**Exercise 11 – INOUT to fill-in templates**

Businesses generate a lot of documents, most will require data to be included from some source, manually typed, cut/paste or from a database. Often these documents are set up as templates and where data needs to be added, they will include a **tag** to show where the data goes and what data should go there, i.e. <Insert Name Here> or <PhoneNumber>, etc. usually the tags are enclosed in the < and > characters, but they can be anything.

In this exercise, imagine if you will that the Automobile Club has a reporting requirement to some governing body that wants to know what type of identification documents were used by the members and how many of each, and this report needs to be lodged on a regular basis, say every three months. The best approach would be to create a template of the report with tags appropriately named and positioned within the template. Then every three months, a script can be run that will pass the template into a stored procedure which fills in the details from the database. The template will ensure the document is standard and correct and the stored procedure will ensure the data is correct, are you excited? I am, this is really cool stuff, let us do it.

As in Exercise 10, we will simply zip through the creation and running of this and then explain it at the end.

1. First, you need to run the following **CreateTemplateDatabase.sql** file which can be found in Github Chapter4/Exercise11. This file will create another database, complete with a template we need for this exercise.

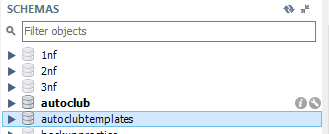


Figure 4.34 New autoclubtemplates database

1. To create the Stored Procedure, open a new SQL Tab and enter the following Stored Procedure definition, there are lots of comments so be sure you get the comments correct.

USE `autoclub`;

DROP procedure IF EXISTS `sp\_CountIdentificationTypes`;

DELIMITER $$

USE `autoclub`$$

CREATE DEFINER=`Tom`@`%` PROCEDURE `sp\_CountIdentificationTypes`( INOUT TemplateText LongText)

BEGIN

-- Create the SELECT statement, the data will be coming from a

-- Sub Query and the returned rows will be in a single field named TheText

-- GROUP\_CONCAT will combine the values from the text

-- in all rows returned into one string and placed the string into @IDCount

SELECT GROUP\_CONCAT( DISTINCT TheText ) INTO @IDCount

-- The FROM clause, next comes our data source

FROM

-- The data source is a Sub Query. The sq after the closing bracket

-- of the subquery is just a name we need to assign it.

-- Sub Query CONCATs the field values into a derived field named TheText.

-- The /n is a NewLine to ensure each row will start on a new line

-- The count is surrounded by brackets

(

SELECT (CONCAT('\n',identificationtype.identificationtype, ' (', Count(identificationtype.ID),')' ) ) as TheText

-- The source of the data

FROM

identification

INNER JOIN identificationtype ON identification.IDType = identificationtype.ID

-- GROUP BY ensures each identificationtype found is grouped together

GROUP BY

identificationtype.identificationtype

) sq;

SELECT

Count(members.Active) INTO @ActiveMemberCount

FROM

members

WHERE

members.Active <> 0;

-- The REPLACE command will replace the <IdentificationData> tag in the TemplateText passed in

-- with the values now in @IDCount

-- The TemplateText is now modified and ready to be passed back

SET TemplateText = REPLACE(TemplateText,'<ReportDate>', CURDATE());

SET TemplateText = REPLACE(TemplateText,'<IdentificationData>', @IDCount);

SET TemplateText = REPLACE(TemplateText,'<TotalMembers>', CONVERT(@ActiveMemberCount,UNSIGNED));

END$$

DELIMITER ;

1. Run the SQL and create the Stored Procedure, it should appear in the Stored Procedures list. You may need to refresh the list

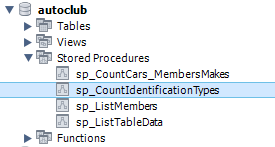


Figure 4.35 New stored procedure in the list

1. So now we need to test it. Open a new SQL Tab and enter the following test script

-- Create a template in a variable to pass into the stored procedure

SET @DocumentTemplate = (SELECT TemplateText FROM autoclubtemplates.templates WHERE TemplateType='GB\_Identification\_Report');

-- Call the stored procedure passing in the template

Call sp\_CountIdentificationTypes(@DocumentTemplate);

-- Select the template variable for output

SELECT @DocumentTemplate;

1. We’ll explain it shortly, run it and check the output which should be

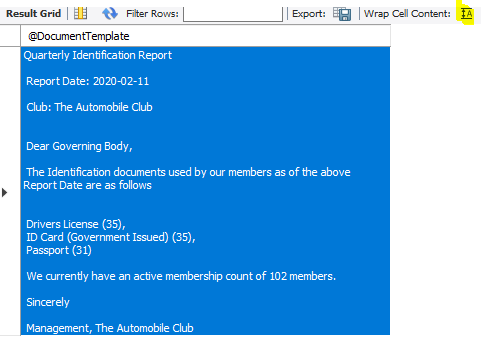


Figure 4.36 Output from the test SQL

Note:

When you run the script, the output will be on a single line in the results grid, click the wrap cell contents button, highlighted in above figure to wrap the contents so you can view them with the line feeds as shown.

So let us break this down, we will start with the .sql script you ran.

**CreateTemplateDatabase.sql**The script creates a new database named **autoclubtemplates** which contains one table only named **templates** which contains two records.

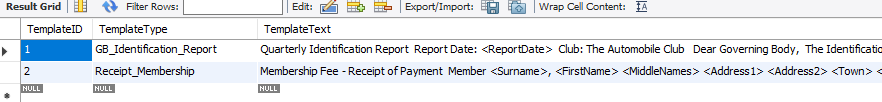


Figure 4.37 Records in the Template table

The records are the templates used in this exercise and the next activity, the raw template data for this exercise is (blank lines have been removed), as you can see, there are three **tags** in the text, these are where the database data will go.

Quarterly Identification Report

Report Date: **<ReportDate>**

Club: The Automobile Club

Dear Governing Body,

The Identification documents used by our members as of the above Report Date are as follows

**<IdentificationData>**

We currently have an active membership count of **<TotalMembers>** members.

Sincerely

Management, The Automobile Club

We will get back to this shortly.

**Stored Procedure**

All of the work happens in the stored procedure, the following is a breakdown of the procedure.

1. CREATE PROCEDURE `sp\_CountIdentificationTypes`( INOUT TemplateText LongText)

The standard declaration for a stored procedure with the name and one INOUT parameter. TemplateText has been assigned a datatype of LONGTEXT to allow it to accept huge templates if required. Even though we are using a small template, LONGTEXT works fine, we are only getting back the text we need, nothing extra.

1. BEGIN

The beginning of the SQL statements

1. SELECT GROUP\_CONCAT( DISTINCT TheText ) INTO @IDCount

Now things are getting interesting, we are getting the data from an embedded SQL statement which returns three, single column records from our database, the column name is **TheText**. We want to assign the values of those three records into the variable @IDCount but we can only assign one value so we use the GROUP\_CONCAT which will take the three, single-column rows of data and combine them on the field named **TheText** into one single string. GROUP\_CONCAT also adds the comma separators.



Figure 4.38 Single string generated from three records by GROUP\_CONCAT

Note:

The string is on separate lines because new lines /n was embedded into the data when each line was built in the SQL code.

1. FROM

The next lines define our data source

1. (

The embedded SQL must be enclosed in brackets, this is the opening bracket

1. SELECT (CONCAT('\n',identificationtype.identificationtype, ' (', Count(identificationtype.ID),')' ) ) as TheText

In this line we are SELECTING a value and putting it in the field named **TheText**, the value we are selecting is built by

CONCAT('\n',identificationtype.identificationtype, ' (', Count(identificationtype.ID),')' ))

|  |  |
| --- | --- |
| '\n' | Add a New Line to the output before the data is output |
| identificationtype.identificationtype | Retrieves the IdentificationType |
| ' (' | Opening bracket for the count to be put in |
| Count(identificationtype.ID) | The count of this identificationtype |
| ')' | Closing bracket for the count to be put in |

1. FROM identification

INNER JOIN identificationtype ON identification.IDType = identificationtype.ID

This is defining the tables where the data is coming from and how they are joined

1. GROUP BY identificationtype.identificationtype

This line ensures the data is grouped by the identificationtype used, this results in three records in our database

1. ) sq;

Closes of the subquery brackets, **sq** is a name we need to assign to the subquery

1. The next section is another query that counts the Active members and assigns the value to the variable @ActiveMemberCount

SELECT

Count(members.Active) INTO @ActiveMemberCount

FROM

members

WHERE

members.Active <> 0;

1. The next three lines replace **tags** in TemplateText with the values

SET TemplateText = REPLACE(TemplateText,'<ReportDate>', CURDATE());

SET TemplateText = REPLACE(TemplateText,'<IdentificationData>', @IDCount);

SET TemplateText = REPLACE(TemplateText,'<TotalMembers>', CONVERT(@ActiveMemberCount,UNSIGNED));

* 1. <ReportDate> is replaced with the current date
  2. <IdentificationData> is replaced with the returned from the GROUP\_CONCAT() statement
  3. <TotalMembers> is replaced with the @ActiveMemberCount value. This value is converted to an UNSIGNED integer during the replacement.

1. END

The end

The stored procedure is receiving the template text in the INOUT parameter, querying the database to get the information required and assigning the information to the relevant tags, finally sending the modified text back out through the INOUT parameter to the calling routine.

Now for the calling routine

**The Test SQL or Calling Procedure**

This is only doing three things, getting the template text, sending it to the stored procedure and then outputting the modified template text.

1. SET @DocumentTemplate = (SELECT TemplateText FROM autoclubtemplates.templates WHERE   
   TemplateType='GB\_Identification\_Report');  
   This line is retrieving the template text from the **autoclubtemplates** database, remember the .sql file I got you to run as first step. It is a simple SELECT statement filtering to the template we want and assigning the entire text to the variable **@DocumentTemplate**. It’s pretty cool you can actually pull data out from another database, remember, we are working in the **autoclub** database.
2. Call sp\_CountIdentificationTypes(@DocumentTemplate);  
   This line is calling the stored procedure, passing in the template text now residing in **@DocumentTemplate.** The stored procedure will modify the text and put the new text back in the same variable **@DocumentTemplate.**
3. SELECT @DocumentTemplate;  
   This line SELECTs the current (modified) value of **@DocumentTemplate** causing it to be output

And you know the resulting output, our template has gone from this

Quarterly Identification Report

Report Date: **<ReportDate>**

Club: The Automobile Club

Dear Governing Body,

The Identification documents used by our members as of the above Report Date are as follows

**<IdentificationData>**

We currently have an active membership count of **<TotalMembers>** members.

Sincerely

Management, The Automobile Club

To this

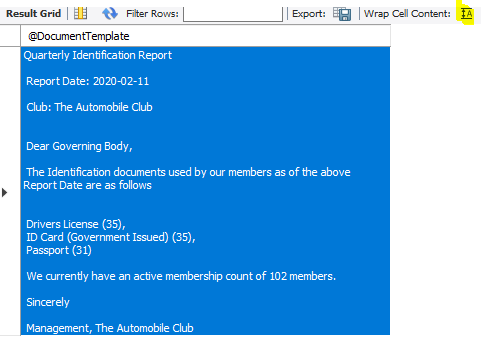


Figure 4.39 The output again for a side by side comparison with the original template text

**The conclusion to Exercise 11 – INOUT to fill-in templates**

Working with templates and data this way automates report and document generation with standardized output, accurate data and minimal interaction with humans. The output can be printed on company letterhead, can include images such as pics and digitized signatures and anything else you can store in the database. The templates can be stored in the same database or in another database and accessed as required, whenever the template needs modifications, the changes will automatically be picked up by the stored procedures, of course if new tags are added to the template, the stored procedure will also need to be modified to retrieve the new tags data and to insert it.

In the next activity you will be creating a stored procedure to generate a report as we have done in Exercise 11.

**Activity 4 – Filling in Templates**

We will continue with the template theme for this activity by creating a Receipt for Membership Fees. The activity will be a little different in that we will provide most of the script, you will fill in the blanks and when you are done and have it working, we will go through the breakdown of the scripts, The template is already supplied in the **autoclubtemplates** database and the text files are all available in **Chapter04/Activity04 resources** folder, let us begin.

1. Open a new SQL Query tab
2. Locate the file **Activity 4 – Filling in the Blanks.txt** from the **Chapter04/Activity04 resources** folder, copy the text in the file and paste it into the SQL Query tab.
3. You will notice there are several **#####** scattered around, your task is to replace all of these with the correct SQL text.
4. When done, run the SQL to create the stored procedure.
5. Next you need to create the test script, that will
   1. Extract the template from the **autoclubtemplates** database, the TemplateType is named **Receipt\_Membership**
   2. Call the stored procedure and pass in a MembersID and the Template text
   3. Return the template text
6. Run the Test Script and check your results.

The template text for the Membership\_Receipt report is

Membership Fee - Receipt of Payment

Member

<Surname>, <FirstName> <MiddleNames>

<Address1>

<Address2>

<Town> <State> <Postcode>

Payment Details

Date Paid : <PaymentDate>

Amount Paid: <Amount>

Message from the club

Dear <FirstName>,

We would like to thank you for renewing your membership.

We hope to see you at the many events we organise for our members during the next year.

Management

**The Solution to Activity 4 – Filling in Templates**

The solution to this activity is as follows

1. You will have opened a new SQL Query tab
2. You will have located the file **Activity 4 – Filling in the Blanks.txt** and pasted the contents into the SQL Query tab



Figure 4.40 Partial screenshot of the **Activity 4 – Filling in the Blanks.txt** file pasted into the SQL tab

1. You will have located all the **#####’s** and replaced them with the proper values, the SQL will now look like (comments have been removed)

DROP procedure IF exists sp\_MembersFeeReceipt;

DELIMITER $$

CREATE PROCEDURE `sp\_MembersFeeReceipt`( IN MemberID VARCHAR(5), INOUT TemplateText LongText)

BEGIN

SELECT

members.Surname,

members.FirstName,

members.MiddleNames,

memberaddress.StreetAddress1,

memberaddress.StreetAddress2,

memberaddress.Town,

states.State,

memberaddress.Postcode,

membershipfees.DatePaid,

membershipfees.FeeAmount,

members.ID

INTO

@Surname,

@FirstName,

@MiddleNames,

@StreetAddress1,

@StreetAddress2,

@Town,

@State,

@PostCode,

@DatePaid,

@AmountPaid,

@MemberNumber

FROM

members

INNER JOIN memberaddress ON memberaddress.MemberID = members.ID

INNER JOIN membershipfees ON membershipfees.MemberID = members.ID

INNER JOIN states ON memberaddress.State = states.ID

WHERE

members.ID = MemberID;

-- The TemplateText is now modified and then passed back

SET TemplateText = REPLACE(TemplateText,'<Surname>', @Surname);

SET TemplateText = REPLACE(TemplateText,'<FirstName>', @FirstName);

SET TemplateText = REPLACE(TemplateText,'<MiddleNames>', COALESCE(@MiddleNames,""));

SET TemplateText = REPLACE(TemplateText,'<Address1>', @StreetAddress1);

SET TemplateText = REPLACE(TemplateText,'<Address2>', COALESCE(@StreetAddress2,""));

SET TemplateText = REPLACE(TemplateText,'<Town>', @Town);

SET TemplateText = REPLACE(TemplateText,'<State>', @State);

SET TemplateText = REPLACE(TemplateText,'<Postcode>', @Postcode);

SET TemplateText = REPLACE(TemplateText,'<PaymentDate>', @DatePaid);

SET TemplateText = REPLACE(TemplateText,'<Amount>', CONCAT("$",FORMAT(@AmountPaid,2)));

END$$

DELIMITER ;

1. After running the SQL, the new stored procedure **sp\_MembersFeeReceipt** will have appeared in the Procedure List

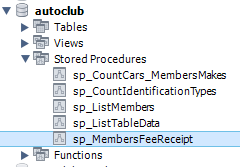


Figure 4.41 New procedure **sp\_MembersFeeReceipt** in procedure list

1. To create the **Test Script** you will have opened a new SQL Query tab and entered the SQL that will have
   1. Extracted the TemplateType named **Receipt\_Membership** from the **autoclubtemplates** database into a variable
   2. Called the stored procedure and pass in a MembersID and the Template text variable
   3. Returned the template text variable as output from the Test SQL

And the Test SQL will look like

SET @DocumentTemplate =

(

SELECT TemplateText

FROM autoclubtemplates.templates

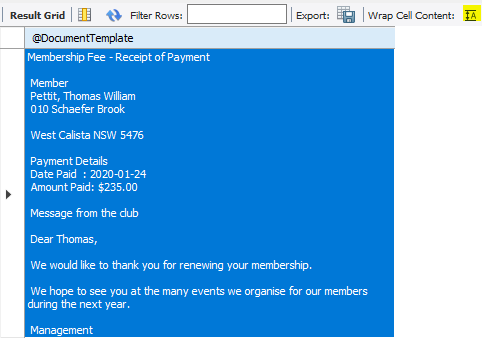
WHERE TemplateType='Receipt\_Membership'

);

Call sp\_MembersFeeReceipt('2',@DocumentTemplate);

SELECT @DocumentTemplate;

1. After running the test script, your output would have been for member 2. You will have clicked the **Wrap Cell Contents** icon to view the formatted text.



**Figure 4.42 Membership Fee Receipt for Member 2**

1. You would have said Woo Hoo, and then tried another MemberID, you can try 1 to 100 although there are some gaps.

Note

The SQL and stored procedure are meant as a learning tool to populate templates and do not contain validation or checks that a MemberID passed in are actually in the dataset, if you enter a number not in the dataset, the details of the last valid member you entered are returned. In a real-life scenario, you would add validation on the parameters.

Lets now break the SQL down and see what it is doing, we’ll start with the Test Script, short and sweet.

**Test Script**

This is almost identical to the test script in Exercise 11 with only two differences

SET @DocumentTemplate =

(

SELECT TemplateText

FROM autoclubtemplates.templates

WHERE TemplateType='Receipt\_Membership'

);

This line retrieves the Template Text for the **Receipt\_Membership** template, it then assigns the text to the **@DocumentTemplate** variable so we can use it later. At this stage the template is the raw text with the tags in place

Call sp\_MembersFeeReceipt('2',@DocumentTemplate);

This line passes a **MembersID** value and **@DocumentTemplate** into the the **sp\_MembersFeeReceipt** stored procedure. When the procedure is finished, **@DocumentTemplate** will have had all the tags replaced with data from the database pertinent to the MembersID passed in.

SELECT @DocumentTemplate;

And finally, this line passes the value in **@DocumentTemplate** back as the output for the test script.

The only differences between this activities test script and that of Exercise 11 is the template we used and the stored procedure it was passed into.

**Stored Procedure**

Although there were many more data fields, variables and replacements performed in this stored procedure, it only consisted of one query only and several **REPLACE** command, some with formatting. There was no fancy GROUP\_CONCAT to be concerned with, so let us break it down

DROP procedure IF exists sp\_MembersFeeReceipt;

DELIMITER $$

CREATE PROCEDURE `sp\_MembersFeeReceipt`( IN MemberID VARCHAR(5), INOUT TemplateText LongText)

BEGIN

This is the standard DROP and CREATE we have been using, the user-defined DELIMITER is the same, nothing new there as with the naming of the stored procedure. We are passing in two parameters, the first is the **MembersID** so we know whom to create the receipt for and the second is the **TemplateText** with the tags in place. And of course BEGIN so we know the SQL will start next

SELECT

members.Surname,

members.FirstName,

members.MiddleNames,

memberaddress.StreetAddress1,

memberaddress.StreetAddress2,

memberaddress.Town,

states.State,

memberaddress.Postcode,

membershipfees.DatePaid,

membershipfees.FeeAmount,

members.ID

The first part is a standard SELECT statement listing all the fields we are extracting from the database, each field has the source table name included so we know where to get the data from

INTO

@Surname,

@FirstName,

@MiddleNames,

@StreetAddress1,

@StreetAddress2,

@Town,

@State,

@PostCode,

@DatePaid,

@AmountPaid,

@MemberNumber

So this is a little more interesting, the INTO command is saying take all the data from the select statement and put it into the following variables. The variable start with the @ sign to indicate it is a variable and they are listed in the exact order of the fields, this is important, get them out of order and you will end up with data in the wrong variable. Where possible, try to name the variables after their source field, it just makes it easier when the time comes to use them.

FROM

members

INNER JOIN memberaddress ON memberaddress.MemberID = members.ID

INNER JOIN membershipfees ON membershipfees.MemberID = members.ID

INNER JOIN states ON memberaddress.State = states.ID

WHERE

members.ID = MemberID;

The FROM clause is standard, it is joining the tables as we need them to be.

The WHERE clause is the filter, filtering on the **members.ID** field and the value to filter on is in the passed in parameter **MemberID**.

Now comes the fun stuff, the above query filtered and read the data and assigned the values to all those variable, now we can use the variables to replace the tags. The lines will be grouped for explanation purposes, by the way, the order of the lines is not relevant, they will work just fine in this order too.

-- The TemplateText is now modified and then passed back

SET TemplateText = REPLACE(TemplateText,'<Surname>', @Surname);

SET TemplateText = REPLACE(TemplateText,'<FirstName>', @FirstName);

SET TemplateText = REPLACE(TemplateText,'<Address1>', @StreetAddress1);

SET TemplateText = REPLACE(TemplateText,'<Town>', @Town);

SET TemplateText = REPLACE(TemplateText,'<State>', @State);

SET TemplateText = REPLACE(TemplateText,'<Postcode>', @Postcode);

SET TemplateText = REPLACE(TemplateText,'<PaymentDate>', @DatePaid);

These lines are taken out of order to the SQL and grouped as they are the same, we have explained the REPLACE command in Exercise 11 so we do not need to go into too much detail in it, these lines are simply dropping the variables into where the tags were.

SET TemplateText = REPLACE(TemplateText,'<MiddleNames>', COALESCE(@MiddleNames,""));

SET TemplateText = REPLACE(TemplateText,'<Address2>', COALESCE(@StreetAddress2,""));

We have spoken about COALESCE before, we are using it in these two lines because it is highly probable that these two lines could contain NULL values and we just don’t want them. COALESCE is looking at the variable name, if it contains a value then the value of the variable is used otherwise if it contains a NULL then an empty string is used to replace the tag. A NULL will cause an error and the stored procedure will fail so handling NULLs is paramount. You may even include the COALESCE in all REPLACE statements if you are really concerned, or you could check all the variables before getting into the REPLACE statement, i.e. **@StreetAddress2 = COALESCE(@StreetAddress2,"");** will have worked and the COALESCE would not be required in the REPLACE.

SET TemplateText = REPLACE(TemplateText,'<Amount>', CONCAT("$",FORMAT(@AmountPaid,2)));

AmountPid is a currency value so here we do two things,

1. The FORMAT command is telling it we want two decimal places in the value, this is so the value is not rounded if there are decimals.
2. CONCAT is prepending a dollar sign ($) in front of the value

And then the formatted text is placed where the tag is

END$$

DELIMITER ;

And of course, close the user-defined DELIMITER and reset the default.

And we are done.