Table of Contents

[About this document 2](#_Toc267727466)

[Overview 2](#_Toc267727467)

[Getting Started. 3](#_Toc267727468)

[How to define a template. 3](#_Toc267727469)

[How to provide data. 3](#_Toc267727470)

[Using ADO.Net DataTables 4](#_Toc267727471)

[Using Variables 5](#_Toc267727472)

[Creating the report. 6](#_Toc267727473)

[Iterating over DataTables 7](#_Toc267727474)

[Filtering Data 8](#_Toc267727475)

[Handling Events 9](#_Toc267727476)

[BeforeScanRecord Event 9](#_Toc267727477)

[AfterScanRecordEvent 10](#_Toc267727478)

[Generating conditional output (‘IF’ Construction) 10](#_Toc267727479)

# About this document

This document intends to be a quick start up guide for using NTemplates. It introduces the most important features and concepts about the product to help you quickly understand them so that you can jump into the examples and start working right away.

Overview.

NTemplates is a non visual reporting component for .NET written 100% in C#. It allows your end users to define custom report templates (RTF format). This way, your application just needs to specify the placeholders for letting the user build a template, and the required data for building the report. NTemplates will generate the report based on the users’ report template.

The final report is also a RTF document, so it can be edited or used in any other way an RTF document could be used.

NTemplates includes a very simple interpreted programming language. This allows templates complexity vary from very simple placeholder replacement, to much more complex report definitions; including IF and LOOP constructions.

# Getting Started.

## How to define a template.

A report template is a simple .rtf document that can be written using Microsoft Word or other RTF editors. It must contain at least one placeholder. Placeholders are ‘special words’ that NTemplates will replace with actual data to generate a report.

Placeholders, like programming sentences and functions need to be written enclosed by # simbol. For example, these could be some valid placeholders for data:

* #DayOfWeek#
* #P.FriendName#

The first one represents a memory variable that could store the day of the week, for example. And the second one, represents a DataColumn called ‘FriendName’ inside a DataTable object, which TableName’s property value is ‘P’.

So, imagine a simple RTF document with this content:

Hello #P.name#. Today is #DayOfWeek#, and I’d just wanted to let you know about NTemplates.

If your application provide values for these placeholders, let’s say ‘Monday’ and ‘John’ and then ask NTemplates to generate a report, the result would be as follows:

Hello John. Today is Monday, and I’d just wanted to let you know about NTemplates.

The process preserves the formatting options. So if you formatted the text like this:

Hello **#P.name#**. Today is *#DayOfWeek#*, and I’d just wanted to let you know about NTemplates.

Then you would get this result:

Hello **John**. Today is *Monday*, and I’d just wanted to let you know about NTemplates.

A simple rule to follow is to format the whole placeholder. Note that in the above example, the enclosing # symbols have the same format than the enclosed word. If you did something like *#****Day***Of*Week*# then the engine would not be able to replace the placeholder and exception could be raised.

## How to provide data.

NTemplates exposes a simple object oriented API to work with. Data can be provided to NTemplates engine in the form of some ADO.Net DataTable or List<T> instances; or simple variables, depending on the requirements.

### Using ADO.Net DataTables

The following code snipped shows how to pass several DataTable instances to the DocumentCreator. In this case is assumed that the tables are taken from inside a DataSet, but a Dataset is not required.

//Document creator represents the API to the reporting engine.

DocumentCreator dc = new DocumentCreator();

//Add tables contained inside a Dataset

foreach (DataTable dt in MyData.Tables)

dc.AddDataTable(dt);

After the Data Tables are added to the DocumentCreator, they are ready to be used by NTemplates.

### Using Variables

Variables may be added to the document creator using the corresponding Add method specifying a unique name. Later, they may be accessed through their name using a Get method.

Currently, four primitive data types are suppurted: Int32, String, DateTime and Double.

The following matrix show what are the methods needed to work with variables:

|  |  |  |
| --- | --- | --- |
| **DataType** | **Method** | **Action** |
| Int32 | AddInt32(string variableName, int value) | Adds or updates a variable of type int. |
| int GetInt32(string variableName) | Returns a value of type int for the specified name. |
| String | AddString(string variableName, string value) | Adds or updates a variable of type string. |
| GetString(string variableName) | Returns a value of type string for the specified name. |
| DateTime | AddDateTime(string variableName, DateTime value) | Adds or updates a variable of type DateTime. |
| GetDateTime(string variableName) | Returns a value of type DateTime for the specified name. |
| Double | AddDouble(string variableName, double value) | Adds or updates a variable of type double. |
| GetDouble(string variableName) | Returns a value of type double for the specified name. |

All variables and tables are stored inside an object called DataManager. Besides the abovementioned methods, the DataManager class provides methods for clearing all variables and tables from inside it. Those methods are DataManager.ResetVariables() and DataManager.ResetTables() respectively. They are not likely to be used frequently but they may be necessary in some scenarios.

Finally, the CreateDocument method is called. It’s parameters represent the full path for the report template and the output document (to be created) respectively.

### Creating the report.

After adding all the necessary variables and tables, everything is set up to create the report. The easiest way of doing this is by calling the CreateDocument method. It has several overloads:

public void CreateDocument(string inputFile, string outputFile)

This overload creates a document named with the parameter specified for outputFile argument (full path). For example:

dc.CreateDocument(@"..\InputTemplate.rtf", @"..\OutputReport.rtf");

Another overload let’s you pass a report template as a stream. For example, you may store your report template in a database and load it to a memory stream. Or perhaps it could be retrieved as an array of bytes from a WCF service and then you could load it into a memory stream to use it on the client side.

public void CreateDocument(Stream inputStream, string outputFile)

On the other hand, you may be programing a server side reporting application. In that case perhaps it could be necessary to store the resulting report into a memory stream to return it to the client or to store it in a database. For those purposes the DocumentCreator class provides the CreateDocumentToMemoryStream method which in turn has two overloads:

public MemoryStream CreateDocumentToMemoryStream(string inputFile)

public MemoryStream CreateDocumentToMemoryStream(Stream inputStream)

The first one allows specifying a file path representing the report template. The second one allows to pass a stream containing the template data. It assumes the RTF text is encoded as ASCII.

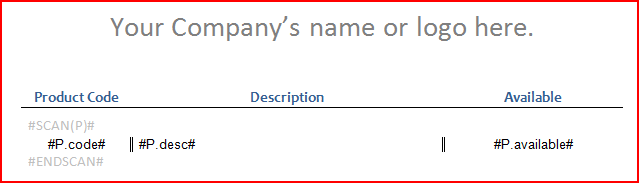
With these four methods for creating reports you have full flexibility to integrate NTemplates in your .Net application.

## Iterating over DataTables

The SCAN sentence is available for iterating DataTables and Lists. Supose we have a simple DataTable representing a product list. For our example we’ll use a DataTable named ‘P’ with 3 columns:

* Product Code
* Product Description
* Availability

Suposing we need to list all the records, we could create a template like this one:



As you can see, the placeholders for the fields we want to report appear in the same line (this is not a restriction). The preceding line contains only the SCAN(P) sentence and the following line contains only the closing ENDSCAN.

This means that anything between SCAN and ENDSCAN will be repeated for each record in the DataTable ‘P’.

Note that the placeholders are contained inside single cells of table in the RTF document and only the internal vertical borders of the cells are visible. This is just the desired appearance for the report in this example. The important thing to note here is that the table will automatically grow to contain all the records.

The SCAN and ENDSCAN words are presented in grey. This is just for better visualization of the template. Those words will be removed from the final report anyway.

Important:

It’s mandatory to write the SCAN and ENDSCAN words, including the enclosing simbols # with the same format and colour. And when used inside a table, they must be placed inside the same cell. Otherwise unpredicted output could be produced.

### Filtering Data

Tipically when creating reports with a plain data structure (no hierarchical data) you will provide a DataTable or List<T> ready to be used, meaning that the DataTable will already contain the necessary information filtered. However, you can apply aditional filters by reporting only those records that meet certain condition. For that purpose the SCAN FOR construction is available.

#SCAN(TableName) FOR (<Boolean Condition>)#

… Some RTF Text: #TableName.ColumnName#

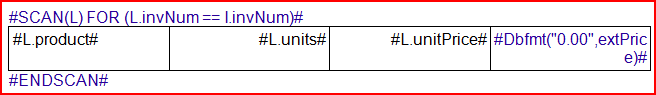
#ENDSCAN#

<Boolean Condition> represents a c# expression that evaluates to true or false. The restriction is that the expression must refer only to known variables or table columns. [[1]](#footnote-1)

You can nest as much SCAN loops as needed. This allows you to build parent-child reports. But any time you refer to a table column NTemplates will consider it’s current value.

Imagine you have nested two scans. In the inner-most you use a column value of the outer-most for any reason. NTemplates will pick the value of that field for the current record. That is, the record that is being scanned.

The following picture is an example of an invoice reporting template. As you can see, two DataTables are being used: ‘I’ for ‘Invoice’ and ‘L’ for ‘Lines’. The condition used in this case makes the SCAN loop over all the table L, but only outputs the records that match the condition (line number == invoice number).



The right-most column is presenting a formatted value. We´ll be analyzing formatting functions ahead in this document.

## Handling Events

When NTemplates is performing a SCAN, it throws some events that your application could handle in order to perform calculations such as totals. All of these events provide information packed in their corresponding ‘Event Args’ class. These classes all derive from NTemplates.ScanRecordEventArgsBase providing a common set of properties with useful information.

**ScanRecordEventArgsBase class description.**

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Record | System.Data.DataRow | Current Record. |
| RecordPosition | System.Int32 | The index of the current record |
| DataManager | NTemplates.DataManager | The data manager object used for accessing data managed by the engine. |
| Cancel | System.Boolean | When set to true, aborts the loop. |
| TableName | System.String | The name of the DataTable being scanned. |

### BeforeScanRecord Event

Description: Raised just before a record is scanned.

Syntax:

BeforeScanRecord(object sender, BeforeScanRecordEventArgs e)

**BeforeScanRecordEventArgs class description.**

(Inherits ScanRecordEventArgsBase)

Properties

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| Skip | System.Boolean | When set to true, the record is skipped. |

Notes:

In the context of this event, the inherited properties Record and RecordPosition refer to the record to be scanned, as the prefix ***‘Before’*** sugests.

This event could be used to inspect the record data and perform some actions just before it’s presented at the report, or to provide additional filtering capabilities by setting the Skip property to true.

### AfterScanRecordEvent

Description: Raised just after a record is scanned.

Syntax:

AfterScanRecord(object sender, AfterScanRecordEventArgs e)

**AfterScanRecordEventArgs class description.**

(Inherits ScanRecordEventArgsBase)

Properties

|  |  |  |
| --- | --- | --- |
| **Property** | **Type** | **Description** |
| MatchesScanCondition | System.Boolean | Indicates wether the record just scanned matches the loop condition or not. |

Notes:

In the context of this event, the inherited properties Record and RecordPosition refer to the record just scanned, as the prefix *‘****After’*** sugests.

This event could be used to inspect the record data and perform some actions just after it’s presented at the report. For example to perform calculations such as totals.

## Generating conditional output (‘IF’ Construction)

Besides SCAN loop construction, you could place IF-ELSE constructions in the document.

#IF (<Boolean Condition>)#

… Some RTF Text: #TableName.ColumnName#

[#ELSE#]

… Some other RTF text you wish to place here….

#ENDIF#

<Boolean Condition> represents a c# expression that evaluates to true or false. The restriction is that the expression must refer only to known variables or table columns.

As with the SCAN construction, you could nest as many IF’s as needed.

You could also combine SCAN and IF constructions to generate conditional formatting for example, like this:



1. This means, DataTables or variables known to the DataManager object. [↑](#footnote-ref-1)