example, "Oops! Something went wrong!" is a good non-specific error message for a whole range of potential errors.

Model-Level Error Handling

Model-level error handling differs from platform-level error handling in that a bespoke error management system is designed into the model using expressions and other KnowledgeKube functionality. This enables your model to react to errors and take appropriate action without interrupting the session. This also helps to detect errors that don't necessarily cause significant issues and may otherwise go undetected. Taking action within the model like this is known as **Catching** an error.

Effective error handling revolves around three key elements:

- **Error Expression Code Block** - A standard format for error handling expressions that you can re-use throughout your models. Refer to **"Error Expression Code Block"** on the facing page for more information.
- **Shared Error Logging Data Source** - An external table for storing and viewing error details. The table should be connected to a data grid which is hosted in a dedicated model or in a specific question group in your main model. Example field names can be seen in the *WriteData* expression in Figure 4-3 on page 40.

- **Error Page** - A page that the user will be directed to in the event of an error. You can set this up to let them return to an earlier point in the model from which they can continue working. You could also create an optional 'Contact Us' system, which lets users submit specific details about the error if they want to.

Model-level error handling is not compatible with **Data Flow**. A process package is self-contained, and you will have to refer to the 'Recently Executed' log in the tool itself to determine where an error occurred.

Error Expression Code Block

The most important thing to do is use a consistent format for your error handling expressions. You will need to use these expressions multiple times throughout a model and they cannot be stored in an action. As such, you should create a template for the error handling expression that you store in a dedicated model or file. You can then copy the expression block and place it wherever it is needed.

You should place the error handling expressions at key points in your model, wherever numerous or complex operations take place, such as:

- Operations involving filters, databases or other external data.
- Form Load expressions.
- Whenever you use *ShowForm* or a 'Submit' button.
- Whenever an action is called.

An example of a complete template:

```
1  OnErrorJumpTo(ErrorDescription,"True"):ErrorHandler;
2
3  ModelName:="ModelName";
4  ActionQuestion:="Question: AuthenticateUser";
5  LocationOfError:="ShowForm";
6
7  //-----------------------------------------------------------------------------;
8
9  #ErrorHandler:
10 {
11 DateRaised:=GetDateTimeFormat("","");
12 ErrorRaisedBy:=GetAuthenticatedUser();
13 IsActive:=1;
14
15 ErrorReference:="SiteName" & substr(NewId(),0,5);
16
17 //Write data to SystemErrorsDatasource;
18
19 SystemErrorId:=WriteData("ModelName,ErrorReference,ActionQuestion,ErrorRaisedBy,
   DateRaised,LocationOfError,ErrorDescription,IsActive","","SystemErrorsDatasource",
   "False","{0:yyyy-MM-ddTHH:mm:ss}");
20
21 ShowForm("ErrorPage");
22
23 Stop;
24 }
```

Figure 4-3: Example of an error handling expression block, including a WriteData expression.

In this example we have separated the *OnErrorJumpTo* expression from the *ErrorHandler* labelled statement. We assign a series of key details to variables and direct the user to a new question group. We have specified the application and exact question that caused the error. We have also recorded time and user details as well as an 'IsActive' indicator, which is a simple numerical value to show that the bug is active. When the bug is dealt with and resolved, this field will be updated in the table accordingly.

The 'ErrorReference' is a unique identifier made up of the site name and a GUID. In this case, our team manages multiple sites and uses numerous models for each site, so clear separation and logging of bugs at that level is required. Finally, the user is directed to a specific question group. This group will typically contain a generic message advising the user that an error has occurred and that they will be taken back to an earlier point in the site.

You can use the *RaiseError* function to deliberately prompt an error to test your error handling procedures.

Error Steps

When you are using very large expressions, simply knowing it has failed may not be enough as you will have to check hundreds of lines of expressions for the error. Instead, you should break the expression up into key sections and pass a value to a variable relevant to the step, to allow you to track the expression's progress. This will not affect the operation of

the expression at all but when the expression fails you can check the value of the variable to see which section was the last to pass a value to it; that section is the one that failed. For example:

```
1  OnErrorJumpTo(errMsg,true):ErrorHandling;
2
3  ErrorStep:=1;
4
5  if:(MainStatus = 0)
6  {
7  DateCreated:=GetDateTimeFormat("","");
8  RequesterName:=ReplaceTextWithPattern(GetAuthenticatedUser(),AuthenticationPattern,"");
9  VariableApprover:=CostCenterApprover;
10 AssignedApprover:=CostCenterApprover;
11 VariableCostCentre:=CostCentre;
12 };
13
14 ErrorStep:=2;
15 DefaultAddress:=if(AddressesFound = 0 | UseThisAddress = "No",DeliveryAddress,DefaultAddress);
16
17 if(AddressesFound = 0,WriteData("DefaultAddress,RequesterName","","DEFAULTADDRESS_DS"),
18 if(IsMatch(UseThisAddress,"No") & GetFormMode() <> 2,WriteData("DefaultAddress","DefaultAddressId","DEFAULTADDRESS_DS"),0));
19 GetDataSourceRow("DEFAULTADDRESS_DS", "DefaultAddressFilter");
20
21 OrderDeliveryID:=if(IsMatch(UseThisAddress,"Yes"),DefaultAddress,DeliveryAddress);
22
23 if(IsMatch(PreapprovedRequest,"No") & MainStatus = 0 & GetFormMode() = 1,updateStatusAndText(RequiresApproval),
24 if(IsMatch(PreapprovedRequest,"Yes") & MainStatus = 0 & GetFormMode() = 1,updateStatusAndText(AllocatedToProcurer),0));
25
26 ErrorStep:=3;
27
28 Onbehalfofusername:=UserResults;
29
30 BehalfOfUser:=if(IsMatch(VariableWhoIsTheRequestFor,"Other"),UserResults,"N/A");
31
32 if(RequestFor = "Someone else",GetDataSourceRow("KnowledgeKube_DS_OnBehalfLookUp","LookUpOnBehalfUser"),0);
33
34 if(MainStatus = 0,updateStatusAndText(RequiresApproval),0);
35
36 if(UseThisAddressResult = 0 | AddressesFound = 0,GetDataSourceRow("AddressesTableDefaultList","GetDeliveryLine1"),0);
```

Figure 4-4: Example of a large expression split into steps.

The above expression block has been divided into three stages using an 'ErrorStep' variable. If the expression failed in step 2, then the variable would have a value of 2 as the expression did not reach step 3. Combined with the other error handling information this will help you identify and resolve the error. You could also change the numbers to specific terms and include those in the *WriteData* expression for your error table, to give you even more information.

[THIS PAGE INTENTIONALLY LEFT BLANK]

5

Using External Data Sources

Incorporating external data sources into your application will allow you to use much larger amounts of data.

There are some key points you should consider when using data sources:

- Before you get started you should map out the types of data you will be using and the relationships between them. Once you start referring to data sources it can be quite difficult to make changes, particularly if numerous models refer to the same data sources. This is especially important if you cannot directly edit the data you will be using for your data source.
- If multiple tables use the same data, such as a User ID, avoid repeating these fields as much as you can. Use foreign keys and table joins to link tables by mutual data, rather than repeat it every time.
- Check that aliases have been set to the required fields in your data source. You need to create aliases for any fields that you will be passing values into or retrieving values from.
- When using *WriteData* expressions you should make good use of the *WithDBTrans* function to ensure that the failure of one transaction in a series of interdependent ones does not cause problems for the rest. Refer to **"Using WithDBTrans"** on page 51.
- You should use a *FormLoad* expression or a Data Source Connect Expression attribute to define exactly when to connect to a data source. Similarly for data grids, filters and multiple choice questions, use the Connect Condition to do the same thing. This will let you define exactly when the data source is accessed and reduce unnecessary traffic. Always bear in mind that filters will not work until you have connected to a data source. You cannot use a filter condition or a Refresh Data Source Expression to prompt a connection to the data source, only to refresh the data. As such, you should only attempt to connect to a data source when necessary and be precise in any expressions you use to connect.
- When using a data grid, only select the fields in the data source that the grid will use. Do not include every field and simply remove the unnecessary ones when configuring the grid. This will reduce traffic from the data source and return the data you need faster.
- Try to reuse data sources as often as you are able to, instead of creating multiple data sources that retrieve the same data. This will reduce traffic to and from the data source, and allow data to be returned quicker.
- If you remove a field from a table that is referred to in filters or expressions etc., do not use the Refresh This Data Source When option as it will then break those filters and potentially cause expressions to fail. First correct the filters and expressions and then refresh fields. If you use a variable to store column aliases then you will only need to update the variable value. This technique is detailed in **"Using Identifiers"** on page 15.

In situations where you refer to the same data source repeatedly you may encounter issues where values from prior iterations are retained. For example, if you have executed a *GetDataSourceRow* expression then the data source aliases will receive and store a value. If you then execute a *WriteData* expression the application will pass new values to the aliases. If some of the aliases do not receive new data then the old values will be written along with the new ones. This is especially problematic if multiple users are using the same expressions or entering responses into the same aliases. One way of avoiding this is to use an action to pass neutral values into the aliases prior to adding new ones.

In these cases it is best to use a naming convention to distinguish between your data sources and their purposes. For example, if you have a data source that is used purely to display values, add a term such as 'Loading', 'Lookup' or 'Read

Only' to its name. For data sources that will write data, use 'Write' or 'WriteData' in the name. Aliases should be similarly named, especially aliases for write data sources. This allows you to generate distinct aliases for the same database and refer to it only when and how you need to.

You must also bear in mind how often the data source is refreshed and avoid refreshing it unnecessarily. Aim to use already-returned data as much as possible before querying the data source again. For example, if you have a series of multiple choice questions, each returning their responses from the data source, you should only refresh once all questions have been answered. You do not need to refresh after each question. You should also bear this in mind when setting refresh conditions on filters applied to those questions.

Connect Conditions

Certain KnowledgeKube elements, such as data source grids and multiple choice questions, have an option to determine when they connect to their data source. Ordinarily, they will connect by default and remain connected - i.e. if the 'Connect to data source when condition is met' option is not selected. Whilst it may be convenient to leave them permanently connected and essentially 'forget' about them, it is best to set a connection condition so that your application only calls the data source when required. This reduces traffic to and from your data source, and reduces the amount of unnecessary data stored in KnowledgeKube. Similarly, you should disable an element from connecting if it isn't being used.

You must also be considerate when multiple elements refer to the same data source, as they will not work until a dependant connection is made. For example, if a filter on a data grid relies on the response from a multiple choice question, which also uses the same data source, then the filter will not work until the multiple choice question has connected to the data source. As a further result, the grid will not display anything, regardless of its own connect condition.

There are also security issues to consider. You don't want the application to return any information that a particular user isn't permitted to access, whether or not the information is displayed to them. While you can filter what a user sees, data fetched from a data source will exist in the browser's session memory, from where it could potentially be accessed by other means. For example, if your application fetches financial data from a source containing details of several users, you should ensure the data retrieved from the database is filtered to exclude anything the user should not see, such as the details of other users in the database. If the data source returned all user information and only filtered it on the front-end then you are potentially breaching confidentiality and also relying on the filter to work without error. If a malicious user was able to exploit the application to view other information in the session, this would be catastrophic.

When you disconnect an element from a data source, all of the information is lost and anything referring to it will no longer have access to the information. It is not kept in memory.

Using T-SQL Data Sources

Before you start using a T-SQL data source, create a Database Diagram in SQL of all of the required tables and their relationships to each other. Save this diagram to your computer so that you can make notes and refer to it in the future, especially if you are working on a group project. Maintain a good data structure within SQL; disorganised or confusing data will make implementing T-SQL much more difficult, so use a consistent naming convention for your data.

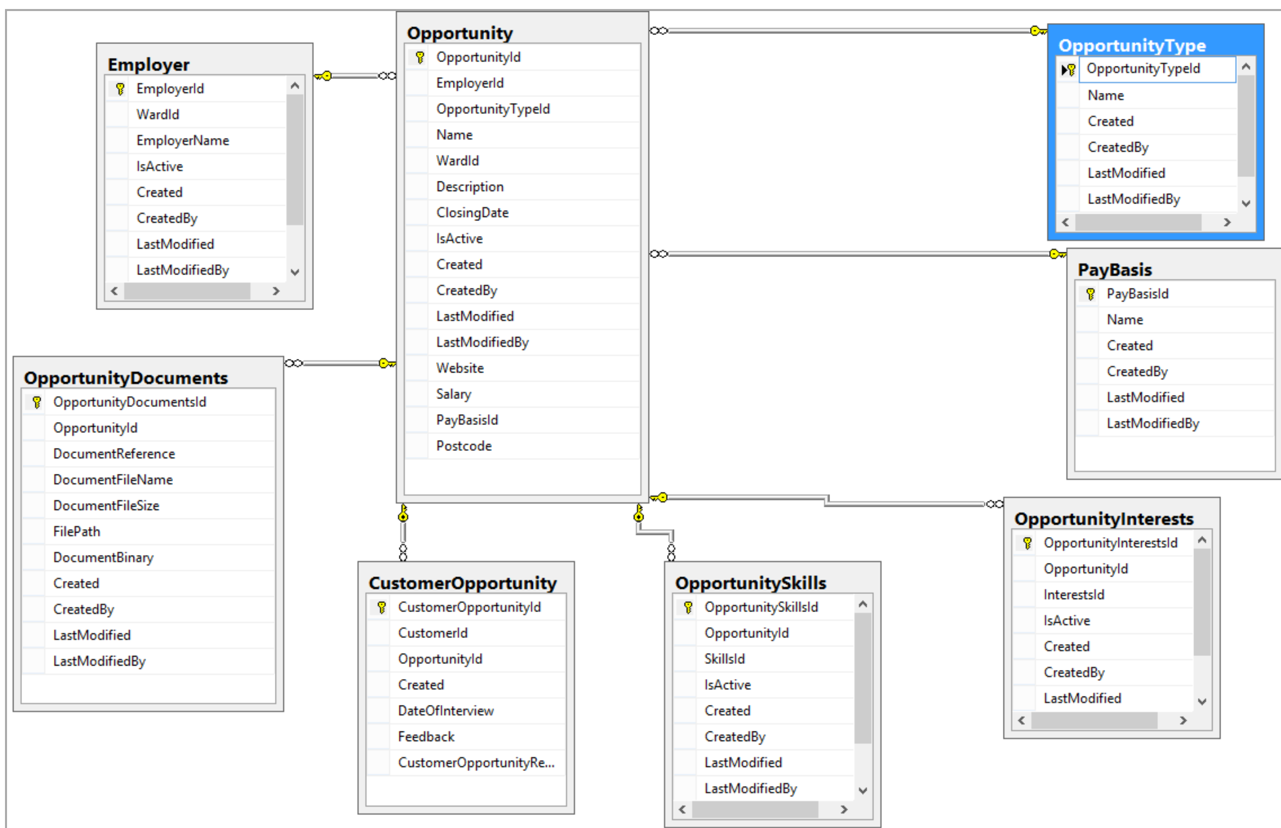

Figure 5-1: Table diagram created in SQL 2012 showing tables and their joins.

It is recommended that you give any fields that are used to join more than one table together the same name. This lets you easily identify how and where the tables are connected.

When adding the data source, set the 'Maximum Rows' to 0 to return unlimited rows and prevent issues reading the data and constructing the new table. KnowledgeKube applies a default of 200 rows and you may find this inadequate.

When writing your joins in the Query panel, use a consistent method so as to avoid confusion. For example:

```
JOIN OpportunityType ON [Opportunity].[OpportunityTypeID] = [OpportunityType].
[OpportunityTypeID]
```

Updating either the **Fields** of the data source or the *Select* statements in the query editor will not automatically update the other. You must be sure to keep both sets of information consistent, or the data source will stop working as it should.

When you have designed your table in the editor, copy everything in the Query window into a Notepad document for easy reference and editing, before clearing the window to save memory. It will be much easier to edit rows in Notepad and then copy the data back into the editor rather than rewrite them all.

Creating an SQL Database

Creating and managing an external SQL database for use with KnowledgeKube requires a good understanding of SQL. As such, you will need someone with knowledge of SQL to create the database(s) required for your model. If you will not be involved in this part of the project then this section will not apply to you.

Before you begin, there are some key points to bear in mind that will help you to avoid problems with your model's SQL data source:

- Set an auto-generating single integer primary key in an appropriate column.
- Always base your index off a primary clustered key and never use a non-clustered index on its own.
- Ensure that columns are set to the right data types for values such as dates, currency values and so on.
- Indices can help partition data but generally only if there are more than ten records. This can help make filters work faster as well.
- Be aware that KnowledgeKube contains a list of reserved keywords and if a column is named the same as one of these keywords it may lead to errors. This should be easy to avoid if you are using a naming convention, but be aware of it in case you encounter issues.

As an example, the **Saved Form Registry** uses reserved keywords to pass data to column headers in **CDS**:

- FormNumber
- FormReference
- FormStatusID
- FormStatusText
- KnowledgeGroupText
- ProductText
- CreatedDate
- LastUpdated
- InceptionDate
- ExpiryDate
- UserName
- UserEmail
- UserForename
- UserSurname

For configuration data it is recommended you store information in a normalised form unless it is transaction data. For example:

- If you have an ordering system, store the information as an Order table, an Order Line table, a Products table and so on, with references linked by a common detail such as ID.
- Avoid storing all of your data in one large table as doing so is less secure (more capacity for errors to damage all of your data at once), slower and more difficult for SQL to process.

- Use clear, simple and consistent names for your tables. This will help avoid mistakes and allow new people working on your project to adapt quickly.
- Maintain referential integrity to allow other developers to see the data relationships immediately.

Be sure to have as much information about the data you need to include in your tables as possible so that you can avoid errors and rework. Use synonyms or views on the SQL server if the table is likely to move or change.

Date/Time Format Conventions

When writing date and time information to or from a SQL Server database you should only ever use the recommended string format. This format varies slightly depending on the function in which it is used. If you are creating a normal date/time function, such as **GetDateTimeFormat**, the string should be written as follows:

```
yyyy-MM-ddTHH':'mm':'ss
```

If you use it in a data function, such as **WriteData**, the string should be written like this:

```
{0:yyyy-MM-ddTHH':'mm':'ss}
```

You would usually format the date twice in order to get the correct output in KnowledgeKube, once using the GetDateTimeFormat function and then again with the WriteData function.

When using the WriteData function it is important that you always specify all five date and time parameters, even the ones that are optional. Using any other string or missing any of the parameters will cause the date/time format to be ignored.

An example of this format being used in a GetDateTimeFormat expression might look as follows:

```
GetDateTimeFormat("yyyy-MM-ddTHH':'mm':'ss","en-GB");
```

In this case, the expression would return the following value:

```
2015-05-22T12:33:45
```

Data Sources and Variables

When working with data sources it is recommended that you minimise the amount of times you query them. Doing so will reduce the need for the application to make constant connections to an external source, which reduces the strain on the database while also making the expressions easier to view. One way this can be done is by saving the value from a data source in a variable and using that in any further expressions.

For example, the following example uses two data source connections, both of which are queried twice:

```
if:(GetDataSourceRow("DataSource","FilterOne","True", 1) =
GetDataSourceRow("DataSource","FilterTwo","True", 1))
{
ResultVariable:= "A";
Stop;
}

if:(GetDataSourceRow("DataSource","FilterOne","True", 1) <>
GetDataSourceRow("DataSource","FilterTwo","True", 1))
{
ResultVariable:= "B";
Stop;
}
```

A more efficient and server-friendly way of achieving the same end results would be to first use a *GetDataSourceRow* expression and pass the value into a variable instead:

```
DataSourceVarOne:=GetDataSourceRow("DataSource","FilterOne","True", 1);
DataSourceVarTwo:=GetDataSourceRow("DataSource","FilterTwo","True", 1);
```

At this point the compound expressions can be rewritten using these variables as qualifiers, which will make it easier to view and require less external connections:

```
if:(DataSourceVarOne = DataSourceVarTwo)

{
ResultVariable:= "A";
Stop;
};

if:(DataSourceVarOne <> DataSourceVarTwo)

{
ResultVariable:= "B";
Stop;
};
```

Using Filters

Each filter is associated with a specific data source, and as such they cannot be copied to or used with others. Only one filter can be used when filters are referenced in expressions or applied to questions that use data sources.

There are important differences in where you use filters, either directly on a data source or with an element that uses that data source. Refer to **"Primary and Secondary Level Filtering"** on the facing page. Filters can be prompted to make a new call to the data source whenever the conditions by which they are filtering change. This should only be done when

necessary, to prevent unnecessary network traffic, amongst other reasons. Refer to **"Refreshing Filters"** below for more information.

Filters are configured using the same logic as expressions, so ensure that you construct each argument correctly. Errors caused by conflicting or unnecessary filter commands may not be easy to detect. You should also be careful of false-positives where the correct information is technically returned but not in a specific enough way to prevent it being returned in other circumstances. For example, avoid using *Like/Not Like* conditions when you can use *Equals/Not Equals* instead, and never use a '%' at the beginning of a *Like* condition. This is especially important for T-SQL data sources, where you should only ever use a '%' at the end of a query.

You can also use the 'Order By' condition on a filter to further affect how the results are returned. This is especially relevant in cases where data can be written to the data source in any order, such as many different users entering personal data, but then needs to be viewed alphabetically later in a data grid. You can also use a field that is not necessarily displayed for ordering, such as Row ID or account number, depending on requirement.

Primary and Secondary Level Filtering

Adding a filter directly to a data source is referred to as **Primary Level Filtering**. This means it will only return data from the provider matching the filter criteria. This is useful as it allows you to query a potentially huge data source but only return what you need. If your primary level filter relies on values from the application, those values will need to have been generated before the data source is called otherwise the filter will fail and the data will not be called.

Applying a filter to data returned from a filtered data source is referred to as **Secondary Level Filtering**. Refining data in this manner allows for very precise information to be returned and allows your application to run more efficiently. If you were to call absolutely everything and then filter it later, you are still holding all of that redundant data in session memory and including it in any refresh or write operations. This will slow your application down and add major traffic to your network. If you refresh a secondary level filter, this will make a new call to the database as per its own conditions, not the conditions of the primary filter, which can cause inappropriate data to be returned. For more information about refreshing data sources, refer to **"Refreshing Filters"** below.

Refreshing Filters

A **Primary Level** filter cannot refresh the data source, even if its conditions are met. It is used purely to dictate what data is initially called from the database. A **Secondary Level** filter, or a filter used in an expression will trigger a new call, or refresh the data. In these cases it is important that the conditions of the secondary filters are compatible with the ones on the primary, so that it is still returning information within the context of the original filter. Otherwise, you may return information you don't need. Filter levels are covered in more detail in**"Primary and Secondary Level Filtering"** above.

Filters can be set to refresh using one of the following triggers:

- **The Filter Condition Changes** - This is when a application factor, such as a variable value, changes. Please note that only changes made within the application can prompt a refresh. For example, if the values in the database are overwritten directly using a *WriteData* expression, the information in a data grid will not update - i.e. the filter will not refresh - because no change was made to anything in the application. In these cases you will need to implement another way of prompting the filter to refresh, an example of which is described below.
- **The Following Condition is Met** - The filter will make a new call to the data source when an expression resolves as true. You can also use a variable name in this field, if the value of the variable will be either 1 or 0. You must be especially careful using *If* statements as the 'fail' outcome will prevent the filter from working at all and cause an error. Ensure that you compose your expression accurately, because if the filter is unable to refresh or initiate, if calling the data source for the first time, then this will cause an error and you may not be able to continue. In these cases you will need to generate the values the filter relies on before it is used or set a default value via an expression. A simple use of this condition would be to use a button to pass a value to a variable that the filter will then respond to, effectively creating a manual refresh button.

The 'Connect to...' condition on an element such as a data grid will supersede any refresh conditions on a filter, meaning that a filter will never be able to make a call to the data source if the data grid is not connected. Refer to **"Connect Conditions"** on page 44 for more information.

It is essential for the filter to refresh the data only when required, such as when new data is written to the data source. You must also make sure that multiple filters on the same form do not have similar or identical refresh conditions, as each one will trigger their own refresh. Leaving a filter in a permanent 'on' state will cause the filter to refresh and call the data source every time a post-back of any kind is made. While this may not have much impact on a single user, the traffic to and from the data source will only be compounded as more users interact with it.

This is critically important in expressions that refer to a filter, such as *ForEachDataSourceRow*, because the filter is referenced every time the expression executes. For example, if you were to query a database containing one thousand rows using a *ForEachDataSourceRow* expression with a filter with a refresh condition of '1', the filter will trigger a refresh every single time the expression resolves as true. Even if this only applies to 50% of the rows in the database, this still means that the data source has been refreshed *five hundred* times. If another user were to execute the same expression at the same time then this effect is doubled and so on. This can very quickly become a significant issue.

Please note that if multiple elements refer to the same data source then prompting a refresh on one of them will update the information for all of them. For example, a form has two data grids each using the same data source but they only have a single field in common. The first data grid uses that field as a condition in a filter, whilst the second shows otherwise different information with its own separate filter. The filters on both grids are set to automatically refresh when a certain condition is met, though the condition is different for each. New data is written into the first grid and a condition is included to prompt the filter to refresh and show the new records. However, when the filter in the first grid refreshes, the information in the second grid is also updated regardless of whether its own filter refresh condition is satisfied or not. Furthermore, the second grid will appear to have new information added to it seemingly without any input. If multiple users are accessing a site at once this could become very confusing and quite visually distracting.

Attention must also be paid to question ordering within a form. If the condition a filter relies on is not available at the point the filter is used, it will not work correctly. Questions that are used to provide values to a filter must be appear in the form earlier than the filter. For example, if you are using a multiple choice question to filter a data grid, this question must be placed before the grid in the form. If it appears after, the filter will have no data when the form loads and the grid will show no data. It will only update once the multiple choice question is interacted with and the filter then refreshes the grid.

Secondary filters cannot be used to refresh multiple choice questions. Similar to primary level filters, they can only limit the available data. They will not refresh the available options in the question even if the field the question uses is the one that is updated.

CSV Data Sources

This is a unique type of data source that permits KnowledgeKube to treat CSV data as though it were a tabular database, which can be filtered and used to populate data grids like other types of data sources. This CSV data is stored inside a standard KnowledgeKube variable.

The contents of the variable can be generated from a database table, as well as used to pass data back to a table. Using CSV in this way not only reduces traffic to the data source, it also reduces the possibility of making mistakes with the data on the data source, as you can write the data when you are satisfied.

Before you use a CSV data source, consider that they should primarily be used to contain small amounts of data locally for the duration of a session. If a CSV contains more than 250 rows it will cause severe slow-downs, as each row will be checked every time a postback occurs, which is very processor-intensive.

For this reason you should also avoid using *ForEachDataSourceRow* and *ForEachCSVListItem* functions, as they also constantly re-check the CSV data. Instead, use the dedicated **CSV List Functions** to analyse your data much more precisely. These are fully explained in the **Advanced Expression Cookbook**.

Also note that semi-colons (;) cannot ordinarily be used in a CSV list as the expression engine uses these to denote the end of a line or process. However, you can use a constant to place a semi-colon into a list, if applicable, using the process described in **"Writing Expressions"** on page 28.

Using WithDBTrans

The *WithDBTrans* function is used to implement an **Explicit Transaction** in KnowledgeKube, which is a series of expressions executed together in a single transaction.

When the expressions are executed by the function it performs three key actions:

- It executes all expressions as a single un-interruptible transaction preventing any further transactions from affecting the data source until the first one has completed.
- It only alters the data source if all of the expressions execute successfully. If an expression fails then the data source will be reverted to its last stable state and no changes will be made.
- It will automatically close the transaction once it is completed, preventing data loss and conflicts.

Because multiple expressions can be contained within the function this allows you to make a single call to the data source whilst executing a series of operations. This will reduce network traffic and processor load.

This function should be used when executing time-sensitive operations or operations that are dependant on each other to complete. Otherwise, if one stage of an operation fails, subsequent operations may also fail or build on corrupt data. Other users referring to the same data source may also interrupt or affect your transactions, particularly if you are querying the exact same data. The *WithDBTrans* function will protect these operations by ensuring they have executed fully before allowing any further transactions.

[THIS PAGE INTENTIONALLY LEFT BLANK]