# GeoFront Engineering Case Study

A company called GeoFront Engineering want to create an expenses system to allow their employees to quickly and easily register claims for costs incurred whilst travelling between sites. By combining a KnowledgeKube model with an SQL-compatible data source they can implement a system that will record a user's claim details, which an approver can then accept or decline.

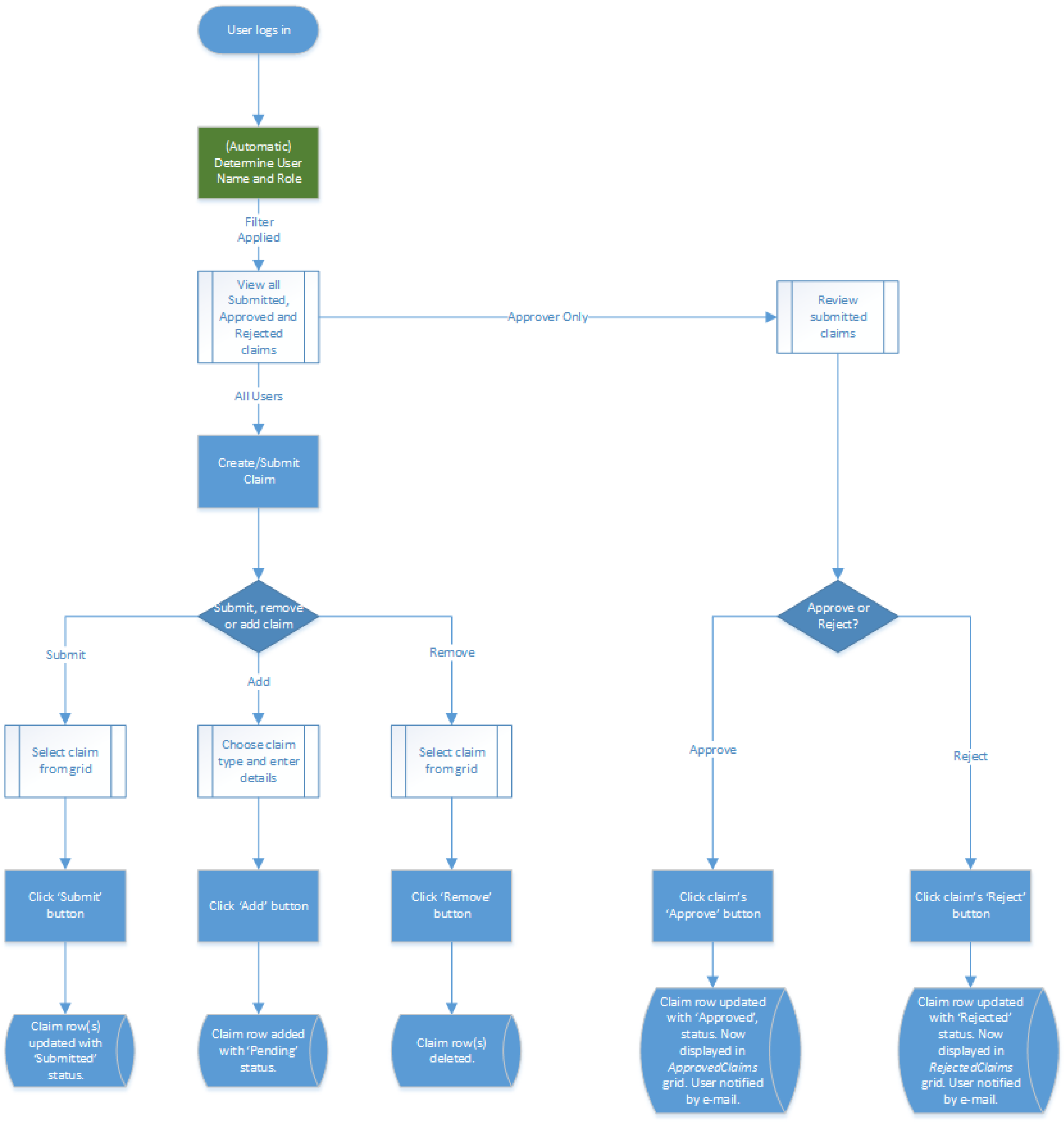

User logs in
(Automatic) Determine User Name and Role
Filter Applied
View all Submitted, Approved and Rejected claims
Approver Only
Review submitted claims
All Users
Create/Submit Claim
Submit, remove or add claim
Submit
Add
Remove
Select claim from grid
Choose claim type and enter details
Select claim from grid
Click 'Submit' button
Click 'Add' button
Click 'Remove' button
Claim row(s) updated with 'Submitted' status.
Claim row added with 'Pending' status.
Claim row(s) deleted.
Approve or Reject?
Approve
Reject
Click claim's 'Approve' button
Click claim's 'Reject' button
Claim row updated with 'Approved', status. Now displayed in ApprovedClaims grid. User notified by e-mail.
Claim row updated with 'Rejected' status. Now displayed in RejectedClaims grid. User notified by e-mail.

*Figure 1-1: Flow diagram for normal and approver users.*

# Create Your Model

You have been asked to create the expenses system for GeoFront Engineering. It will need to run for a full tax year, from April 1st of the current year to April 1st of the following year.

Create a new model to contain the expenses system and give it a suitable name, reference and time frame. This particular questionnaire is for the North American office so assign the correct region and territory to the model as well.

Create a published group name called 'Human Resources' and assign your model to it.

## Add Initial Question Groups

To begin with, add the following question groups of the type **Form**:

- *PersonalInformation* - This group will be used to display the personal information and previous claims for the logged-in user.
- *RegisterClaim* - The system will allow employees to record two different types of expenses claim: travel and accommodation. This group will host the two key question sets for creating a claim as well as listing any pending claims the user may want to submit.
- *TravelClaim* - This will contain the questions relevant to travel expenses and will be contained in the *RegisterClaim* group with a placeholder.
- *AccommodationClaim* - This will contain the questions relevant to accommodation expenses and will be contained in the *RegisterClaim* group with a placeholder.
- *ApproverGrids* - This will contain a series of data grids that will present all claims submitted by all users, which the approver to approve or reject, and display them accordingly.

Add navigation buttons to the *PersonalInformation*, *RegisterClaim* and *ApproverGrids* question groups.

## Add Questions and Responses

Next, you need to add some basic questions to your question groups. In the *PersonalInformation* group add **Read-Only Text** questions to display the user's details. This information will be retrieved from a data source so you only need to add the questions at this point. Leave the question text blank for the moment as they will be mapped to variables at a later point.

In the *RegisterClaim* group add a **Date** question and a **Multiple Choice** question to list the claim types with the following responses:

- --Please Select One--
- Travel
- Accommodation

Set the first response as the default and ensure that the question will not accept this as an answer.

You also need to add **Placeholder** questions for the *TravelClaim* and *AccommodationClaim* question groups.

Next, add the following questions to the *TravelClaim* question group:

| Type | Question Text |
|---|---|
| Yes/No | Public Transport? |
| Currency | Ticket Cost (£) |
| Number | Distance (Miles) |
| Currency | Fuel Cost (£) |
| Yes/No | Company Car? |

After that, add the following questions to the *AccommodationClaim* question group:

| Type | Question Text |
|---|---|
| Number | Duration (Days) |
| Currency | Food Cost (£) |
| Currency | Phone/Internet Costs (£) |

Add all of the questions from both tables to the expression parser as you will need to call upon their values later on.

## Add Expressions and Attributes

Add a *ShowGroupForm* expression to your navigation buttons to allow the user to move from the *PersonalInformation* group to the *RegisterClaim* or *ApproverGrids* groups and back again. Since the *ApproverGrids* form should only be accessible to users with the correct role, a visibility expression will be added to the relevant navigation button later on.

You should also add *CauseButtonValidation* attributes to any 'Back' buttons to allow the user to return to a prior group without needing to answer any mandatory questions.

You now need to add visibility attributes to the following questions:

| Question | Outcome |
|---|---|
| TravelPlaceholder | This should only display its group if 'Travel' is selected as the claim type. |

| | |
|---|---|
| AccommodationPlaceholder | This should only display its group if 'Accommodation' is selected as the claim type. |
| Ticket Cost (£) | This should only appear if the *Public Transport?* question is answered 'Yes'. |
| Distance (Miles) | This should only appear if the *Public Transport?* question is answered 'No'. |
| Fuel Cost (£) | This should only appear if the *Public Transport?* question is answered 'No'. |
| Company Car? | This should only appear if the *Public Transport?* question is answered 'No'. |

Finally, add placeholder attributes to the placeholder questions in the *RegisterClaim* question group to link them to *TravelClaim* and *AccommodationClaim* question groups respectively.

Create the following variables to collect the user details from the data source:

- *GetUserName*
- *GetLineManager*
- *GetOffice*
- *GetEmailAddress*

These variables will not receive their values until the data source is connected. For now map these variable names to the respective read-only text questions in the *PersonalInformation* question group so that their values will be displayed on the page.

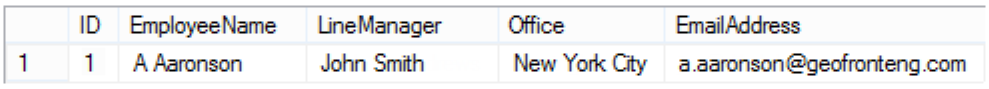
## Create Users and Roles

Create a series of standard users to represent the site employees. Ensure they all have the role *SiteEmployee* and are added to the *SiteEmployees* workgroup. Create two users with the *Approver* role to represent human resources employees and add them to the *HumanResourcesEmployees* workgroup. Make a note of all of the user details as they will be needed again when you make your table in the data source.

Assign a role-based visibility expression to the navigation button that directs the user to the to the *ApproverGrids* question group that will prevent anyone who is not an approver from seeing it.

## Using Data Sources

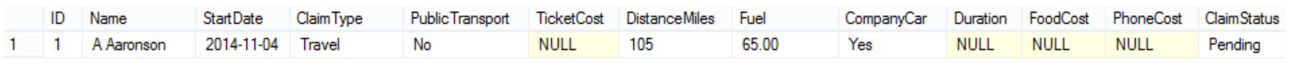
At this point you have built a basic model with a few questions. You will now use data sources to provide it with its full functionality. You need to create two tables in an SQL database. The first, *UserDetails*, will contain columns for the user's name, their line manager, the location of their office and their email address. These columns will hold the information that will be called and displayed in the read-only questions in the *PersonalInformation* question group.



| | ID | EmployeeName | LineManager | Office | EmailAddress |
|---|---|---|---|---|---|
| 1 | 1 | A Aaronson | John Smith | New York City | a.aaronson@geofronteng.com |

*Figure 1-2: Example of an employee in the UserDetails table.*

The second table, *ClaimResults*, will contain rows representing individual claim submissions, created using the user's responses to the expense detail questions. Name each column after the keyword of the corresponding question making sure they are exactly the same. Further to this, add a column for the user's name and another column called "ClaimStatus".



| | ID | Name | StartDate | ClaimType | PublicTransport | TicketCost | DistanceMiles | Fuel | CompanyCar | Duration | FoodCost | PhoneCost | ClaimStatus |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | A Aaronson | 2014-11-04 | Travel | No | NULL | 105 | 65.00 | Yes | NULL | NULL | NULL | Pending |

*Figure 1-3: Example of a pending travel claim in the ClaimResults table.*

Now these tables have been created you can integrate them with your model. Add a new Data Source called *UserDetailsDataSource*, connect it to your SQL database and link it to the *UserDetails* table. Once this is done, create another connection called *UserClaimsDataSource* and link it to the *ClaimResults* table. Finally, add another connection called *ApproverClaimsDataSource* and link it to the *ClaimResults* table as well.

For *UserDetailsDataSource* and *UserClaimsDataSource* add all fields from each table. The *ApproverClaimsDataSource* will be used to list approved and rejected claims. As the claims have been reviewed at this point they only need basic details for reference, so only add the *ID*, *Name*, *StartDate*, *Claim Type* and *Status* columns for this connection.

Finally, return to the *UserClaimsDataSource* and create aliases for all of its columns. Because the column names match the question keywords, the aliases need to as well. You could use your own values for the column aliases but ensure that you change the relevant question keywords and variables to match. Add aliases to all columns in the *UserDetailsDataSource* except the *ID* column.

## *Display Values From a Data Source*

There is no direct way to connect a data source to a read-only question. To display information from a data source as read-only values in KnowledgeKube you must query the relevant data source, retrieve the appropriate values and assign them to variables for use by the rest of the model.

Add a *FormLoad* expression question at the very beginning of the *PersonalInformation* question group to execute a *GetAuthenticatedUser* expression and return its value to the variable *UserNameVar*. This name will be used to query the *UserDetails* table for the rest of the user's information.

Next, create an action called *GetUserDetailsAction* to assign the value of each column alias from the *UserDetailsDataSource* to the respective variable you created for the read-only questions in the *PersonalInformation* question group. An example of the expression in the *GetUserDetailsAction* is as follows:

```
GetUserName:=EmployeeNameAlias;

GetLineManager:=LineManagerAlias;

GetOffice:=OfficeAlias;

GetEmailAddress:=EmailAddressAlias;
```

Finally, add another *FormLoad* expression to execute a *ForEachDataSourceRow* expression on the *UserDetailsDataSource* that uses the *GetUserDetailsAction* and a filter that only returns results matching the name stored in the *UserNameVar* variable.

As a result of these expressions, when a user logs in their details will be automatically retrieved from the user data table and used to populate the relevant detail fields.

## *Using The WriteData Expression*

Now that the data sources are connected to your model, you need to write data back to them and display the results in a meaningful way. First, you need to add a button to the *RegisterClaim* question group that allows users to add a new claim. Add a *WriteData* expression to write back all of the question responses into the *UserClaimsDataSource*. The *WriteData* expression must include the alias of every column in the table with the exception of the *ID* column.

As there is no question that generates a value for the *ClaimStatus* alias, create a variable with this name and include a line in the button's expression that passes the string "Pending" into the *ClaimStatus* variable before the *WriteData* expression is executed. A 'Pending' claim is not reviewed by an approver so a user can create a claim and progress it when they are satisfied.

## *Using Data Grids and Filters*

After a user has logged in they should be able to see their previously submitted, approved and rejected claims. Add a **Data Source Grid** to the *PersonalInformation* question group called *AllUserClaimsGrid* and connect it to the *UserClaimsDataSource*. When you create the grid definition, omit the *ID* and *Username* columns as they are not required. Create a filter that will only show claims with a status of "Submitted", "Approved" or "Rejected" for that logged in user and ensure it automatically refreshes every time it is used.

Next, you will use a grid to present pending claims to the user, allowing them to review the claims prior to submitting them for approval. Create a grid in the *RegisterClaim* question group and call it *PendingClaimsGrid*, positioning it underneath the claim detail fields. Connect it to the *UserClaimsDataSource* and add all columns to the grid definition. Set the *ID* column to be hidden and omit the *Name* column, then add a filter to the grid so that it filters the results by the user's name and automatically refreshes every time it is used. This allows a user to review all of their pending claims, and will update in real-time when they add new ones.

At this point you have made everything a standard, non-approver user needs to see. Approvers need to be able to review submitted claims from all users and either approve or reject them on an individual basis. Add the following data grids to the *ApproverGrids* question group:

- *SubmittedGrid*
- *ApprovedGrid*
- *RejectedGrid*

The *SubmittedGrid* needs to be connected to the *UserClaimsDataSource*. Create the grid definition and set the *ID* column to be hidden and omit the *ClaimStatus* column. Approvers should be able to authorise and reject all submitted claims except ones they have submitted themselves. Add a filter that will only return an appropriate list of claims by testing the name of the claimant and the status of each claim.

Next, add two **Action Link** columns to contain an 'Approve' and 'Reject' button respectively. To each you will need to add an action that will update the *ClaimStatus* column entry for that row to "Approved" or "Rejected". Add a *SendEmail* expression to these buttons that will inform the user that their claim has either been approved or rejected.

The *ApprovedGrid* and *RejectedGrid* need to be connected to *ApproverClaimsDataSource* and be given filters to show rows that have been approved or rejected, respectively. This means that whenever an approver either approves or rejects a submitted claim it will automatically appear in the appropriate grid.

## *Using The String List Functions*

At this point a user is able to review their previous claims and add new claims by writing them to the data table with a status of 'Pending'. In order to progress, the user will have to formally submit their claim to an approver for it to be approved or rejected. To do that you need to allow a user to individually select which claims they want to progress and then update the status of those claims to "Submitted" so that an approver can view them.

Add a **Check Box** column, position it in the far right of the grid and give it the keyword *CheckBoxColumn* so the user can select which claim(s) to submit. To prevent different selections from overwriting each other, and to allow multiple selections, you will need to implement the *StringListItemExists*, *StringListAddItem* and *StringListRemoveItem* expressions. Together these expressions will pass the row ID to a variable and check if it is already present in the string list - if it isn't, the ID should be added to the list; if it is, the ID should be removed.

Create a variable called *CheckBoxVar* then create an action called *CheckBoxAction* and write an expression that will use all three list function expressions to pass the row ID to the *CheckBoxVar*. The *Item* argument will need to be the keyword of the *ID* column in the data grid. An example of the complete *CheckBoxAction* expression is as follows:

```
CheckBoxVar:=

if(StringListItemExists(CheckBoxVar, ",", IDColumn),

StringListRemoveItem(CheckBoxVar, ",", IDColumn),

StringListAddItem(CheckBoxVar, ",", IDColumn));
```

After you have done this, assign the action to the check box column in the *PendingClaimsGrid*.

Finally, add an *IsSelectedExpression* **Grid Expression** that will check whether the row ID is present in the *CheckBoxVar* string list. If it is then the tick box will stay selected and empty if not.

Next, add a button below the grid with a *ForEachDataSourceRow* expression to update the value in the *ClaimStatus* column of each claim that has been ticked by the user. This requires an action that first updates the value of the *ClaimStatus* variable to "Submitted", then uses a *WriteData* function to update rows matching the filter criteria with the new status. Any rows which do not match the filter - in other words, any that have not had their check boxes ticked - will stay pending for the time being.

You should also add a 'Remove' button that will execute a *DeleteData* expression to remove any selected rows from the table, allowing a user to delete claims they don't want to submit.

| | |
|---|---|
| Actions | ✓ |
| Attributes | ✓ |
| Buttons | ✓ |
| CDS | |
| Data Processing | |
| Data Source Grids | ✓ |
| Data Sources | ✓ |
| Documents | |
| E-Mail Templates | |
| Expressions | ✓ |
| Language Translations | |
| Model Scheduler | |
| Model Variables | ✓ |
| Output Designer | |
| Rating Definitions | |
| Repeating Question Groups | |
| States | |
| Summary Page Designer | |
| Users/Roles/Workgroups | ✓ |
| Validators | |

## Abstract

The GeoFront Engineering case study was created to demonstrate how certain features in KnowledgeKube could be implemented in a true-to-life example.

By following this study, the user will be shown how to construct a simple expenses claim system for employees of the fictional GeoFront Engineering company. It will allow registered users to create, manage and submit claims that an approver will then accept or decline.

The key functionality of this model involves the use of data sources, and demonstrates examples of writing new data rows, updating existing rows, and fetching data as required. Data from the database is then presented in an appropriate format throughout the various sections of the application.

This case study describes the implementation of core functionality only; the flexibility of the KnowledgeKube application allows users to expand models to include bespoke features according to individual client requirements.

## Prerequisites

Before you begin this case study you will need to be familiar with **List Functions** as well as the **Data Source Functions** from the KnowledgeKube guide. Knowledge of SQL and the ability to create/edit a database will be advantageous, however if you are not able to create a simple, SQL-compatible database with several tables yourself you will need someone who can set this up for you.