Responsys Personalization Language (RPL)

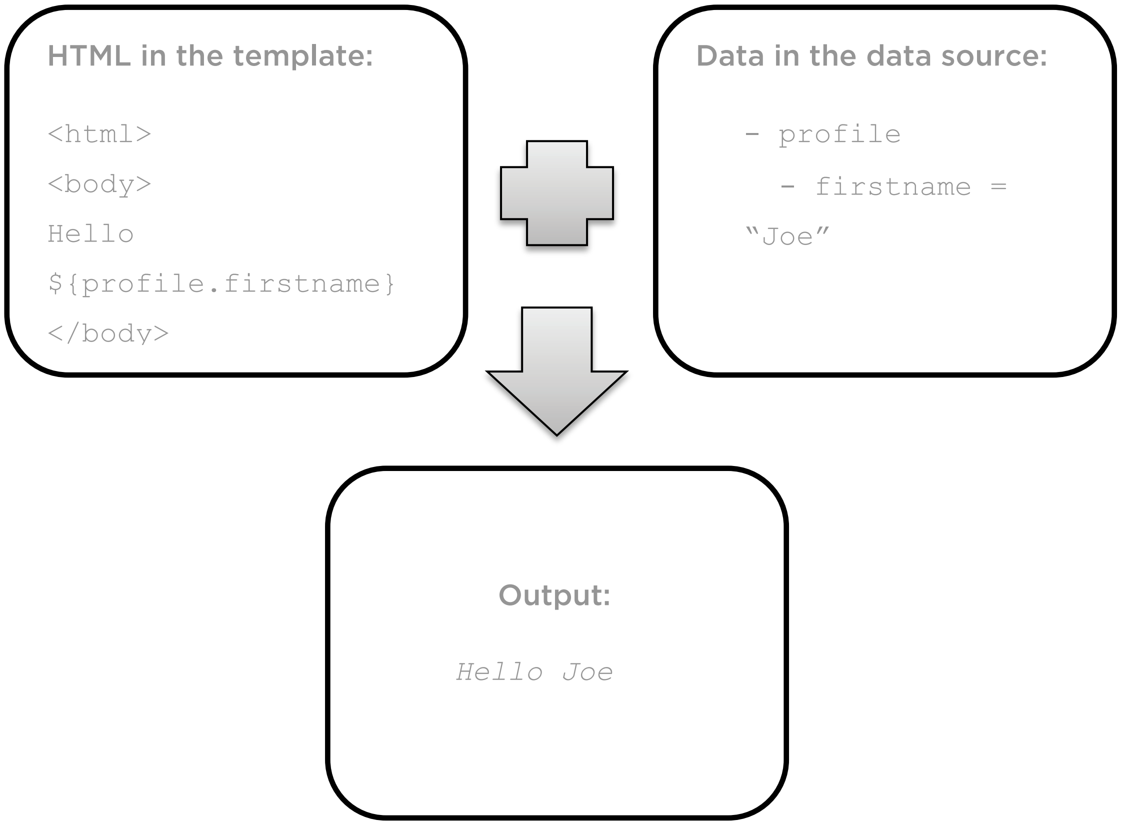
User Guide and Language Reference

Chapter 1. Introduction

Responsys Personalization Language (RPL) is the language used for generating text output based on templates. RPL is designed to generate HTML and Text for email, web and other uses.

Although RPL has many programming capabilities, it is not a full-blown programming language like PHP or Java. It is implemented to work only from within the Interact Suite. Other than that, you can accomplish many programming tasks using this language.

The following diagram shows how RPL executes templates and produces output.



Oracle Responsys personalization in the past

For a long time, personalization in Interact was based on what is commonly known as built-in functions. Built-in functions were developed through customer needs over the years and are a Responsys proprietary language. This language is limited as a programming language in its ability to do sophisticated personalization in an efficient manner. Over time, the next generation of the language was envisioned and it was decided to create a language more akin to a programming language to expand the power of personalization capabilities of Interact.

When the Message Designer for Email is turned on for an account, existing campaigns using the built-in functions will continue to work. You can create new campaigns that leverage the Message Designer for Email and RPL.

Document conventions

Variable names, template fragments, etc. are written like this: **`variableName`**.

If something should be replaced with a value, it is written like this: *`Hello yourName!`*.

New terms are written in *italics.*

Definitions of built-ins, directives, and methods are written like this:

```
Definition
```

Code examples are written like this:

```
Code example
```

Data model examples are written like this:

```
Data example
```

Output examples are written like this:

```
Output Example
```

Chapter 2. Getting Started

This chapter provides an introduction to RPL. Subsequent chapters provide greater detail. After you read this chapter, you will be able to write simple but useful RPL templates.

About templates

A *template* is a text document that can produce HTML, text, and other formats. This document is stored in the Content Library and can be edited, manipulated and controlled using the Content Library, or by working in the Message Designer for Email.

You can upload templates as files, and they will be stored in the Content Library. You can also enter definitions for text generation using the Message Designer for Email.

About output

Assume that you need a HTML message for an e-shop application, similar to this:

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome Joe!</h1>
  <p>Our latest product:
  <a href="products/greenmouse.html">green mouse</a>!
</body>
</html>
```

Let's say that the user's first name depends on who the message is sent to, as defined in the profile list, and the latest product should come from a supplemental table and thus can change at any time. In this situation, you can't just enter the user's first name and the URL and name of the latest product into the HTML. You can't use static HTML.

RPL's solution for this situation is to use a template instead of static HTML. The template contains instructions to RPL (highlighted in the following example):

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>Welcome ${profile.firstName}!</h1>
  <p>Our latest product:
  <a href="${latestProduct.url}">${latestProduct.name}</a>!
</body>
</html>
```

The template is stored in the Content Library, usually like a static HTML page. But when someone launches this page, RPL transforms the template on-the-fly to plain HTML by replacing the ${…}-s with up-to-date content (e.g., replacing *`${ profile.firstName}`* with the recipient's name), and sends the result to the recipient's inbox. As a result, the recipient receives the HTML without RPL instructions. The template stored in the Content Library does not change during this process, so the transformation will happen again and again for each individual. This ensures that the displayed information is always up-to-date.

About data retrieval

An important idea behind RPL is that presentation logic is separated from the data. The template contains only presentation issues, that is, visual design and formatting. It does not include instructions for finding out who the current recipient is, or how to query the database to find out what the latest product is. RPL determines what values to display by retrieving and processing the contents of rows in the datasources defined for a message. The way these values are calculated can change while the templates can remain the same, and the look of the email can be changed without touching anything but the template. This separation is especially useful when the template author and campaign manager are not the same individual.

The basic data utilization is defined in the campaigns and forms and needs no additional programming. However, RPL includes advanced data processing mechanisms that might be required for data operations. This section of the document describes working with standard profile and profile extension tables. Advanced functionality, such as working with supplemental tables, will be described later in this document.

In addition to data from the profile related tables, RPL provides a way to add more data. For example, you can assign additional values using the `assign`, `global` and `local` constructs.

About the data model

This section describes data structures and data coming from campaign variables, datasources, user defined data values, and data added by RPL constructs. Working with data from your datasources will be explained later in this document.

All data that the template can use is organized in the *data model*. The data model is a tree-like structure similar to folders and files on your computer. RPL uses the template and the data model to generate output.

The data in the template used in the examples above can be represented in the data model as:

```
  (root)
   |
   +- profile
   |    |
   |    +firstName = "Joe"
   |
   +- latestProduct
        |
        +- url = "products/greenmouse.html"
        |
        +- name = "green mouse"  Compare this with what you saw in the
template earlier: ${firstName} and ${latestProduct.name}.
```

As an analogy, the data model is similar to a computer file system: the root and `latestProduct` correspond to folders and `profile.firstName`, `url` and `name` correspond to files. `Url` and `name` are in the `latestProduct` directory.

The data model can be represented as the following tree structure:

```
(root)
  |
  +- mouse
  |   |
  |   +- size = "small"
  |   |
  |   +- price = 50
  |
  +- elephant
  |   |
  |   +- size = "large"
  |   |
  |   +- price = 5000
  |
  +- python
  |   |
  |   +- size = "medium"
  |   |
  |   +- price = 4999
  |
  +- test = "It is a test"
```

About values and types

A *value* is something that a variable can store. For example, in the following data model:

```
(root)
 |
 +- user = "Big Joe"
 |
 +- today = Jul 6, 2007
 |
 +- todayHoliday = false
 |
 +- lotteryNumbers
 |   |
 |   +- (1st) = 20
 |   |
 |   +- (2st) = 14
 |    |
 |    +- (3rd) = 42
 |    |
 |    +- (4th) = 8
 |    |
 |    +- (5th) = 15
 |
 +- cargo
      |
      +- name = "coal"
      |
      +- weight = 40
```

The value of the `user` variable is "Big Joe" (a string), the value of `today` is Jul 6, 2007 (a date), the value of `todayHoliday` is false (a boolean, i.e. yes/no). The value of `lotteryNumbers` is the sequence that contains 20, 14, 42, 8, 15. `lotteryNumbers` contains multiple values (for example, the value of the second item is 14), but `lotteryNumbers` itself is a single value, the value of `cargo` is a hash. A value need not be stored in a variable, for example we have the value 100 here:

```
<#if cargo.weight < 100>Light cargo</#if>
```

Temporary results of calculations are also values. 20 and 120 in the following example are values. When this template is executed, it will print 120:

```
${cargo.weight / 2 + 100}
```

Combing the last two examples, as the result of dividing the two values, 40 (the `weight` of the `cargo`) and 2, a new value 20 is created. Then 100 is added to it, so the value 120 is created. Then 120 is printed (${...}), and the template execution continues.

Each value is of a specific *type*. For example, the value of the `user` variable is of type string, and the `lotteryNumbers` variable is of type sequence. The value type is important because it determines to a large extent how and where you can use the value.

For example:

- `${user / 2}` is an error, but `${cargo.weight / 2}` is not and prints 20, since division makes sense for a number, but not for a string
- Using dot like in `cargo.name` makes sense only if cargo is a hash.
- You can use `<#list ...>` only with sequences.
- The condition of `<#if ...>` must be a boolean.

A value can be of multiple types at the same time, although it's rarely utilized. For example, in the data model below `mouse` is both a string and a hash:

```
(root)
 |
 +- mouse = "Jerri"
     |
     +- age = 12
     |
     +- color = "brown"
```

If you merge this template with the above data model:

```
${mouse}       <#-- uses mouse as a string -->
${mouse.age}   <#-- uses mouse as a hash -->
${mouse.color} <#-- uses mouse as a hash -->
```

the output will be:

```
Jerri
12
brown
```

Data model type

In the data model, the root is a value of type hash. When you write something like `user`, it means that you want the "user" variable stored in the root hash. This is similar to `root.user`, except that there is no variable called "root", so that would not work. We call this root the root namespace.

Note that in that our example, the data model, that is the root hash, contains further hashes and sequences (`lotteryNumbers` and `cargo`). This is because a hash contains other variables, and those variables have a value, which can be of any type, including a hash or sequence.

Supported types

RPL supports the following types:

- Scalars:
  - String
  - Number
  - Boolean
  - Date
- Containers:
  - Hash
  - Sequence
  - Collection
- Subroutines:
  - Methods and functions
  - User-defined directives
- Miscellaneous/seldom used:
  - Node

About scalars

A variable that stores a single value (`size`, `price`, and `test`).

To use a scalar in a template, you specify its path from the directory, separated by a period. For example, to access the price of a mouse, write `mouse.price`. When you put the `${...}` code around an expression like this (`${mouse.price}`, you are instructing RPL to output the corresponding text based on the value of the scalar.

These are the basic values of one of the following type:

String

Simple text.

Number

RPL does not distinguish between whole numbers and non-whole numbers. For example, for example 3/2 will be always 1.5, not 1.

Note that "50" and the number 50 are different in RPL. The former is a string of two characters, while the latter is a numerical value.

Boolean

A boolean value represents a logical true or false (yes or no). For example, whether or not the visitor has logged in. Typically, you use booleans as the condition of the `if` directive, for example: `<#if loggedIn >...</#if> or <#if price == 0>...</#if>`; in the last case, the result of the price == 0 part is a boolean value.

Date

A date variable stores date/time related data. It has three variations:

- A date with day precision (often referred to simply as "date") as April 4, 2003
- Time of day (without the date), as 10:19:18 PM. Time is stored with millisecond precision.
- Date-time (sometimes called "time stamp") as April 4, 2003 10:19:18 PM. The time part is stored with millisecond precision.

Due to system limitation, RPL cannot always determine which parts of the date are in use (i.e., if it is date-time, or a time of day, etc.). The solution for this problem is discussed later in this chapter.

It is possible to define date values directly in templates using the `date`, `time` and `datetime` built-ins. These built-ins are described in *"Chapter 5. Built-in Reference"*.

IMPORTANT: Bear in mind that RPL distinguishes strings from numbers and booleans, so the string "150" and the number 150 are different.

About containers

Container variables hold other variables, referred to as *sub-variables*. RPL supports the following container types:

Hash
A variable that acts as a directory (`root`, `mouse`, `elephant`, and `python`). Hashes store other variables called *sub-variables*, by name (e.g., `mouse` and `size`).
A hash associates a unique lookup name with each of its sub-variables. The name is an unrestricted string. A hash does not define an order for the sub-variables in it. The variables are accessed by name.

Note that since a value can have multiple types, it is possible for a value to be both a hash and a sequence. In this case, the value supports index-based access as well as access by lookup name. However, typically a container will be either a hash or a sequence, not both.

As the value of the variables stored in hashes and sequences (and collections) can be anything, it can be a hash or sequence (or collection) as well. This way you can build arbitrarily deep structures.

The data-model (or its root) is a hash. It is also called the root namespace.

Sequence
Associates an integer number with each of its sub-variables. A sequence is similar to a hash, but it stores sub-variables sequentially by index number.

The first sub-variable is associated with 0, the second with 1, the third to 2, and so on. These numbers are often called the *indexes* of the sub-variables. The sub-variables are ordered. Sequences are usually dense, i.e., all indexes up to the index of the last sub-variable have an associated sub-variable, but it's not strictly necessary. The type of the sub-variable values need not be the same.

For example, in the following data model, `animals` and `whatnot.fruits` are sequences:

```
(root)
 |
 +- animals
 |   |
 |   +- (1st)
 |   |   |
 |   |   +- name = "mouse"
 |   |   |
 |   |   +- size = "small"
 |   |   |
 |   |   +- price = 50
 |   |
 |   +- (2nd)
 |   |   |
 |   |   +- name = "elephant"
 |   |   |
 |   |   +- size = "large"
 |   |   |
 |   |   +- price = 5000
 |   |
 |   +- (3rd)
 |       |
 |       +- name = "python"
 |       |
 |       +- size = "medium"
 |       |
 |       +- price = 4999
 |
 +- whatnot
     |
     +- fruits
         |
         +- (1st) = "orange"
         |
         +- (2nd) = "banana"
```

To access a sub-variable of a sequence, you use a numerical index in square brackets. Indexes start from 0, thus the index of the first item is 0. To get the name of the first animal you write `animals[0].name`. To get the second item in `whatnot.fruits` (which is the string "banana") you write `whatnot.fruits[1]`.

Note that since a value can have multiple types, it is possible for a value to be both a hash and a sequence. In this case, the value supports index-based access as well as access by lookup name. However, typically a container will be either a hash or a sequence, not both.

Collection
A restricted sequence. You cannot access its size or retrieve its sub-variables by index, but you can list them with the `list` directive.

About subroutines

Subroutines include methods, functions, and user-defined directives.

About methods and functions

Methods and *functions* are used to calculate a value, influenced by the parameters you give to it.

The difference between methods and functions is that methods are typically defined by the language and the data-model, and functions are defined in templates using the `function` directive. Both methods and functions can be used in the same way.

Methods/functions are first-class values, just like in functional programming languages. This means that functions/methods can be the parameters or return values of other functions/methods, you can assign them to variables, and so on.

RPL provides predefined methods as described in *"Chapter 7. Method Reference"*. It is also possible to create methods and functions using the `function` directive.

Consider the provided method `avg`, which is placed in the root of the data model as part of the language definition. This method can be used to calculate the average of numbers.

The usage of methods will be explained in detail in a later section, but the following example illustrates methods:

```
The average of 3 and 5 is: ${avg(3, 5)}
The average of 6 and 10 and 20 is: ${avg(6, 10, 20)}
The average of the price of a python and an elephant is:
${avg(animals.python.price, animals.elephant.price)}
```

produces this output:

```
The average of 3 and 5 is: 4
The average of 6 and 10 and 20 is: 12
The average of the price of a python and an elephant is:
4999.5
```

About user-defined directives

A *user-defined directive* is a reusable template fragment which can be used as an RPL tag. Creating user-defined directives is described later in this chapter.

User-defined directives (such as macros), are first-class values similar to functions/methods.

For example, assume the value of a variable `box` is a user-defined directive that prints an HTML message with a title bar and a message. You can use the `box` variable in the template as shown in the following example:

```
<@box title="Attention!">
  Too much copy-pasting may lead to
  maintenance headaches.
</@box>
```

Using methods/functions versus user-defined directives

As a general rule, you should use a user-defined directive instead of a function/method in the following cases:

- The output (the return value) is markup (HTML, XML, etc.). The main reason is that the results of functions are subject to automatic XML escaping (due to the nature of ${...}), while the output of user-defined directives are not (due to the nature of <@...>; its output is assumed to be markup, and hence already escaped).

- The side-effect is important and not the return value, for example, a directive whose purpose is to add an entry to the server log. In fact, a user-defined directive cannot have a return value, but some type of feedback is still possible by setting non-local variables.
- You need to perform flow control.

About miscellaneous types

Node variables represent a node in a tree structure, and are used mostly with XML processing.

A node is similar to a sequence that stores other nodes, referred to as *child nodes*. A node stores a reference to its container node, referred to as the *parent node*. The node value can be of multiple types, for example both a node and a number. In such cases, it can store a number as the "pay-load". In addition to the topological information, a node can store meta data such as a node name, a node type (string), and a node namespace (string). For example, if the node symbolizes an h1 element in an XHTML document, then its name is h1, its node type is element, and its namespace can be http://www.w3.org/1999/xhtml. The designer of the data model determines what this meta data means and whether it is used. Working with templates

Templates are programs you write in RPL. A template comprises the following:

Text	Text is printed to the output as is.
Interpolation	Produces a calculated value in the output. Interpolations are delimited by ${ and }.
RPL tags	Instructions to RPL and will not be printed to the output. You use RPL tags to call directives. RPL tags are similar to HTML tags.
Comments	Comments are ignored by RPL and are not written to the output. RPL comments are similar to HTML comments, but are delimited by <#-- and -->.

In the following example, the template sections are highlighted so that you can easily see them. Comments are highlighted in blue, interpolations are highlighted in green, RPL tags are highlighted in yellow, and text is highlighted in gray. The [BR]'s indicate line breaks, and HTML tags are not highlighted (RPL reads HTML as text and does not interpret it in any way).

```
<html>[BR]
<head>[BR]
  <title>Welcome!</title>[BR]
</head>[BR]
<body>[BR]
  <#-- Greet the user by his/her name -->[BR]
  <h1>Welcome ${profile.user}!</h1>[BR]
  <p>We have these animals:[BR]
  <ul>
  <#list animals as being>
    <li>${being.name} for ${being.price} Euros</li>
  </#list>
  </ul>
</body>[BR]
</html>
```

RPL distinguishes upper case and lower case letters. For example, `${name}` is not the same as `${Name}` or `${NAME}`.

Keep in mind the following rules:

- You can use interpolations only in text and in string literal expressions, described later in this chapter.

- You cannot place an RPL tag inside another RPL tag or inside an interpolation. For example, the following is incorrect:

```
<#-- Wrong code →
<#if <#include 'foo'>='bar'>...</#if>
```

- You can place comments inside RPL tags and interpolations. For example:

```
<h1>Welcome ${user <#-- The name of user -->}!</h1>[BR]
<p>We have these animals:[BR]
<ul>[BR]
<#list <#-- some comment... --> animals as <#-- again... -->
being>[BR]
...
```

Working with RPL tags

You use RPL tags to call directives. For example, the following code:

```
<#list animals as being>[BR]
    <li>${being.name} for ${being.price} Euros[BR]
  </#list>[BR]
```

calls the `list` directive.

There are two kind of RPL tags:

Start tag

<#directivename parameters>

End tag

</#directivename>

This is similar to HTML or XML syntax, except that the RPL tag name begins with #. If the directive does not have any content between the start and end tags, you must use the start tag with no end tag. For example, `<#if something>...</#if>`, but just `<#include something>`.

There are two types of directives: predefined directives and user-defined directives. User-defined directives begin with `@`, for example `<@mydirective parameters>`, and predefined directives begin with `#`. One further difference is that if the directive has no nested content, you must use a tag such as <@mydirective parameters />, similarly to XML (e.g. <img ... />). User-defined directives are described in more detail later in this chapter.

RPL tags must be properly nested. For example, the following code is incorrect because the `if` directive is both inside and outside of the nested content of the `list` directive:

```
<ul>
<#list animals as being>
  <li>${being.name} for ${being.price} Euros
  <#if user == "Big Joe">
     (except for you)
</#list> <#-- WRONG! The "if" has to be closed first. -->
</#if>
</ul>
```

Note that RPL does not care about the nesting of HTML tags because RPL reads HTML as flat text and does not interpret it in any way.

If you try to use a non-existing directive (e.g., you mistype the directive name), RPL will not process the template and produce an error message.

RPL ignores superfluous white space inside RPL tags. So you can write this:

```
<#list[BR]
  animals       as[BR]
     being[BR]
>[BR]
${being.name} for ${being.price} Euros[BR]
</#list    >
```

You may not, however, insert white space between the < or </ and the directive name.

For a complete list and descriptions of predefined directives, see *“Chapter 6. Directive Reference”*.

The template at a glance

The simplest template is an HTML or plain text file. When the system emails a user, RPL sends that HTML as is. However, to make the message to be more dynamic, you can include instructions for RPL in the HTML:

Interpolations	Interpolations contain instructions to convert an expression to text and to insert that text into the output. An interpolation begins with `${` and ends with `}`. Note that interpolations do not obtain a value, they only execute an expression.
RPL tags	RPL tags are a similar to HTML tags. The tags start with `#`. User-defined RPL tags use `@` instead of `#`. RPL tags refer to *directives*. You can think of "RPL tag" and "directive" as synonyms.
Comments	Comments are similar to HTML comments, but they are delimited by `<#--` and `-->`. Anything between these delimiters is not written to the output.

Everything in the template except interpolations, RPL tags, and comments is considered static text, and is printed to the output as is (with the exception of rules for space truncation, discussed later in this document).

For detailed information about working with templates, see “Working with Templates” later in this chapter.

Most commonly used directives

This quick overview describes the three most commonly used directives. For descriptions of all RPL directives, see *“Chapter 6. Directive Reference”*.

if

With the `if` directive, you can conditionally skip a section of the template. For example, assume that you want to greet the boss, Big Joe, differently from other users:

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <h1>
    Welcome ${profile.user}<#if profile.user == "Big Joe">, our
beloved leader</#if>!
  </h1>
  <p>Our latest product:
  <a href="${latestProduct.url}">${latestProduct.name}</a>!
</body>
</html>
```

The example above instructs RPL that the ", our beloved leader" should be used only if the value of the variable `profile.user` is equal to the string "Big Joe". In general, things between the *`<#if condition>`* and *`</#if>`* tags are skipped if the condition is false.

Let's look at the above example in detail:

== is an operator that tests whether the values at its left and right side are equivalent. The results is a boolean value `true` or `false`. On the left side of == we have referenced a variable with the syntax that should be already familiar; this will be replaced with the value of the variable. In general, unquoted words inside directives or interpolations are treated as references to variables. On the right side of ==, we specified a literal string. Literal strings in templates must always be enclosed in quotation marks.

IMPORTANT: In the example, `${profile.user}` is used outside of the `if` directive, and `profile.user` inside of the `if` directive. `${user}` is needed since we are in the HTML context and the `${…}` construct indicates the beginning of RPL instructions. When inside the `if` statement, we are already inside an RPL context; therefore, RPL will treat the code as instructions and the `${…}` is not needed.

Assuming the price of pythons is 0, the following example:

```
<#if animals.python.price == 0>
  Pythons are free today!
</#if>
```

produces this output:

```
Pythons are free today!
```

In this example, a number is specified directly. Note that the number is not enclosed in quotes; if it was, RPL would misinterpret it as a string.

To check whether the values are not equivalent, use the != operator. For example, if the price of pythons is not 0, the following code:

```
<#if animals.python.price != 0>
  Pythons are not free today!
</#if>
```

produces this output:

```
Pythons are not free today!
```

You can use the < operator to compare values. For example:

```
<#if animals.python.price < animals.elephant.price>
  Pythons are cheaper than elephants today.
</#if>
```

You can use the `<#else>` tag to specify what to do if the condition is false. For example, the following code:

```
<#if animals.python.price < animals.elephant.price>
```

```
  Pythons are cheaper than elephants today.
<#else>
  Pythons are not cheaper than elephants today.
</#if>
```

If the price of python is less than the price of elephant, the example will print `Pythons are cheaper than elephants today`. Otherwise, it will print `Pythons are not cheaper than elephants today.`

If a variable has a boolean value (true/false), you can use it directly as the condition of `if`:

```
<#if animals.python.protected>
  Warning! Pythons are protected animals!
</#if>
```

list

The `list` directive is useful when you want to list something. For example, merging this template with the data model we used earlier to demonstrate sequences:

```
<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  <#list animals as being>
  <tr><td>${being.name}<td>${being.price} Euros
  </#list>
</table>
```

produces this output:

```
<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  <tr><td>mouse<td>50 Euros
  <tr><td>elephant<td>5000 Euros
  <tr><td>python<td>4999 Euros
</table>
```

The generic format of the `list` directive is:

```
<#list sequence as loopVariable>repeatThis</#list>
```

The `repeatThis` part will be repeated for each item in the sequence, one after the other, starting from the first item. In all repetitions `loopVariable` will hold the value of the current item. This variable exists only between the `<#list ...>` and `</#list>` tags.

As another example, we list the fruits of the example data model:

```
<p>And BTW we have these fruits:
<ul>
<#list whatnot.fruits as fruit>
 <li>${fruit}
</#list>
<ul>
```

The `whatnot.fruits` expression references a variable in the data model.

include

You can use the `include` directive to insert the content of another file into the template.

For example, you have to show the same copyright notice on several pages. You can create a file that contains only the copyright notice, and insert that file anywhere you need the copyright notice.

The copyright file shown below is called *copyright_footer.html* and it is stored in *cms://contentlibrary/common:*

```
<hr>
<i>
Copyright (c) 2000 <a href="http://www.acmee.com">Acmee Inc</a>,
<br>
All Rights Reserved.
</i>
```

use the following code to include the copyright notice in your template:

```
<html>
<head>
  <title>Test page</title>
</head>
<body>
  <h1>Test page</h1>
  <p>Content...
```

```
<#include "cms://contentlibrary/common/copyright_footer.html">
</body>
</html>
```

and the output will be:

```
<html>
<head>
  <title>Test page</title>
</head>
<body>
  <h1>Test page</h1>
  <p>Content...
<hr>
<i>
Copyright (c) 2000 <a href="http://www.acmee.com">Acmee Inc</a>,
<br>
All Rights Reserved.
</i>
</body>
</html>
```

If you change the copyright_footer.htm , the new copyright notice will appear on all pages produced with this template.

For more information about the `include` directive, see *“Chapter 6. Directive Reference”*.

data

Use the `data` directive to obtain data from additional tables by querying the database.

Usually, personalization data is automatically added to the template. However, data from additional tables used for personalization is not automatically included, since obtaining it requires additional criteria. Before using the `data` directive, you need to set up datasources in the system by including them in the campaign. When you include a datasource into a campaign, you need to give aliases for the datasource and its columns. You also need to identify which columns are used as lookup keys (i.e. the columns used to specify the criteria for the data to be returned). Interact is declarative in this respect. RPL uses only the aliases, and performs a full check as follows:

- If a field is used as a lookup key in the `filter` section of the `data` directive, RPL checks the definition of the campaign to ensure that the field was declared as such. Failure to do that will result in an error.
- If a field is used as a returned field in the `fields` section, RPL checks the definition of the campaign to ensure that the field was declared in the campaign with the given alias.

The examples in this section assume that you have created the aliases.

Example

To obtain data, you specify a data element with its three parts:

the `data` specification, the `filter` specification, and the returned `fields` specification as shown in the following example:

```
These items are on sale right now!
<table>
<tr><th>Item</th><th>Total</th></tr>
<#data orders as order>
  <#filter custid=profile.custId>
  <#fields orderId product quantity unitPrice >
    <tr id="sale${order.orderId}">
      <td>${order.product}</td>
      <td>${order.unitPrice * order.quantity}</td>
    </tr>
</#data>
</table>
```

The sections in the example are as follows:

The data declaration section (highlighted in blue in the example)	Specifies the structural nature of the request for data, including the datasource being used, the filter, and the fields being returned.
The declaration of the looping variable (shown in **purple** in the example)	Specifies the namespace hash being created, with the requested fields, as the source for each one of the records returned from the inquiry. The looping section can refer to this namespace to obtain the required fields, as shown in the section highlighted in yellow.
The looping section (highlighted in yellow in the example)	This section is repeated for each record obtained. This section uses the namespace and the fields as specified in the `fields` declaration section. These

	are interpolations (`${…}`, but they could also be used in any other valid expression. The `offer` namespace hash is updated once for each iteration, with the data as present in each record from the given datasource.

The following table shows a database table called SALES, with the fields by which the table is indexed marked with an *.

ID*	CUSTID*	PRODUCT*	QUANTITY	UNIT_PRICE
1	0001	Paylesss Shoes – Size 13	1	25.99
2	0001	Gucci Sunglasses	2	229.99
3	0002	Coach Handbag	1	349.99
4	0003	Coach Handbag	2	349.99
5	0003	Giants Classic Cap	3	45.99

You can set up the datasources as follows:

Database/Field	Alias	Lookup
SALES database	orders	
ID field	orderId	No
CUSTID field	custid	No
PRODUCT field	product	Yes
QUANTITY field	quantity	No
UNIT_PRICE field	unitPrice	No

If the profile.custId is "0001", the result of the previous example will be:

```
<table>
<tr><th>Item</th><th>Total</th></tr>
```

```
    <tr id="sale1">
      <td>Payless Shoes - Size 13</td>
      <td>25.99</td>
    </tr>
    <tr id="sale1">
      <td> Gucci Sunglasses </td>
      <td>459.98</td>
    </tr>
</table>
```

content

NOTE: To use the `content` directive, your account needs to be configured for its use. Please request access from your account representative.

In most cases, you add the contents of the message directly as HTML in the template. However, sometimes you need to create content dynamically, based on conditions in the recipient record. For instance, you might want to insert products based on location, or offers in nearby stores.

This type of content is usually defined by specific schemas that provide a structure that breaks down the content into consumable pieces and meta data (e.g. product name, description, images, price, size, etc.) The `content` directive is used for such situations.

While the `data` directive is useful for content has a lot of variance, the `content` directive is more suitable for content that repeats across multiple recipients. For example, if you launch a campaign with a million recipients, the `data` directive results in a million queries for the content. However, the content might have only 100 variations. The `content` directive is useful in this example as it automatically caches the results, resulting in 100 queries. In addition, the content directive allows for greater-than/less-than semantics useful for dates.

To use the `content` directive, the content must be loaded into Oracle Responsys in supplemental tables. After the content is uploaded and available in the system, a data source must be declared in the Campaign Workbook. You should declare the datasource with a unique alias that will be used to refer to the content in the `content` directive. Each field that will be used in a filter query should also be declared and aliased.

In addition to the efficiency on low variants, the content directive helps you with image paths. If your assets are hosted in the Oracle Responsys Content Library and you include the path to that asset in your content structure (with "/contentlibrary/" as the root folder), Oracle Responsys will automatically replace the image path and make the asset available on the content delivery network so it can be accessed from the external internet at launch.

For example, the following database of events across the United States. The data will be stored in a supplemental table called US_EVENTS.

EVENT_ID	DESCRIPTION	CATEGORY	VENUE_ NAME	REGION	DATE_ LOCAL
1	Eagles in concert	CONCERT	HP Pavillion	Bay Area	4/10/16 0:00
2	Knicks vs. Golden State	NBA	Oracle Arena	Bay Area	9/14/16 0:00
3	The Wizard of Oz	THEATRE	Eugene O'Neill Theatre	New York	5/30/16 0:00
4	Red Wings vs. Canucks	NHL	Rogers Arena	Toronto	8/10/16 0:00
5	Lakers vs. Golden State	NBA	Oracle Arena	Bay Area	5/5/16 0:00
6	Trailblazers vs. Kings	NBA	Sleep Train Arena	Bay Area	6/17/16 0:00
7	Cardinals vs. Giants	MLB	AT&T Park	Bay Area	6/1/16 18:00
8	Angels vs. As	MLB	Oracle Coliseum	Bay Area	6/3/16 19:30
9	Cardinals vs. Braves	MLB	Busch Stadium	Central	11/3/16 0:00

And the datasource is declared as follows:

Database/Field	Alias	Lookup
US_EVENTS	Events	
DESCRIPTION	DESCR	
CATEGORY	CATEGORY	Yes
VENUE_NAME	VENUE	
REGION	EVENT_REGION	Yes
DATE_LOCAL	DATE	Yes

The `content` directive can then be used to retrieve content as follows:

```
<#assign today=.today>
<#assign next_week=dayadd(today,7)>
<table>
<tr><th>Description</th><th>Venue</th><th>Date</th></tr>
<#content Events as event>
  <#filter CATEGORY="MLB" && EVENT_REGION=PROFILE.REGION && DATE gte
today && DATE lt next_week>
<tr><td>${event.DESCR}</td><td>${event.VENUE}</td><td>${event.DATE}<
/td></tr>
</#content >
</table>
```

The result of the previous example produces at most 3 events in the user's region.

NOTES: If paths are used in the content table with prefixes that begin with the string /contentlibrary/, they will be translated to the externally available URL (CDN) path.

Note the use of the special variable `.today` in the example. This produces the time of today at midnight. If you use `.now` instead, the query will not be cached as each invocation of `.now` results in a different date and time.

As shown in the example, the content directive is composed of three parts:

The content declaration (highlighted in blue in the example)	This section is defined by the `<#content` directive and its options, and the `<#filter` sub-directive. It defines the data source to use and the filter to run.
The declaration of the namespace for the loop (shown in purple in the example)	This declaration identifies the hash that will be used to return the data from each row of the content. Note that in addition of the hash, two other fields are defined: ***ns*_has_next** Boolean value, defines if there are more rows in the namespace. ***ns*_index** Numeric value, identifies the row number for each content. The name of these additional variables is composed of the namespace (shown as *ns* above) indicated in the `<#content` directive (*event* in the example), with the suffixes indicated. In the example, the variables will be `event_has_next and event_index`.
The loop (highlighted in yellow in the example).	This loop is executed once for every row. If the loop encounters a `break` directive, the looping will end.

The example, based on the filter criteria, produces the following output:

Cardinals vs. Giants	MLB	AT&T Park	Bay Area	6/1/16 18:00
Angels vs. As	MLB	Oracle Coliseum	Bay Area	6/3/16 19:30

```
<table>
<tr><th>Description</th><th>Venue</th><th>Date</th></tr>
<tr><td>Cardinals vs. Giants</td><td>AT&T Park</td><td>6/1/16
18:00:00</td></tr>
<tr><td>Angels vs. As</td><td>Oracle Coliseum</td><td>6/3/16
19:30:00</td></tr>
</table>
```

Using directives together

You can use directives as many times on a page as you want, and you can nest directives similarly as you can nest HTML elements. For example, the following code will list the animals and print the name of large animals in a bigger font:

```
<p>We have these animals:
<table border=1>
  <tr><th>Name<th>Price
  <#list animals as being>
  <tr>
    <td>
      <#if being.size == "large"><font size="+1"></#if>
      ${being.name}
      <#if being.size == "large"></font></#if>
    <td>${being.price} Euros
  </#list>
</table>
```

Note that RPL does not interpret text outside RPL tags, interpolations and comments. Therefore, it does not interpret the font tags in the example as badly nested ones.

Handling missing variables

The data model often has variables that are optional (i.e., sometimes missing). RPL does not support referring to missing variables unless you explicitly specify what to do if a variable is missing.

This section describes the two most common ways of handling missing variables.

NOTE: RPL treats both non-existent variables and a variables with null values as missing.

Specifying a default value

When referring to a variable that might be missing, you can specify a default value. To do this, follow the variable name with an exclamation mark (`!`) and the default value.

In the following example, when `user` is missing, “Anonymous” will be used as the value. When `user` is not missing, the default value is ignored:

```
<h1>Welcome ${profile.user!"Anonymous"}!</h1>
```

Please note that missing field names at the root namespace will result in empty strings, and they will appear to have a value. For instance:

```
<h1>Welcome ${FIRRST_NAME!”Unknown”}!</h1>
```

(Notice the double ‘R’)

Will never result in Unknown since the variable is assumed to have a value.

Specifying a default value for sub-variables

To specify a default value for a sub-variable, follow the sub-variable name with a ! as shown in the following example:

```
<h1>Welcome ${user.firstname!"Anonymous"}!</h1>
```

Note that the above example is correct if only `firstname` is missing but `user` is not. If `user` is missing as well, an error will occur that terminates template processing. To avoid such errors, enclose both the top-level variable and the sub-variable in parenthesis:

```
<h1>Welcome ${(user.firstname)!"Anonymous"}!</h1>
```

Testing for missing values

To find out whether a value is missing, follow the variable name with double question marks (`??`).

As shown in the following example, you can combine `??` with the `if` directive to skip the greeting if the `user` variable is missing a value:

```
<#if profile.user??><h1>Welcome ${user}!</h1></#if>
```

Please note that in the prior examples, the use of `profile.user` will be missing only if they were not aliased on the datasources in the campaign design. If they were aliased, the result is that the variable is present. If want to check whether a given value was a null or empry string, or a null in the case of numbers and dates, you can use the `?isnull builtin` (explained in detail later.) Briefly, `?isnull` is usually used in `if` directives. The following is an example, with the erroneous FIRRST_NAME of previous examples:

```
<h1>Welcome <#if
FIRRST_NAME?isnull>Unknown<#else>${FIRRST_NAME}</#if>!</h1>
```

The builtin `?isnull` is usually applied towards empty strings, and to numbers and dates coming from the personalization record. You cannot create null dates or numbers in RPL other than from the personalization record.

Important: Regarding variable accessing with multiple steps, such as `animals.python.price`, writing `animals.python.price!0` is correct only if `animals.python` is never missing and only the last sub-variable, `price`, is possibly missing (in which case here we assume it's 0). If `animals` or `python` is missing, the template processing will stop with an "undefined variable" error. To prevent this, you have to write `(animals.python.price)!0`. In that case the expression will be 0 even if `animals` or `python` is missing. The same logic applies to `??; animals.python.price??` versus `(animals.python.price)??`.

Testing for missing sub-variables

To test whether a value is missing in a sub-variable, follow the sub-variable name with double question marks (`??`). For example:

```
<#if user.firstname??><h1>Welcome ${user}!</h1></#if>
```

Note that the above example is correct if only the value of `firstname` is missing, but not of `user`. If `user` is missing as well, an error will occur that terminates template processing. To avoid such errors, enclose both the top-level variable and the sub-variable in parenthesis:

```
<#if (user.firstname)??><h1>Welcome ${user}!</h1></#if>
```

Working with expressions

When you supply values for interpolations or directive parameters you can use variables or more complex expressions. For example, if x is the number 8 and y is 5, the value of (x + y)/2 resolves to the numerical value 6.5.

In interpolations, use `${expression}` where expression gives the value you want to insert into the output as text. So ${(5 + 8)/2} prints “6.5”.

In directives, use `<#directivename expession>`, for example `<#if expression>...</#if>`.

Quick overview (cheat sheet)

If you already know RPL or are an experienced programmer, you can use this section as a quick reference.

Specifying values directly

Type	Example
String	"Foo" or 'Foo' or "It's \"quoted\"" or r"C:\raw\string"
Number	123.45
Boolean	true, false
Sequence	["foo", "bar", 123.45], 1..100
Hash	{"name":"green mouse", "price":150}

Retrieving variables

Type	Example
Top-level variable	user
Retrieving data from a hash	user.name, user["name"]
Retrieving data from a sequence	products[5]
Special variables	main

String operations

Operation	Example
Interpolation or concatenation	"Hello ${user}!" (or "Inter" + "act")
Getting a character	name[0]

Sequence operations

Operation	Example
Concatenation	users + ["guest"]
Sequence slice	products[10..19] or products[5..]

Hash operations

Operation	Example
Concatenation	passwords + {"joe":"secret42"}

Numeric/Boolean expressions

Expression Type	Example
Arithmetical calculations	(x * 1.5 + 10) / 2 - y % 100
Comparison	x == y, x != y, x < y, x > y, x >= y, x <= y, x &lt; y, ...etc.

Expression Type	Example
Logical operations	!registered && (firstVisit || fromEurope)
Built-ins	name?upper_case
Method call	repeat("What", 3)

Missing value handler operators

Action	Example
Default value	name!"unknown" or (user.name)!"unknown" or name! or (user.name)!
Missing value test	name?? or (user.name)??

Specifying values directly

This section describes how to specify values directly instead of as a result of calculations.

Specifying strings

To specify a string value directly, enclose the text in single or double quotation marks, for example `"some text"` or `'some text'`. The two forms are equivalent. If the text contains the character used for the quoting (either " or ') or backslashes (/), you have to precede that character with a backslash; this is called *escaping*. You can type any other character, including line breaks, in the text directly. For example:

```
${"It's \"quoted\" and
this is a backslash: \\"}

${'It\'s "quoted" and
this is a backslash: \\'}
```

produces this output:

```
It's "quoted" and
this is a backslash: \

It's "quoted" and
this is a backslash: \
```

NOTE: Alternately, you can simply type the above text into the template without using ${...}.

The following table shows the list of all supported escape sequences. All other usage of a backlash in string literals is an error and any attempt to use the template will fail.

Escape sequence	Meaning
\″	Quotation mark (u0022)
\′	Apostrophe (a.k.a. apostrophe-quote) (u0027)
\\\\	Back slash (u005C)
\n	Line feed (u000A)
\r	Carriage return (u000D)
\t	Horizontal tabulation (a.k.a. tab) (u0009)
\b	Backspace (u0008)
\f	Form feed (u000C)
\l	Less-than sign: <
\g	Greater-than sign: >
\a	Ampersand: &
\xCode	Character given with its hexadecimal Unicode code (UCS code)*

*The Code after the \x is 1 to 4 hexadecimal digits. For example this will put a copyright sign into the string: ″\xA9 1999-2001″, ″\x0A9 1999-2001″, ″\x00A9 1999-2001″. When the character directly after the last hexadecimal digit can be interpreted as hexadecimal digit, you must use all 4 digits.

Note that the character sequence `${` (and `#{`) has special meaning. It is used to insert the value of expressions (typically, the value of variables, as in ″Hello ${user}!″). If you want to print ${, you should use raw string literals as explained below. You can also use the sequence \x0024 to replace the $ sign. For example, the following produces the string ${ABC} in the output: ″\x0024{ABC}”.

About raw string literals

Raw string literals are a special kind of string literal. In raw string literals, the backslash and `${` have no special meaning, they are considered plain characters. To indicate that a string literal is a raw string literal, put an `r` directly before the opening quotation mark. For example:

```
${r"${foo}"}
${r"C:\foo\bar"}
```

produces this output:

```
${foo}
C:\foo\bar
```

Specifying numbers

To specify a numerical value directly, type the number without quotation marks. Use the dot as the decimal separator and do not use any grouping separator symbols. You can use - or + to indicate the sign (+ is redundant). RPL does not support scientific notation (so 1E3 is incorrect). In addition, you cannot omit the 0 before the decimal separator (so .5 is incorrect).

Examples of valid number literals are: 0.08, -5.013, 8, 008, 11, +11

Note that numerical literals such as 08, +8, 8.00 and 8 are equivalent as they all symbolize the number eight. Thus, ${08}, ${+8}, ${8.00} and ${8} will all produce the same output.

Specifying booleans

To specify a boolean value, write `true` or `false` without quotation marks.

Specifying sequences

To specify a literal sequence, list the sub-variables separated by commas, and enclose the entire list in square brackets ([]). For example:

```
<#list ["winter", "spring", "summer", "autumn"] as x>
${x}
</#list>
```

produces this output:

```
winter
spring
summer
autumn
```

The items in the list are expressions, so the following is correct:

```
[2 + 2, [1, 2, 3, 4], "whatnot"]
```

In the above example, the first sub-variable is the number 4, the second is another sequence, and the third sub-variable is the string "whatnot".

You can define sequences that store a numerical range with start..end, where start and end are expressions that resolve to numerical values. For example `2..5` (without the square brackets) is the same as `[2, 3, 4, 5]`, but the former is much more efficient as it occupies less memory and is faster. You can define decreasing numerical ranges too, for example, 5..2. Furthermore, you can omit the end, for example 5.., in which case the sequence will contain 5, 6, 7, 8, ...etc. up to the infinity.
WARNING: Using the <#list> directive on infinite sequences can cause an infinite loop that will cause the template execution not to terminate.

Specifying hashes

To specify a hash in a template, list the key/value pairs separated by commas, and enclose the list in curly brackets ({}). The key and value within a key/value pair are separated by a colon. For example:

```
{"name":"green mouse", "price":150}.
```

Note that both names and values are expressions. However, lookup names must be strings.

Retrieving variables

Retrieving top-level variables

To access a top-level variable, use the variable name. For example, the expression `user` will evaluate to the value of variable stored with name “user” in the root. So the following example prints the value stored in that variable:

```
${user}
```

A top-level variable can be created in multiple ways. Its most important use is to retrieve fields from a personalization record.

If there is no such top-level variable defined in the personalization record or assigned with `<#assign>`, RPL will return an empty string.

In this expression, the variable name can contain only letters (including non-Latin letters), digits (including non-Latin digits), underline (_), dollar sign ($), at sign (@) and hash (#). Furthermore, the name must not start with digit.

Personalization fields and campaign variables appear as top level variables in RPL. However, you can create additional higher level variables with the `assign`, `local` and `global` directives.

Retrieving data from a hash

If you have a hash as a result of an expression, you can get its sub-variable with a dot and the name of the sub-variable. Assume that you have this data model:

```
(root)
 |
 +- book
 |   |
 |   +- title = "Breeding green mice"
 |   |
 |   +- author
 |       |
 |       +- name = "Julia Smith"
 |       |
 |       +- info = "Biologist, 1923-1985, Canada"
 |
 +- test = "title"
```

You can read the title with `book.title`, since the book expression will return a hash. Applying this logic further, you can read the name of the author with this expression: `book.author.name`.

Alternately, you can specify the sub-variable name with an expression: `book["title"]`. You can specify any expression that evaluates to a string in the square brackets, for example `book[test]` .

The following examples are equivalent:

```
${book.author.name}
${book["author"].name}
${book.author.["name"]}
${book["author"]["name"]}
```

When you use the dot syntax, the same restrictions apply regarding the variable name as with top-level variables (name can contain only letters, digits, _, $, @, etc.). There are no such restrictions when you use the square bracket syntax, since the name is the result of an arbitrary expression. Note that if the sub-variable name is * (asterisk) or **, you do not have to use square bracket syntax.

Unlike top level variables, undefined variables within a hash do not return the empty string. Rather, if an attempt is made at retrieving a variable that does not exist, an error will result and RPL will terminate further processing, resulting in a launch failure. You can avoid susch failures with the use of the exclamation point operator (`!`), the double question mark operator (`??`), or `<#attempt>` and `<#recover>`.

Retrieving data from a sequence

This is the same as for hashes, but you can use only the square bracket syntax, and the expression in the brackets must evaluate to a number, not a string. For example, to get the name of the first animal of the example data model, use `animals[0].name`.

Retrieving special variables

Normally you do not need to use special variables.

Special variables are defined by the RPL. To access them, you use the `.variable_name` syntax. Special variables are described in *"Chapter 9. Special Variables"*.

String operations

Interpolation or concatenation

To insert the value of an expression into a string, you can use `${...}` and `#{...}` in string literals. ${...} behaves similarly as in text sections. For example, assuming that user is "Big Joe":

```
${"Hello ${user}!"}
${"${user}${user}${user}${user}"}
```

produces this output:

```
Hello Big Joe!
Big JoeBig JoeBig JoeBig Joe
```

Alternatively, you can use the + operator to achieve the same result. This method is called *string concatenation*. It is the preferred method. For example:

```
${"Hello " + user + "!"}
${user + user + user + user}
```

This will print the same as the example with the ${...}.

IMPORTANT: Interpolations work only in text sections (e.g. <h1>Hello ${name}!</h1>) and in string literals (e.g. <#include "/footer/${company}.html">). A common mistake is using `<#if ${isBig}>Wow!</#if>`, which is syntactically incorrect. Instead, write `<#if isBig>Wow!</#if>`. `<#if "${isBig}">Wow!</#if>` is incorrect as well, since the parameter value will be a string, and the `if` directive requires a boolean value, so it will cause a runtime error.

Getting a character

You can get a single character of a string at a given index in the same way as you read the sub-variable of a sequence, e.g. `user[0]`. The result is a string whose length is 1; RPL does not have a separate character type. As with sequence sub-variables, the index must be a number that is at least 0 and less than the length of the string. Otherwise, an error will terminate template processing.

Since the sequence sub-variable syntax and the character getter syntax clashes, you can use the character syntax only if the variable is not a sequence as well, since in that case the sequence behavior prevails. As a workaround, you can use the string built-in, e.g. `user?string[0]`.

For example, assuming that `user` is “Big Joe”:

```
${user[0]}
${user[2]}
```

produces this output (note that the index of the first character is 0):

```
B
g
```

Sequence operations

Concatenation

You can concatenate sequences in the same way as strings, with +. For example:

```
<#list ["Joe", "Fred"] + ["Julia", "Kate"] as user>
- ${user}
</#list>
```

produces this output:

```
- Joe
- Fred
- Julia
- Kate
```

NOTE: Do not use sequence concatenation for many repeated concatenations, such as for appending items to a sequence inside a loop. Use it for things such as <#list users + admins as person>. Although concatenating sequences is fast and its speed is independent of the size of the concatenated sequences, the resulting sequence will always be a little slower to read then the original two sequences. This means that the result of many repeated concatenations is a sequence that is slow to read.

Sequence slice

With [`firstindex..lastindex`] you can get a slice of a sequence, where `firstindex` and `lastindex` are expressions that evaluate to number. For example, if `seq` stores the sequence "a", "b", "c", "d", "e", "f" then the expression *seq[1..4]* will evaluate to a sequence that contains "b", "c", "d", "e" (since the item at index 1 is "b", and the item at index 4 is "e").

The lastindex can be omitted, in which case it defaults to the index of the last item of the sequence. For example, if `seq` stores the sequence "a", "b", "c", "d", "e", "f" again, then seq[3..] will evaluate to a sequence that contains "d", "e", "f".

An attempt to access a sub-variable past the last sub-variable or before the first sub-variable of the sequence will cause an error and terminate template processing.

Hash operations

Concatenation

You can concatenate hashes in the same way as strings, with +. If both hashes contain the same key, the hash on the right-hand side of the + takes precedence. Example:

```
<#assign ages = {"Joe":23, "Fred":25} + {"Joe":30, "Julia":18}>
- Joe is ${ages.Joe}
- Fred is ${ages.Fred}
- Julia is ${ages.Julia}
```

produces this output:

```
- Joe is 30
- Fred is 25
- Julia is 18
```

NOTE: Do not use hash concatenation for many repeated concatenations, such as adding items to a hash inside a loop. It's the same as with the sequence concatenation.

Arithmetical calculations

RPL supports the following arithmetic functions:

Addition: +

Subtraction: -

Multiplication: *

Division: /

Modulus (remainder): %

Operator precedence is defined in *"Appendix B. Operator Precedence"*.

Example:

```
${100 - x * x}
${x / 2}
${12 % 10}
```

assuming that x is 5, the example produces this output:

```
75
2.5
2
```

Both operands must be expressions which evaluate to a numerical value. The example below will cause an error, since "5" is a string and not the number 5:

```
${3 * "5"} <#-- WRONG! -->
```

There is an exception to the above rule. The + operator is used to concatenate strings as well as numbers. As a result, if a string is on one side of + and a numerical value on the other side, the numerical value will be converted to a string (using the format appropriate for the language of the page) and then use the + as a string concatenation operator. For example: `${3 + "5"}` will output 35.

Generally, RPL does not convert strings to numbers.

If you want only the integer part of the result of a calculation, you can use the `int` built-in. For example:

```
${(x/2)?int}
${1.1?int}
${1.999?int}
${-1.1?int}
${-1.999?int}
```

Assuming that x is 5, the example produces this output:

```
2
1
1
-1
-1
```

Comparison

This section describes how to compare two values, using examples of the `if` directive.

The usage of the `if` directive is: `<#if expression>...</#if>`, where `expression` must evaluate to a boolean value; otherwise, an error will terminate template processing. If the value of expression is true, the code between the begin and end tags will be processed; otherwise it will be skipped.

To test two values for equality, use the = operator (or == as in Java or C).

To test two values for inequality, use the != operator. For example, assume that user is “Big Joe”:

```
<#if user = "Big Joe">
  It is Big Joe
</#if>
<#if user != "Big Joe">
  It is not Big Joe
</#if>
```

The `user = "Big Joe"` expression evaluates to the boolean true, so the output will be:

```
It is Big Joe
```

The expressions on both sides of the = or != must evaluate to a scalar. Furthermore, the two scalars must be of same type (i.e. strings can only be compared to strings and numbers can only be compared to numbers). Otherwise, an error will terminate template processing. Note that RPL does exact comparison, so string comparisons are case and white space sensitive. This means that "x", "x " and "X" are not equal values.

For numerical and date values, you can also use <, <=, >= and >. You cannot use these operators for strings.

Example:

```
<#if x <= 12>
  x is less or equivalent to 12
</#if>
```

Keep in mind that RPL interprets `>` as the closing character of an RPL tag. For this reason, you must enclose the expression in parentheses, for example `<#if (x > y)>`. Alternately, you can use &gt; and &lt;, for example`<#if x &gt; y>`. Note that entity references such as &gt; &lt; are only supported in arithmetical comparisons, they are not supported in other RPL tags. As another alternative, you can use `lt` instead of `<`, `lte` instead of `<=`, `gt` instead of `>` and `gte` instead of `>=`.

Logical operations

RPL supports the following standard logical operators:

Logical or: **||**

Logical and: **&&**

Logical not: **!**

These operators work only with boolean values. Otherwise, an error will terminate template processing.

Example:

```
<#if x < 12 && color = "green">
  We have less than 12 things, and they are green.
</#if>
<#if !hot> <#-- here hot must be a boolean -->
  It's not hot.
</#if>
```

About built-ins

Built-ins provide certain functionality that is always available. Typically, a built-in provides a different version of a variable, or information about the variable in question. The syntax for accessing a built-in is the same as for accessing a sub-variable in a hash, except that you use the question mark instead of a dot. For example, to get the upper case version of a string: `user?upper_case`.

The following section lists the most commonly used built-ins. You can find a complete list of built-ins in *"Chapter 5. Built-in Reference"*.

Built-ins to use with strings

Built-in Name	Description
html	The string with all special HTML characters replaced by entity references (E.g. < with &lt;)
cap_first	The string with the first letter converted to upper case
lower_case	The lowercase version of the string
upper_case	The uppercase version of the string
trim	The string without leading and trailing white spaces

Built-ins to use with sequences

Built-in Name	Description
size	The number of elements in the sequence

Built-ins to use with numbers

Built-in Name	Description
int	The integer part of a number (e.g. -1.9?int is -1)

Example:

```
${test?html}
${test?upper_case?html}
```

Assuming that `test` stores the string “Tom & Jerry”", the output will be:

```
Tom &amp; Jerry
TOM &amp; JERRY
```

Note the `test?upper_case?html`. Since the result of `test?upper_case` is a string, you can use the `html` built-in with it.

Another example:

```
${seasons?size}
${seasons[1]?cap_first} <#-- left side can by any expression -->
${"horse"?cap_first}
```

Assuming that `seasons` stores the sequence "winter", "spring", "summer", "autumn", the output will be:

```
4
Spring
Horse
```

Calling methods

You call methods using a comma-separated list of expressions in parentheses. These expressions are called *parameters*. The method call passes the parameters to the method, which returns a result. The result is the value of the method call.

For example, assume the method called `avg` returns an average of its parameters, 3 and 5 in the following example:

```
${avg(3, 5)}
```

produces this output:

```
4
```

Handling missing values

If you try to access a missing variable, an error will occur that terminates template processing. To handle such cases, you can use two special operators: the default value operator and the missing value test operator. These operators handles missing variables, as well as cases when a method call does not return a value. The missing variable can be top-level variable, hash sub-variable, or sequence sub-variable.

NOTE: RPL treats null values as missing values. For example, if the value of a field property is null, RPL treats it the same way as if that property did not exist. As a result, a method call that returns null is treated as a missing variable.

Using the default value operator

```
unsafe_expr!default_expr
or
unsafe_expr!
or(unsafe_expr)!default_expr
or
(unsafe_expr)!
```

The default value operator `!` allows you to specify a default value when the value is missing.

Example:

```
${xml.animals.mouse!"No mouse."}
```

Assuming no variable called `xml.animals.mouse` exists, produces this output:

```
No mouse.
```

IMPORTANT: Undefined fields in the root namespace return an empty string. As such, they will not return the default expression value.

The default value can be any kind of expression, it does not have to be a string. For example, you can write `xml.hits!0` or `xml.colors!["red", "green", "blue"]`. There is no restriction regarding the complexity of the expression that specifies the default value, for example you can write: `cargo.weight!(item.weight * itemCount + 10)`.

WARINIG: If you have a composite expression after the !, such as `1 + x`, always use parentheses, for example `${x!(1 + y)} or ${(x!1) + y)}`, depending on which interpretation you meant. This is because the precedence of ! (when used as the default value operator) is very low on the right of the expression. This means that, for example, RPL interprets `${x!1 + y}` as `${x!(1 + y)}` when it should mean `${(x!1) + y}`.

If the default value is omitted, it will be an empty string, and an empty sequence, and an empty hash at the same time. (This is possible because RPL allows multi-type values.) This means that you cannot omit the default value if you want it to be 0 or false. For example:

```
(${xml.animals.mouse!})
```

produces this output:

```
()
```

WARNING: Due to syntactical ambiguities, `<@something a=x! b=y />` will be interpreted as `<@something a=x!(b=y) />`, that is, the b=y will be interpreted as a comparison that gives the default value for x, rather than the specification of the b parameter. To prevent this, write: `<@something a=(x!) b=y />`.

Using the! operator with sub-variables

The following example:

```
product.color!"red"
```

Handles the case when the sub-variable `color` is missing, but not when the top-level variable `product` is missing. That is, the `product` variable must exist, otherwise template processing will terminate with an error. To avoid this situation, enclose the variable name in parentheses as shown in the following example:

```
(product.color)!"red"
```

This will handle the case when `product.color` is missing. In this case, if `product` is missing or `product` exists but it does not contain `color`, the result will be "red", and no error will occur.

When enclosed in parentheses, any component of the expression may be undefined, while without the parentheses only the last component of the expression may be undefined.

You can use the default value operator with sequence sub-variables as well:

```
<#assign seq = ['a', 'b']>
${seq[0]!'-'}
${seq[1]!'-'}
${seq[2]!'-'}
${seq[3]!'-'}
```

produces this output:

```
a
b
-
-
```

NOTE: A negative sequence index (as seq[-1]!'-') will always cause an error.

Using the missing value test operator

```
unsafe_expr??
or
(unsafe_expr)??
```

The missing value test operator `??` determines whether a value is missing and returns either true or false.

For example:

```
<#if xml.animals.mouse??>
  Mouse found
<#else>
  No mouse found
</#if>
```

Assuming the variable `mouse` is not present, produces this output:

```
  No mouse found
```

For sub-variables, the rules are the same as with the default value operator: enclose both the top-level and sub-variable in parentheses for example `(product.color)??`.

Grouping expressions

You can use parentheses to group any expressions. Some examples:

```
                                   <#-- Output will be: -->
${3 * 2 + 2}                       <#-- 8 -->
${3 * (2 + 2)}                     <#-- 12 -->
${3 * ((2 + 2) * (1 / 2))}         <#-- 6 -->
${"green " + "mouse"?upper_case}   <#-- green MOUSE -->
${("green " + "mouse")?upper_case} <#-- GREEN MOUSE -->
<#if !( color = "red" || color = "green")>
  The color is nor red nor green
</#if>
```

White space in expressions

RPL ignores superfluous white space in expressions. This means that the following are all equivalent:

```
${x + ":" + book.title?upper_case}
```

```
${x+":"+book.title?upper_case}
```

```
${
   x
 + ":"   +  book    .   title
   ?   upper_case
      }
```

Operator precedence

RPL operator precedence is similar to that of other programming languages, except RPL defines additional operators which are unique to RPL.

For details about operator precedence, see *"Appendix B. Operator Precedence"*.

Working with interpolations

The format of interpolations is `${expression}`, where `expression` can be any kind of expression (e.g. `${100 + x}`).

The interpolation is used to insert the value of the expression converted to a string. Interpolations can be used only in two places: in text sections (e.g., <h1>Hello ${name}!</h1>) and in string literal expressions (e.g., <#include "/footer/${company}.html">).

WARNING: A common mistake is using `<#if ${isBig}>Wow!</#if>`, which is syntactically incorrect. Instead, write `<#if isBig>Wow!</#if>`.

`<#if "${isBig}">Wow!</#if>` is incorrect as well, since the parameter value will be a string, and the `if` directive requires a boolean value.

The result of the expression must be a string, number or date value. This is because interpolations automatically convert only numbers and dates to strings. Other types of values (such as booleans and sequences) must be converted to strings manually, or an error will terminate template processing.

Inserting strings

If the interpolation is in a text section (i.e., not in a string literal expression), the string that it will insert will be automatically escaped if an escape directive is in effect.

We strongly recommend that you utilize this to prevent cross-site-scripting attacks and badly formed HTML pages. Here's an example:

```
<#escape x as x?html>
  ...
  <p>Title: ${book.title}</p>
  <p>Description: <#noescape>${book.description}</#noescape></p>
  <h2>Comments:</h2>
  <#list comments as comment>
    <div class="comment">
      ${comment}
    </div>
  </#list>
  ...
</#escape>
```

This example shows that when generating HTML, you should put the entire template inside the `escape` directive. Thus, if the `book.title` contains an &, it will be replaced with &amp; in the output so the page remains well-formed HTML. If a comment contains tags such as <iframe> (or any other element), they will become to &lt;iframe&gt; and the like for other tags, rendering them harmless. If the data model does contain HTML, you have to neutralize the enclosing escape with a noescape. For example, if `book.description` in the above example is stored as HTML in the database. Without the enclosing escape, the template would look like the following:

```
...
  <p>Title: ${book.title?html}</p>
  <p>Description: ${book.description}</p>
  <h2>Comments:</h2>
  <#list comments as comment>
    <div class="comment">
      ${comment?html}
    </div>
  </#list>
  ...
```

This does the same as the earlier example, but here you may forget some ?html, which is a security risk. In the earlier example, you may forget some noescape-s, which gives bad output too, but does not pose a security risk.

Inserting numerical values

If the expression evaluates to a number, the numerical value will be converted to a string according the default number format. This might include the maximum number of decimals, grouping, and others. Usually, the system sets the default number format, but you can set it using `number_format setting`. For more information, see the `setting` directive in *"Chapter 6. Directive Reference"*. You can also override the default number format for a single interpolation using the `string` built-in.

The decimal separator and other symbols, such as the group separator, depend on the current locale (language, country) that is also set by the system. For example, this template:

```
${1.5}
```

produces output similar to this, if the current locale is English:

```
1.5
```

but if the current locale is Hungarian, the example produces output similar to this because Hungary uses comma as the decimal separator:

```
1,5
```

You can modify the formatting for a single interpolation with the `string` built-in.

Inserting date/time values

If the expression evaluates to a date, the numerical value will be transformed to text according to the default format. Usually, the system sets the default format, but you can change them using the `date_format`, `time_format` and `datetime_format` settings in the `setting` directive. For more information, see the `setting` directive in *"Chapter 6. Directive Reference"*.

You can override the default formatting for a single interpolation using the `string` built-in.

WARNING: To display a date as text, RPL must be able to determine which parts of the date are in use: only the date part (year, month, day), only the time part (hour, minute, second, millisecond), or both. Due to technical limitations, it is not possible to detect this automatically for some variables. If you cannot find out whether the data model contains such variables, you must use the `date`, `time` and `datetime` built-ins; otherwise, an error will terminate template processing.

Inserting boolean values

An attempt to print boolean values with interpolation causes an error and terminates template processing. For example, this will cause an error: `${a == 2}`.

However, you can convert booleans to strings with the `?string` built-in. For example, to print the value of the "married" variable (assuming it's a boolean), write `${married?string("yes", "no")}`.

Conversion rules

The exact rules for converting an expression value to a string (which is then subject to escaping) are shown below, in order:

1. If the value is a number, it is converted to a string in the format specified with the number_format setting.
2. If the value is any type of date (time or date-time), it is converted to a string in the format specified with the time_format, date_format, or datetime_format setting, depending on whether the date information is time-only, date-only, or a date-time. If the date type (date, time, or date-time) cannot be detected, an error will occur.
3. If the value is a string, there is no conversion.
4. In all other cases, an error will terminate template processing.

Defining your own directives

You can define user-defined directives using the macro directive.

A *macro* is a template fragment associated with a variable. You can use that variable in your template as a user-defined directive to help with repetitive tasks. For example, the following code creates a macro variable that prints a big ``Hello Joe!'':

```
<#macro greet>
  <font size="+2">Hello Joe!</font>
</#macro>
```

The macro directive does not print anything; it just creates the macro variable, in this case, a variable called greet. Everything between *<#macro greet>* and *</#macro>* (called *macro definition body*) will be executed when you use the variable as a user-defined directive. You use user-defined directives by writing @ instead of # in the RPL tag and using the variable name as the directive name. Also, the end tag for user-defined directives is mandatory. So you use greet like this:

```
<@greet></@greet>
```

Since <anything></anything> is equivalent to <anything/>, you should use the shorter form:

```
<@greet/>
```

Both examples produce this output:

```
  <font size="+2">Hello Joe!</font>
```

Note that since everything between `<#macro ...>` and `</#macro>` is a template fragment, it can contain interpolations (${...}) and RPL tags (e.g. <#if ...>...</#if>).

Specifying parameters

Let's improve the `greet` macro so it can use any name. For this purpose, you can use parameters. You define the parameters after the name of the macro in the `macro` directive. Macro parameters are local variables. For more information about local variables, see *"Defining variables in the template"*.

The following example defines one parameter for the `greet` macro, `person`:

```
<#macro greet person>
  <font size="+2">Hello ${person}!</font>
</#macro>
```

you can then use this macro as:

```
<@greet person="Fred"/> and <@greet person="Batman"/>
```

which is similar to HTML syntax. The example produces this output:

```
 <font size="+2">Hello Fred!</font>
 and   <font size="+2">Hello Batman!</font>
```

The value of the macro parameter is accessible in the macro definition body as a variable (`person`). As with predefined directives, the value of a parameter (the right side of =) is an RPL expression. This means that the quotation marks around "Fred" and "Batman" are required. `<@greet person=Fred/>` means that you use the value of variable `Fred` for the `person` parameter, rather than the string "Fred". Parameter values need not be a string, they can be a number, a boolean, a hash, a sequence, ...etc. In addition, you can use complex expressions on the left side of = (e.g. `someParam=`*`(price + 50)*1.25)`*.

User-defined directives can have multiple parameters. For example, to add a new parameter color:

```
<#macro greet person color>
  <font size="+2" color="${color}">Hello ${person}!</font>
</#macro>
```

then you can use this macro as:

```
<@greet person="Fred" color="black"/>
```

The order of parameters is not important, so the following example and the previous example are equivalent:

```
<@greet color="black" person="Fred"/>
```

When calling the macro, you can use only those parameters defined in the macro directive and must specify a value for all parameters (in this case: `person` and `color`). The following example produces an error:

```
<@greet person="Fred" color="black" background="green"/>
```

because the `background` parameter is not defined for the macro.

The following example also produces an error:

```
<@greet person="Fred"/>
```

because the value of `color` is not specified.

If a parameter contains the same value most of the time, you can specify that value in the macro definition. To do this, define the parameter as: `param_name=`*`usual_value`*. This way, when using the macro you do not have to specify the parameter since its value is already provided. To use a different value, specify that value when using the directive. For example, to use "black" for `color` unless you specify a different value, define the parameter as:

```
<#macro greet person color="black">
  <font size="+2" color="${color}">Hello ${person}!</font>
</#macro>
```

Now, you can use the macro as:

```
 <@greet person="Fred"/>
```

since it is equivalent to `<@greet person="Fred" color="black"/>` because the value of `color` parameter is known. If `color` is "red", use the macro as:

```
 <@greet person="Fred" color="red"/>
```

NOTE: Per RPL expression rules, the expressions `someParam=foo` and `someParam="${foo}"` are different. The first case uses the value of variable `foo` as the value of the parameter. The second case uses a string literal with an interpolation, so the value of the parameter will be a string -- in this case, the value of `foo` converted to text -- regardless of the variable type. As another example: `someParam=3/4` and `someParam="${3/4}"` are different. The second case converts ¾ to a string, so if the directive expects a numerical value for `someParam`, an error will occur.

Nested content

User-defined directives can have nested content, similarly to predefined directives. To do this, use the `nested` directive. For example, the following code creates a macro that draws borders around its nested content:

```
<#macro border>
  <table border=4 cellspacing=0 cellpadding=4><tr><td>
    <#nested>
  </tr></td></table>
</#macro>
```

`<#nested>` executes the template fragment between its start and end tags. So the following example:

```
<@border>The bordered text</@border>
```

produces this output:

```
   <table border=4 cellspacing=0 cellpadding=4><tr><td>
     The bordered text
   </td></tr></table>
```

You can call the `nested` directive multiple times, for example:

```
<#macro do_thrice>
  <#nested>
  <#nested>
  <#nested>
</#macro>
<@do_thrice>
  Anything.
</@do_thrice>
```

produces this output:

```
   Anything.
   Anything.
   Anything.
```

Note that if you do not use the `nested` directive, the nested content will not be executed. In the following example, nested content will be ignored, since the `greet` macro does not use the `nested` directive:

```
<@greet person="Joe">
  Anything.
</@greet>
```

produces this output:

```
<font size="+2">Hello Joe!</font>
```

The nested content can be any valid RPL, including other user-defined directives. For example:

```
<@border>
  <ul>
  <@do_thrice>
    <li><@greet person="Joe"/>
  </@do_thrice>
  </ul>
</@border>
```

produces this output:

```
  <table border=4 cellspacing=0 cellpadding=4><tr><td>
      <ul>
    <li><font size="+2">Hello Joe!</font>

    <li><font size="+2">Hello Joe!</font>

    <li><font size="+2">Hello Joe!</font>

  </ul>

  </tr></td></table>
```

The local variables of a macro are not visible in the nested content. Because the `y`, `x`, and `count` are the local variables of the macro and are not visible outside of the macro definition, the following example:

```
<#macro repeat count>
  <#local y = "test">
  <#list 1..count as x>
    ${y} ${count}/${x}: <#nested>
  </#list>
</#macro>
<@repeat count=3>${y} ${x} ${count}</@repeat>
```

Outputs an empty string since these variables are consided to be undefined:

```
    test 3/1:
    test 3/2:
    test 3/3:
```

A different set of local variables is used for each macro call, so the following example is correct:

```
<#macro test foo>${foo} (<#nested>) ${foo}</#macro>
<@test foo="A"><@test foo="B"><@test foo="C"/></@test></@test>
```

and produces this output:

```
A (B (C () C) B) A
```

Using loop variables in macros

User-defined directives can have loop variables. For example, let's extend the do_thrice directive of the earlier examples so it exposes the current repetition number as a loop variable. The name of the loop variable is given when calling the directive, while the value of the variables is set by the directive itself. For example:

```
<#macro do_thrice>
  <#nested 1>
  <#nested 2>
  <#nested 3>
</#macro>
<@do_thrice ; x> <#-- user-defined directive uses ";" instead of
"as" -->
  ${x} Anything.
</@do_thrice>
```

produces this output:

```
  1 Anything.
  2 Anything.
  3 Anything.
```

You pass the value of the loop variable for a loop (i.e. repetition of nested content) as the parameter of nested directive. The name of the loop variable is specified in the user-defined directive open tag (<@...>) after the parameters and a semicolon.

A macro can use more the one loop variable. The order of variables is significant. In the following example, c, halfc, and last are loop variables:

```
<#macro repeat count>
  <#list 1..count as x>
    <#nested x, x/2, x==count>
  </#list>
</#macro>
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>
```

produces this output:

```
  1. 0.5
  2. 1
  3. 1.5
  4. 2 Last!
```

You can specify a different number of loop variables in the directive start tag (after the semicolon) than with the `nested` directive. If you specify less loop variables after the semicolon, the extra values in the `nested` directive will not be used, since there is no loop variable to hold those values. For example, the following code is correct:

```
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>
<@repeat count=4 ; c, halfc>
  ${c}. ${halfc}
</@repeat>
<@repeat count=4>
  Just repeat it...
</@repeat>
```

If you specify more variables after the semicolon than with the `nested` directive, the last few loop variables will not be created (i.e. will be undefined in the nested content).

For more information, see *"User-defined directive call"* and *"Macro directive"* sections.

NOTE: You can also define methods in RPL. For more information, see the `function` directive.

You can use namespaces to organize and reuse commonly used macros. For more information about namespaces, see *"About namespaces"* later in this chapter.

Working with variables

In addition to the predefined variables, you can specify your own hashes and variables. In some cases, you do this by declaring your own variables, and in other cases it happens as the implicit result of directives such as `<#list>` and `<#data>`.

Defining variables in the template

In addition to using variables defined in the data model, you can define variables in the template for use only in that template. These temporary variables can be created and replaced using RPL directives. Note that each template processing job has its own private set of these variables that exist while the given page is being rendered. This variable set is initially empty, and will be discarded when the template processing job is finished.

You access variables defined in the template in the same way as variables in the data model.

Template variables take precedence over data model variables with the same name. This means that if a template variable and a data model variable have the same name, the value of the template variable will be used. Note that the value of the data model variable will not be overwritten. If you want to use the data model variable instead of a template variable, use the `.globals` variable. For example, assume the variable `user` in the data model has the value *`Big Joe`*:

```
<#assign user = "Joe Hider">
${user}          <#-- prints: Joe Hider -->
${.globals.user} <#-- prints: Big Joe -->
```

You can define the following variables in a template:

Variable type	Description
Global	Accessible from everywhere in the template, and from the templates inserted with include directive. You can create and replace these variables with the `assign` or `macro` directives.
Local	Can be set only inside a macro definition body, and are only visible from there. A local variable exists only for the duration of

Variable type	Description
	a macro call. You can create and replace local variables inside macro definition bodies with the `local` directive. Macro parameters are local variables.
Loop	Created automatically by directives such as `list` and `data`. These variables exist only between the start and end tags of the directive.

The following example creates and replaces variables with the `assign` directive:

```
<#assign x = 1>  <#-- create variable x -->
${x}
<#assign x = x + 3> <#-- replace variable x -->
${x}
```

produces this output:

```
1
4
```

Local variables overwrite global variables with the same name. Loop variables overwrite local and global variables with the same name. For example:

```
<#assign x = "plain">
1. ${x}  <#-- we see the plain var. here -->
<@test/>
6. ${x}  <#-- the value of plain var. was not changed -->
<#list ["loop"] as x>
    7. ${x}  <#-- now the loop var. hides the plain var. -->
    <#assign x = "plain2"> <#-- replace the plain var, hiding does
not mater here -->
    8. ${x}  <#-- it still hides the plain var. -->
</#list>
9. ${x}  <#-- the new value of plain var. -->

<#macro test>
  2. ${x}  <#-- we still see the plain var. here -->
  <#local x = "local">
  3. ${x}  <#-- now the local var. hides it -->
  <#list ["loop"] as x>
    4. ${x}  <#-- now the loop var. hides the local var. -->
  </#list>
  5. ${x}  <#-- now we see the local var. again -->
</#macro>
```

produces this output:

```
1. plain
  2. plain
  3. local
    4. loop
  5. local
6. plain
    7. loop
    8. loop
9. plain2
```

An inner loop variable overwrites an outer loop variable:

```
<#list ["loop 1"] as x>
  ${x}
  <#list ["loop 2"] as x>
    ${x}
    <#list ["loop 3"] as x>
      ${x}
    </#list>
    ${x}
  </#list>
  ${x}
</#list>
```

produces this output:

```
  loop 1
    loop 2
      loop 3
    loop 2
  loop 1
```

Note that the value of a loop variable is set by the directive invocation that created it (`list` in the example). This is the only way to change the value of a loop variable. However, as shown in the above example, you can temporarily hide a loop variable with another loop variable.

About predefined variables

Predefined variables are defined by the system, and are usually initialized from the campaign definition and other internal values. Some of the more useful ones are `campaign.name` and `campaign.marketingprogram`. For a complete list of predefined variables, see the *"About namespaces"* section later in this chapter.

About namespaces

The set of variables created in the template with the `assign` and `macro` directives and the data from data sources is called a *namespace*. A namespace is a multi-level structure that uses hashes to store values and other hashes.

Normally, you use only the main namespace. Multiple name spaces are used if you are building a reusable collection of macros, functions and other variables (known as a *library*). You should use a separate namespace for each library. This is to avoid name clashes in cases when you have a big collection of macros that you use in several projects or want to share with others. In such cases, it is impossible ensure that the library does not have a macro (or other variable) with the same name as a variable in the data model or in another library used in the template.

Creating a library

This section uses examples to illustrate how to create a library.

Assume you commonly need the variables `copyright` and `mail`. The following example creates those variables:

```
<#macro copyright date>
  <p>Copyright (C) ${date} Julia Smith. All rights reserved.</p>
</#macro>

<#assign mail = "jsmith@acme.com">
```

These variables are stored in the file */contentlibrary/lib/my_test.htm*, in the same directory as the templates.

Assume you want to use this in *content.htm*. If you use `<#include "cms://contentlibrary/lib/my_test.htm">` in content.htm, it will create the two variables in the main namespace. Instead, you want to create these variables in a namespace that is used exclusively by the library. To do this, use the `import` directive instead of the `include` directive. `import` is similar to `include`, but it will create an empty namespace for */contentlibrary/lib/my_test.htm* and execute the macro there. The new namespace will include only the variables of the data model (since they are visible from everywhere), and the two variables from the macro. `import` also creates a new hash variable in the namespace used by the caller of `import` (in this case, *`content.htm`*, which uses the main namespace). The hash variable provides content.htm access to the new namespace and its variables . content.htm will look like the following example:

```
<#import "cms://contentlibrary/lib/my_test.htm" as my> <#-- the hash
called "my" will provide access to the namespace -->
<@my.copyright date="1999-2002"/>
${my.mail}
```

Note how the files accesses the variables in the namespace created for *`/contentlibrary/lib/my_test.htm`* using the newly created hash `my`. The above example will produce this output:

```
  <p>Copyright (C) 1999-2002 Julia Smith. All rights reserved.</p>
jsmith@acme.com
```

If you have a variable called `mail` or `copyright` in the main namespace, that would not cause any confusion, since the two templates use separated namespaces. For example, if you modify the copyright macro in /contentlibrary/lib/my_test.htm:

```
<#macro copyright date>
  <p>Copyright (C) ${date} Julia Smith. All rights reserved.
  <br>Email: ${mail}</p>
</#macro>

<#assign mail = "jsmith@acme.com">
```

and then replace `content.htm` with:

```
<#import "cms://contentlibrary/lib/my_test.htm" as my>
<#assign mail="fred@acme.com">
<@my.copyright date="1999-2002"/>
${my.mail}
${mail}
```

the output is:

```
  <p>Copyright (C) 1999-2002 Julia Smith. All rights reserved.
  <br>Email: jsmith@acme.com</p>
jsmith@acme.com
fred@acme.com
```

This is because when you called the `copyright` macro, RPL temporarily switched to the namespace that was created by the import directive for /contentlibrary/lib/my_test.htm. Thus, the `copyright` macro always sees the `mail` variable that exists there, and not the `mail` variable that exists in the main namespace.

Writing the variables of imported namespaces

Occasionally, you might want to create or replace a variable in an imported namespace. You can do this with the `assign` directive with the *`namespace`* parameter. For example, the following code:

```
<#import "/lib/my_test.RPL" as my>
${my.mail}
<#assign mail="jsmith@other.com" in my>
${my.mail}
```

produces this output:

```
jsmith@acme.com
jsmith@other.com
```

Namespaces and the data model

The variables of the data model defined in the datasources and campaign variables are visible from everywhere. For example, if you have a field/variable called `user` in the data model, /contentlibrary/lib/my_test.htm will access that, exactly as content.htm does:

```
<#macro copyright date>
  <p>Copyright (C) ${date} ${user}. All rights reserved.</p>
</#macro>

<#assign mail = "${user}@acme.com">
```

If user is “Fred”, the following example:

```
<#import "cms://contentlibrary/lib/my_test.htm" as my>
<@my.copyright date="1999-2002"/>
${my.mail}
```

produces this output:

```
  <p>Copyright (C) 1999-2002 Fred. All rights reserved.</p>
Fred@acme.com
```

The variables in the namespace (the variables you create with `assign` or `macro` directives) have precedence over the variables of the data model when you are in that namespace. Thus, the content of the data model does not interfere with the variables created by the library.

NOTE: In some unusual situations, you might need to create variables in the template that are visible from all namespaces. Although you cannot change the data model with templates, you can achieve this with the `global` directive. For more information about the global directive, see *"Chapter 6. Directive Reference"*.

Namespace lifecycle

A namespace is identified by the path that was used with the `import` directive. If you try to import multiple times with the same path, the directive will create the namespace and run the template specified by the path only for the very first invocation of `import`. The later imports with the same path will only create a hash in the same namespace. For example, if content.htm is :

```
<#import "/contentlibrary/lib/my_test.htm" as my>
<#import "/contentlibrary/lib/my_test.htm" as foo>
<#import "/contentlibrary/lib/my_test.htm" as bar>
${my.mail}, ${foo.mail}, ${bar.mail}
<#assign mail="jsmith@other.com" in my>
${my.mail}, ${foo.mail}, ${bar.mail}
```

The output is this, since `my`, `foo` and `bar` provide access to the same namespace:

```
jsmith@acme.com, jsmith@acme.com, jsmith@acme.com
jsmith@other.com, jsmith@other.com, jsmith@other.com
```

Note that namespaces are not hierarchical, they exist independently of each other. That is, if you import namespace N2 while in name space N1, N2 will not be inside

N1. Instead, N1 gets a hash by which it can access N2. This is the same N2 namespace that you would access if you import N2 while in the main namespace.

Each template processing job has its own private set of namespaces. Each template processing job exists only while the given page is being rendered.

NOTE: When namespaces are used during a launch, they are not preserved across multiple recipients. Namespaces are recreated for each record.

Making a library available to others

You can share a library by making it available in the Content Library. To prevent clashes with the names of libraries used by other authors, and to make it easy to write libraries that import other published libraries, there is a de-facto standard that specifies the format of library paths.

The standard is that the library must be available (importable) for templates and other libraries with a path like this:

    /contentlibrary/lib/your_library.htm

For example, if you develop a widget library, then the path of the RPL file to import should be:

    /contentlibrary/lib/ widget.htm

IMPORTANT: The path must not contain upper-case letters. To separate words, use _, as in wml_form (not wmlForm).

These are “best-practice” recommendations. The system will allow other standards.

Handling white space

The control of white space in a template is a problem that to some extent haunts every template engine.

In the following example, we marked the components of the template with colors: text, interpolation, RPL tag. With the [BR]-s represent the line breaks.

```
<p>List of users:[BR]
<#assign users = [{"name":"Joe",        "hidden":false},[BR]
                  {"name":"James Bond", "hidden":true},[BR]
                  {"name":"Julia",      "hidden":false}]>[BR]
<ul>[BR]
<#list users as user>[BR]
  <#if !user.hidden>[BR]
  <li>${user.name}[BR]
  </#if>[BR]
</#list>[BR]
</ul>[BR]
<p>That's all.
```

If RPL were to output all text as is, the output would be:

```
<p>List of users:[BR]
[BR]
<ul>[BR]
[BR]
  [BR]
  <li>Joe[BR]
  [BR]
[BR]
  [BR]
[BR]
  [BR]
  <li>Julia[BR]
  [BR]
[BR]
</ul>[BR]
<p>That's all.
```

This output contains a lot of unwanted spaces and line breaks.

To resolve this problem, RPL provides the tools described in this section.

To ignore certain white space, you can use one of the following methods:

White space stripping
This method automatically ignores typical superfluous white space around RPL tags. It can be enabled or disabled for each template. For more information, see *"Stripping white space"* later in this section.

Trimmer directives
You can use the following trimmer directives: `t`, `rt`, `lt`, `join`. With these directives, you can explicitly instruct RPL to ignore certain type of white space. For more information, see *"Chapter 6. Directive Reference"*.

RPL parameter `strip_text`
This removes all top level text from the template. It is useful for templates that contain only macro definitions and other directives that do not produce output. This parameter removes the line breaks between the macro definitions and between other top level directives to improve readability of the template.

To remove white space from output, you can use the `compress` and `join` directives. For more information, see *"Using the compress directive"* later in this section.

Stripping white space

If this feature is enabled for a template, the following types of white space are automatically ignored:

Indentation white space, and trailing white space at the end of the line
Indentation and trailing white space, including line breaks, will be ignored in lines that contains only RPL tags and/or RPL comments. For example, if a line contains only an `<#if ...>`, then the indentation before the tag and the line break after the tag will be stripped. However, if the line contains `<#if ...>x`, the white space in that line will not be stripped, because the `x` is not an RPL tag. Note that white space is stripped in a line that contains `<#if ...><#list ...>`, but not in a line that

contains `<#if ...> <#list ...>`. This is because the white space between the two RPL tags is embedded white space, not indentation or trailing white space.

White space between the macro, function, assign, global, local, rpl, and import directives

With these directives, white space is stripped if there is only white space and/or RPL comments between the directives. This means that you can put empty lines between macro definitions and assignments as spacing for better readability, without printing needless empty lines. For example, white space will be stripped in the following template, except in the highlighted line:

```
<p>List of users:[BR]
<#assign users = [{"name":"Joe",        "hidden":false},
                  {"name":"James Bond", "hidden":true},
                  {"name":"Julia",      "hidden":false}]>
<ul>[BR]
<#list users as user>
  <#if !user.hidden>
  <li>${user.name}[BR]
  </#if>
</#list>
</ul>[BR]
<p>That's all.
```

produces this output:

```
<p>List of users:[BR]
<ul>[BR]
  <li>Joe[BR]
  <li>Julia[BR]
</ul>[BR]
<p>That's all.
```

Note that enabling white space stripping does not degrade the performance of template execution because white space stripping is done during template loading.

You can enable or disable white space stripping for a template using the `rpl` directive. If you do not specify this with the `rpl` directive, white space stripping will be enabled.

You can enable or disable white space stripping for a single line using the `nt` directive.

Using the compress directive

The `compress` directive works on the generated output, not on the template. The directive inspects the output and removes indentations, empty lines and repeated spaces/tabs. For example:

```
<#compress>
<#assign users = [{"name":"Joe",        "hidden":false},
                  {"name":"James Bond", "hidden":true},
                  {"name":"Julia",      "hidden":false}]>
List of users:
<#list users as user>
  <#if !user.hidden>
  - ${user.name}
  </#if>
</#list>
That's all.
</#compress>
```

produces this output:

```
List of users:
- Joe
- Julia
That's all.
```

Note that `compress` works independently of white space stripping. So it is possible that the white space of template is stripped, and later the produced output is compressed.

By default, a user-defined directive called `compress` is available in the data model. This is the same as the pre-defined directive, except that you may optionally set the `single_line` parameter, which will remove all intervening line breaks. If you replace `<#compress>...</#compress>` in the last example with `<@compress single_line=true>...</@compress>`, you will get this output:

```
List of users: - Joe - Julia That's all.
```

For more information about the compress directive, see *"Chapter 6. Directive Reference"*.

Chapter 3. Working with Forms and Link Tracking

RPL can help you produce links to additional forms in the system and to track clicks to either those forms or to other links.

About forms

Forms provide a way to capture information about your audience. They consist of the form HTML and instructions about what should happen after a visitor submits the form. For example, you can send a follow-up email after the form is submitted. You can create forms in Responsys Interact using the Forms wizard accessed from the Forms tab.

To support “view in browser” functionality, the system allows campaigns to be used as forms as well. That is, a link can be generated to point to a campaign, and when a recipient clicks the link, the campaign will be retrieved as a form.

You can also create external links that point to a URL outside of Interact. These are usually links to your own web site or other sites.

You create form links using the `form` method.

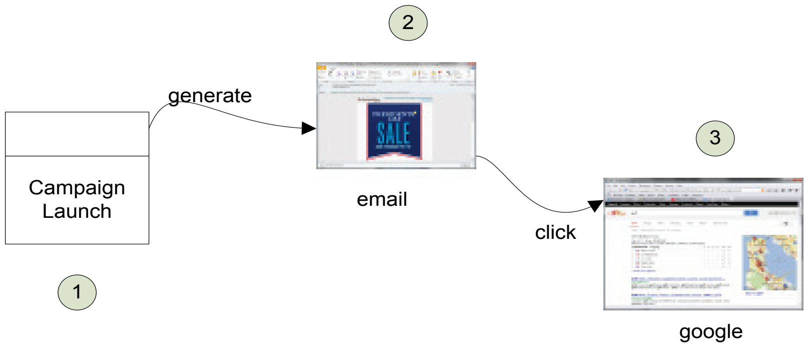
Creating links in personalization

To code a link, use the HTML anchor tag.

The following example creates an external link to Google with a hard-coded query string that searches for the word “golf”:

```
<a href=”http://www.google.com?q=golf”>Search for Golf Clubs</a>
```

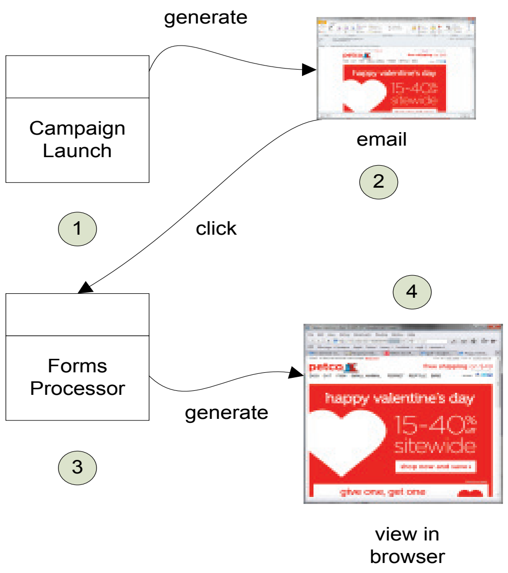
The following illustration shows how the link is generated.



The following example creates a link to a form:

```
<a href=”${form(‘user preferences’)}”>Email Preferences</a>
```

In this example, a link to the internal form processor is created when the personalization occurs. This link is an encoded URL to the Interact form processor. The following diagram illustrates this flow.



About link tables

A *link table* is a collection of links that allows you to track recipient links in a campaign. Each link is defined with a name and a URL.

External links can be, and often are, coded inside the personalization template with the anchor tag, as shown in the previous section. However, when you want to track links, you can code these link URLs in a link table.

Consider the following example:

```
Link name: google search
Link URL: www.google.com?q=golf
```

This example defines a link in a link table to Google, with a predefined query.

Forms can also be coded in the personalization template or can be added as a link in a link table. Consider the following example:

```
Link name: view in browser
Link URL: ${form(‘user preferences’)}
```

This link is added to the link table so that the user preferences link can be tracked.

To track clicks in RPL, you code your anchor tags with the `clickthrough` method that will be described later.

For instance, to track a click for the Google search example, you could write:

```
<a href=”${clickthrough(‘google search’)}”>Search</a>
```

The previous examples show that, while the `form` method uses form names (or campaign names), the `clickthrough` method uses link names.

Passing parameters for forms

When either a form or external link is invoked, sometimes you need to pass data that is known to the personalization. For instance, the customer identifier or as in the previous example with Google, the query might be computed during personalization. In this case, the link is produced with parameter replacement syntax as shown in the following example. The example creates a link using the users preferred sport, with the link naming it as a search for that topic:

```
<a href=www.google.com?q=${extensiontable.favoritesport}>Search for
${extensiontable.favoritesport?cap_first</a>
```

If the user's favorite sport is hockey, the example will produce this output:

```
<a href="www.google.com?q=hockey">Search for Hockey</a>
```

You can use the same technique for links to an internal form. Consider the following example:

```
<a href="${form('user
preferences','"extensiontable.favoritesport")}">View Preferences</a>
```

This form invocation formats and generates the URL that will point to the form processor. The parameters are encoded in the resulting URL.

The field names in the `form` method are specified as string scalars. This indicates to the `form` method that:

1. The value for the field *extensiontable.favoritesport* must be obtained from the recipient record.
2. If the actual field name for *extensiontable.favoritesport* maps to the database field FAVORITE_SPORT, a field named FAVORITE_SPORT is sent with this value and made available to the form processor. With this information, the form processor can personalize with the given name.

A variation of the above mentioned method of passing fields by name is to pass the field name with the provided value. Consider the following example:

```
<a href="${form('user preferences', 'FAVORITE_SPORT=soccer')}">View
Preferences</a>
```

In this case, a field name is passed with an equals sign and the required value. This name can be used in the form processor to retrieve the value of the field. You do not need to specify the data source or column aliases since the field expected in the form processor is known. This skips the step of retrieving the field name from the record, but still sends the parameter to the form with the given field name.

You can use more complex expressions in the creation of links. For example, you can use string concatenation expressions to compute the value:

```
<a href="${form('user
preferences','FAVORITE_SPORT='+profile.sport)}">View Preferences</a>
```

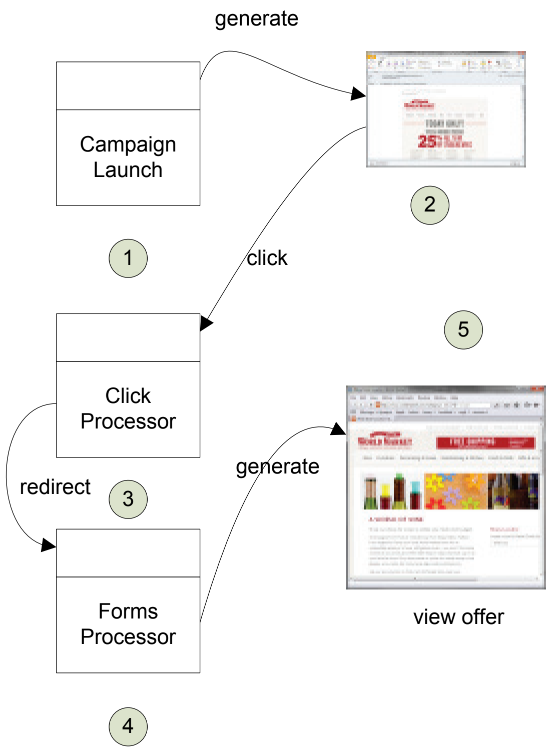
Although the `form` method can be placed in the link table as part of the link URL, its capabilities in a link URL are limited.

Note the use of apostrophes in the previous examples. This is a best practice when using anchors, since the href for the anchor itself is enclosed in double quotes. This will be important with implicit click tracking, described later in this chapter.

About click tracking

To track a link, the link is stored in a link table with a link name as described above. On the live report, the link displays the number of clicks to the given link.

The following diagram illustrates the flow for tracking a link to an internal form.



Tracking can be achieved by the means described below.

Explicit external link tracking

To explicitly track a link, you can add the following code to your personalization template:

```
<a href="${clickthrough('google search',
'extensiontable.favoritesport')}">Search for your favorite sport</a>
```

This will produce a link to the click processor with the given link name encoded. The click processor tracks the click. After a click is processed, the user is redirected to the forms processor or to the final external link. In our example, the external link will be reached.

In this case, the link would have been coded as:

```
Link name: google search
Link URL: www.google.com?q=${extensiontable.favoritesport}
```

Note that the field name in the `clickthrough` method is passed as a string scalar. Two things happen when this is done:

1. The `clickthrough` method knows that the value of the field *extensiontable.favoritesport* needs to be obtained.
2. The URL to the click processor that will be created sends the value of the field with the retrieved value.

Alternately, you can code the `clickthrough` method invocation with the actual value of the field, in the same way as in the `form` method invocation described above. Consider the following example:

```
<a href=”${clickthrough(‘google search‘,
’FAVORITE_SPORT=basketball’)}”>Search for basketball</a>
```

In this case the field name is passed, plus an equals sign, plus the actual value. This method avoids the retrieval of data from the user’s record, but preserves the semantics of sending the field name as before. The field name can be any valid string, including alias names. The click processor uses the given name to retrieve the information.

Explicit internal form tracking

The following example illustrates how to track clicks to a form:

```
<a href=”${clickthrough(‘user info‘, ’profile.firstname‘,
’profile.lastname’)}
```

and the form link could be set up as:

```
Link name: user info
Link URL: ${form(‘user preferences‘,‘profile.firstname‘,
’profile.lastname’)}
```

Explicit link tracking with parameter analysis

In this mode, the links are coded with the `clickthrough` method as before, but with only one parameter, the link name.

When the system encounters the `clickthrough` method, it opens the link table, obtains the URL, and determines the needed parameters from that link URL.

Consider the following example:

```
Link name: google search
Link URL: www.google.com?q=${extensiontable.favoritesport}
```

and the following anchor in the HTML template:

```
<a href="${clickthrough('google search')}">Search</a>
```

Notice that the Link URL requires `extensiontable.favoritesport`, which is sent from the `clickthrough` method call. By analyzing the link URL, the `clickthrough` method will recognize that it needs to send this value.

You can specify parameters in LINK_URL using one of the following formats:

${datasourcealias.fieldalias}
Using this format, the proper data for `datasourcealias.fieldalias` will be sent. You can use this format with either the `form` or `clickthrough` method.

form('formname', 'datasourcealias.fieldalias', …)
You can use this format only with the `form` method.
Not that with this format, you must enclose the field reference in single quotes (').

Implicit link tracking

You can also code click tracking without using the `clickthrough` method. This is done so that the parameters sent by `clickthrough` and the destination link match automatically.

In this way of tracking, the personalization processing examines the links inside the anchor tags and tries to match the href with any link URL. The search is done character by character, matching only identical strings. When the personalization phase detects a match in this fashion, it internally creates an implicit `clickthrough` method call. For example, if the personalization template is:

```
<a href=”www.google.com?q=${extensiontable.favoritesport}”>Search
for your favorite sport</a>
```

and the link table is:

```
Link name: google search
Link URL: www.google.com?q=${extensiontable.favoritesport}
```

RPL will internally code a `clickthrough` method call that will ensure that the required parameters are sent (in this case, the `extensiontable.favoritesport` field). It is important to note that the form method in the template will not be executed since it is replaced by the `clickthrough` method. The form will be computed only after the click is received.

Notice that in the above example, the URL in the anchor and the link URL are identical.

This method works only with anchors. If the links are coded in a different fashion, for instance with javascript, then the manual explicit `clickthrough` method is required.

Implicit link tracking can be used with the `form` method as a link URL and the anchor’s href. The URL and href must be identical. In this case, the `form` method will be replaced with a `clickthrough` equivalent, and the `form` method will not be executed during personalization of the message.

In the implicit link tracking, you must use single quotes. Since the href in the anchor starts with a double quote, its end will be determined by the first double quote after it. If the `form` method is coded as part of implicit click tracking, and the form method has double quotes, the anchor end will be improperly recognized.

The following example is incorrect because of improper use of double quotes:

```
Link name: user preferences
Link URL: ${form("user preferences)}
```

If the above example is used with the following code in the template:

```
<a href=”${form(“user preferences”)}”>User preferences</a>
```

Both the link URL and the href appear to be identical. However, for implicit tracking the href is assumed to be:

```
${form(
```

This is because the first double quotes inside of the `form` method will terminate the anchor. As a result, the form will not be implicitly tracked.

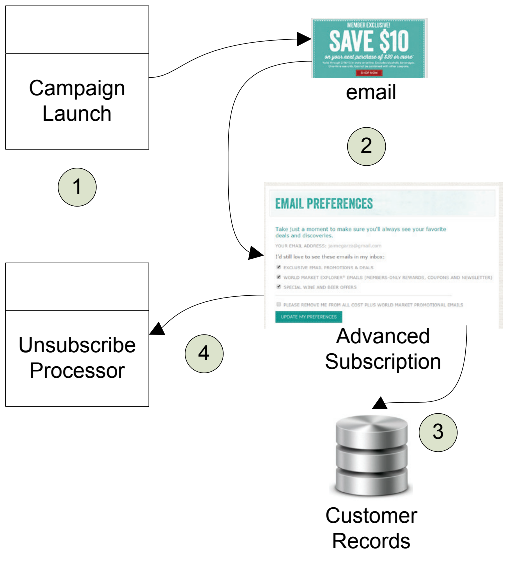
Using tracking parameters

Sometimes it is not enough to use link tables, the `clickthrough` method, or the `form` method. In some complex cases, you might want to perform advanced operations on your own web page, with an eventual call to Oracle Responsys. One such example is the use of advanced unsubscribe forms hosted at an external web site.

Consider the following example:

> You need to create a more complex form for subscriptions in a custom web server. That form will update internal customer records in the customer's database. For reporting purposes, Oracle Responsys must be notified of any final unsubscribes so that a campaign report shows the actual number of unsubscribes with the campain.

In this case, it is important to let the Oracle Responsys server know of a unsubscribe action associated with a campaign so that live reports and other reports attribute that unsubscribe action to the campaign, as illustrated in the following diagram:



Step 4 is performed from the unsubscribe web page when the unsubscribe action is confirmed. In order to notify Oracle Responsys about an unsubscribe action, a pair of parameters, called `_ri_` and `_ei_`, are required in the web call. These parameters should be present in the call of step 4 in the diagram. To use these parameters, both parameters should come from the campaign launch, into the email, then into the unsubscribe form, and finally to Oracle Responsys unsubscribe endpoint.

To create the `_ri_` and `_ei_` parameters, you use the `tracking` method. This method returns the two parameters together as a single string that can be concatenated with other string parameters of a URL. It follows the proper format for URL formation.

Consider the following code:

```
<a href="http://www.example.com/unsubscribe?clientid=${CUSTOMER_ID_}
&tracking(campaign.name)}">Manage Your Subscriptions</a>
```

This code can result in a link with the following content:

```
<a
href="http://www.example.com/unsubscribe?clientid=1587932&_ri_=X0Gzc
2X%3DWQrnHjFHQGmiM4zghSE9SX5lfzaAFJCysuksIfVXyjLNpLOfhKLX%3DtHmjpLxI
LllLjQgLlVXMtX%3DWQrnHjFHQGjzgJty3zdwgS5O2guP3S4DfnLUbn&_ei_=EmKw7b8
wc39AGbHzXbi74rGgqemwVKioX14uG1dUmC7rqFJzl6gjzORDrl6kNnVNY5xF3vZEy46
4L1JwOHe06UA_0Rg-_kshjTchpfTNpojsuBpN">Manage Your Subscriptions</a>
```

In this example, the `tracking` method produced the *_ri_ and _ei_* parameters that are sent to the unsubscribe form in www.example.com

You must now provide the form with a way to call Oracle Responsys on an unsubscribe. This can be achieved by multiple link calling mechanisms in HTML, including anchor tabs, forms, or javascript. More often, the call is embedded in an image tag on the unsubscribe confirmation page as follows:

```
<img height="1" width="1"
SRC="http://rsp.rsys2.net/pub/optout/UnsubscribeOneStepConfirmAction
? _ri_=
X0Gzc2X%3DS%3ANkHzfLgH%3ASRWR%3AAWRS%3AAWRD%3ATCRU%3ASBAR%3AG%3AvVXM
tX%3DS%3ANkHzfLgH%3ASRWR%3AAWRS%3AAWRW%3ATCRS%3ASBAR%3Azg%3AaVXyjLNp
LOfhKLX%3DHIJKxjLkihgkPkQJhu&_ei_=&YES=Remove%20Me">
```

Note that the URL is composed of four sections:

1. The unsubscribe URL. This URL is provided to you as part of the account. In this case, we are using *http://rsp.rsys2.net* as an example. Please refer to your documentation to obtain this address.
2. The unsubscribe sub-address, given as */pub/optout/UnsubscribeOneStepConfirmAction?*
3. The *_ri_* and *_ei_* paratemers, as sent in the email
4. The required &YES=Remove%20Me that is part of the parameters needed by the unsubscribe URL. This is always the same.

For more information, see the `tracking` method.

About internal form processing

This section describes the way forms are processed.

Campaign personalization

The campaign personalization occurs at the time that an email is sent. This is the phase in which the `form` and `clickthrough` method invocations are executed.

At this stage, the fields are used by passing a string literal to the `form` and `clickthrough` methods, and the fields are evaluated for both their name and value.

Internal form processor

The internal form processor receives requests to display a form and works in one of the following ways:

- The `form` parameters are all provided in the URL in the personalization template. In this mode, all data comes from the form link. In this case, the options to the form method can be omitted as {}, or `usedb` can be set as follows:

```
${form('formname', {'usedb':false}, parameter1, parameter2)}
```

  The `form` parameters are all provided by accessing the user's record when personalization occurs. The form processor does not need to retrieve any data as all the data is provided. This is the default, and is the faster of the two ways.

- In other cases, the `form` method invocation should indicate that the form processor must retrieve all data from the recipient record. In this case, the `form` method cannot include any parameters. This is achieved by setting the `usedb` option in the form parameter as follows:

```
${form('formname', {'usedb':true})}
```

  In this case, no additional parameters are allowed, as all data is retrieved from the personalization record inside the form processor.

Downtime form processor

When the system is down for maintenance, a downtime form processor displays a predefined response for the given form, or a page indicating that the system is down for maintenance.

Click processor

The click processor is different from the form processor in that it receives a link name instead of to a form name. It is managed in a more compact way by using internal identifiers. The click processor records a click to the given link, and redirects the request to the link's destination URL, whether an internal form or an external link.

When the click processor receives a request, two things happen:

1. A counter is incremented for the given link, and
2. The link URL is obtained and a redirect is issued to the destination URL.

To properly redirect, the click processor might personalize the link with the data it received. For this reason, it is important to send all data that the link URL will require.

The click processor can only personalize three types of code:

- Link URLs without personalization
- Link URLs with a `form` method in which the parameters are the names of the fields sent by the `clickthrough` method
- Field values of the form ${datasourcealias.fieldalias}

Downtime clickthrough processor

When the system is down for maintenance, the click processor still records that a click was performed for a given link, and attempts to redirect.

The redirection will succeed if the link URL has no replacements, or the link URL has field replacements of the ${datasourcealias.fieldalias} type.

When the Link URL contains embedded `${form(…)}`replacements, the downtime click processor cannot redirect.

About form and clickthrough methods in campaigns

In Mobile SMS campaigns, the `form` and `clickthrough` methods return a temporary shortened version of the link. The shortened URL is valid fo a short period of time.

In email campaigns, these methods return a long URL.

Chapter 4. XML Processing

This chapter describes how to use RPL to process XML documents.

This chapter includes a brief XML overview, but is intended for users who are familiar with XML.

XML terminology

This chapter uses the XML terms and concepts used in the guide.

DOM
The DOM defines the logical structure of documents and the way a document is accessed and manipulated.

The DOM presents an XML document as a tree structure.

Node
Everything in an XML document is a node. For example, the entire document is the `document` node, and every element is an `element` node.

Root node
The topmost node of a tree. In the case of XML documents, it is always the `document` node, and not the top-most element.

Parent node
An immediate ascendant of another node. The root node is the parent of all other nodes in the document.

Child node
An immediate descendant of another node. Note that element attributes are not generally considered child elements.

About the XML document in this chapter

The examples in this chapter use the following XML document:

```
<book>
  <title>Test Book</title>
  <chapter>
    <title>Ch1</title>
    <para>p1.1</para>
    <para>p1.2</para>
    <para>p1.3</para>
  </chapter>
  <chapter>
    <title>Ch2</title>
    <para>p2.1</para>
    <para>p2.2</para>
  </chapter>
</book>
```

The node tree of this document is:

```
document
 |
 +- element book
     |
     +- text "\n  "
     |
     +- element title
     |   |
     |   +- text "Test Book"
     |
     +- text "\n  "
     |
     +- element chapter
     |   |
     |   +- text "\n    "
     |   |
     |   +- element title
     |   |   |
     |   |   +- text "Ch1"
     |   |
     |   +- text "\n    "
     |   |
     |   +- element para
     |   |   |
     |   |   +- text "p1.1"
     |   |
     |   +- text "\n    "
     |   |
     |   +- element para
```

```
      |    |    |
      |    |    +- text "p1.2"
      |    |
      |    +- text "\n    "
      |    |
      |    +- element para
      |       |
      |       +- text "p1.3"
      |
      +- element
           |
           +- text "\n    "
           |
           +- element title
           |    |
           |    +- text "Ch2"
           |
           +- text "\n    "
           |
           +- element para
           |    |
           |    +- text "p2.1"
           |
           +- text "\n    "
           |
           +- element para
                |
                +- text "p2.2"
```

Note that the "\n "-s are the line breaks (indicated here with \n, an escape sequence used in RPL string literals) and the indentation spaces between the tags.

Putting the XML into the data model

If a string with the XML is provided in the example in a variable `xml`, you can execute the following RPL code to create a node:

```
<#assign doc=parsexml(xml)>
...
```

This will create the data node `doc`.

Alternately, you can enter the above XML in multiple lines and with concatenation as:

```
<#assign doc=parsexml(
'<book>' +
'  <title>Test Book</title>' +
'  <chapter>' +
'    <title>Ch1</title>' +
'    <para>p1.1</para>' +
'    <para>p1.2</para>' +
'    <para>p1.3</para>' +
'  </chapter>' +
'  <chapter>' +
'    <title>Ch2</title>' +
'    <para>p2.1</para>' +
'    <para>p2.2</para>' +
'  </chapter>' +
'</book>')>
```

You can also store an XML file in the Content Library, then load it into the data model as follows:

```
<#assign doc=parsexml(load("cms://contentlibrary/books/book.xml"))>
…
```

You can download content from an external site by loading the content from a web server as follows:

```
<#assign
doc=parsexml(load("http://www.example.com/books/book.xml"))>
…
```

NOTE: To be able to download from http or https, your account must be enabled by your Responsys representative.

In general, the source of the XML string can come from any variable. It is common practice to enter XML as part of dynamic variables, but in some instances the XML can come from database fields or from other string constants.

You can provide the XML directly as field of your contact list, and load it as follows:

```
<#assign doc=parsexml(profile.purchases)>
…
```

We do not recommend this practice because it might increase the size of your database if the record's XML becomes large.

Working with imperative XML processing

With *imperative processing*, you write RPL programs that walk the DOM tree to find the different types of nodes. The nodes are used as RPL elements to obtain data.

This section uses the DOM tree of the example document and the variable `doc` created in the previous example.

The examples in this section assume that you put the XML document into the data model as the variable `doc`. The `doc` variable corresponds to the root of the DOM tree, `document`. This section shows an example of how to use the `doc` variable.

Accessing elements by name

The following example prints the title of the book:

```
<h1>${doc.book.title}</h1>
```

and produces this output:

```
<h1>Test Book</h1>
```

Both `doc` and `book` can be used as hashes; you get their child nodes as sub-variables. You describe the path by which you reach the target (element title) in the DOM tree. You might notice that ${`doc.book.title`} seems to instruct RPL to print the `title` element itself, but the example prints its child text node. This works because elements are string variables as well as hash variables. The scalar value of an element node is the string resulting from the concatenation of all its text child nodes. However, trying to use an element as a scalar will cause an error if the element has child elements. For example `${doc.book}` will produce an error.

The following example prints the titles of the two chapters:

```
<h2>${doc.book.chapter[0].title}</h2>
<h2>${doc.book.chapter[1].title}</h2>
```

Here, since `book` has 2 chapter element children, `doc.book.chapter` is a sequence that stores the two element nodes. Thus, we can generalize the above example, so that it works with any number of chapters:

```
<#list doc.book.chapter as ch>
  <h2>${ch.title}</h2>
</#list>
```

When you access an element as a hash sub-variable, it is always a sequence as well as a hash and string. If the sequence contains only one item, then the variable also acts as that item. So, returning to the first example, the following code will print the book title as well:

```
<h1>${doc.book[0].title[0]}</h1>
```

If there is only one book element, and that a book has only one title, you can omit the `[0]`. `${doc.book.chapter.title}` will work as well if the book has only one chapter. If the book has more than one chapter, the previous code will produce an error. If the element book has no `chapter` child, then `doc.book.chapter` will be a 0 length sequence, so the code with `<#list ...>` will work.

It is important to realize that if `book` has no chapters, then `book.chapter` is an empty sequence, so `doc.book.chapter??` will return `true`. To check whether a child node exists, use `doc.book.chapter[0]??` (or `doc.book.chapter?size == 0`). You can use any missing value handler operator (e.g. `doc.book.author[0]!"Anonymous"`), but make sure to include `[0]`.

NOTE: The rule with sequences of size of one is a convenience feature of the XML wrapper (implemented via multi-type RPL variables). It will not work with other sequences.

To finish the example, print all paragraphs of each chapter:

```
<h1>${doc.book.title}</h1>
<#list doc.book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
```

produces this output:

```
<h1>Test</h1>
  <h2>Ch1</h2>
    <p>p1.1
    <p>p1.2
    <p>p1.3
  <h2>Ch2</h2>
    <p>p2.1
    <p>p2.2
```

The above example can also be written as:

```
<#assign book = doc.book>
<h1>${book.title}</h1>
<#list book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
```

Finally, to illustrate general usage of the child selector mechanism, the following example lists all paragraphs of the example XML document:

```

<#list doc.book.chapter.para as p>
  <p>${p}
</#list>
```

produces this output:

```
  <p>p1.1
  <p>p1.2
  <p>p1.3
  <p>p2.1
  <p>p2.2
```

This example shows that hash sub-variables select the children of a sequence of nodes (in the earlier examples, the sequence had one 1 item). In this case, the sub-variable `chapter` returns a sequence of size 2 (since there are two chapters), and then sub-variable `para` selects the `para` child nodes of all nodes in that sequence.

A negative consequence of this mechanism is that `doc.anything.anytingElse` will evaluate to an empty sequence and will not produce an error.

Accessing attributes

In this section, the XML is the same as in the previous section, except that it uses attributes for titles instead of elements:

```
<!-- THIS XML IS USED FOR THE "Accessing attributes" CHAPTER ONLY! -
->
<!-- Outside this chapter examples use the XML from earlier.       -
->

<book title="Test">
  <chapter title="Ch1">
    <para>p1.1</para>
    <para>p1.2</para>
    <para>p1.3</para>
  </chapter>
  <chapter title="Ch2">
    <para>p2.1</para>
    <para>p2.2</para>
  </chapter>
</book>
```

You can access the attributes of an element in the same way as the child elements of an element, except that you use an at sign (@) before the name of the attribute as shown in the following example:

```
<#assign book = doc.book>
<h1>${book.@title}</h1>
<#list book.chapter as ch>
  <h2>${ch.@title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
```

This will produce the same output as the previous example.

Getting attributes follows the same logic as getting child elements, so the result of ch.@title above is a sequence of size one. If there were no title attribute, the result would be a sequence of size 0. Using built-ins here is tricky: to find out whether foo has an attribute called bar, you must write foo.@bar[0]??. (foo.@bar)?? is incorrect because it always returns true. Similarly, if you want a default value for the bar attribute, write foo.@bar[0]!"theDefaultValue".

As with child elements, you can select attributes of multiple nodes. For example, the following code prints the titles of all chapters:

```
<#list doc.book.chapter.@title as t>
  ${t}
</#list>
```

Exploring the DOM tree

The following example enumerates all child nodes of the book element:

```
<#list doc.book?children as c>
- ${c?node_type} <#if c?node_type = 'element'>${c?node_name}</#if>
</#list>
```

produces this output:

```
- text
- element title
- text
- element chapter
- text
- element chapter
- text
```

?node_name returns the name of element for element nodes. For other node types, it also returns something, but that's mainly useful for declarative XML processing, which will be discussed later in this chapter.

If the book element had attributes, they would not appear in the above list. You can get a list that contains all attributes of the element, with the sub-variable @@ of the element variable. If you modify the first line of the XML to this:

```
<book foo="Foo" bar="Bar" baaz="Baaz">
```

then the following example:

```
<#list doc.book.@@ as attr>
- ${attr?node_name} = ${attr}
</#list>
```

produces this output:

```
- baaz = Baaz
- bar = Bar
- foo = Foo
```

RPL provides a convenience sub-variable to list only the children of an element. For example:

```
<#list doc.book.* as c>
- ${c?node_name}
</#list>
```

produces this output:

```
- title
- chapter
- chapter
```

To get the parent of an element, use the `parent` built-in as shown in the following example:

```
<#assign e = doc.book.chapter[0].para[0]>
<#-- Now e is the first para of the first chapter -->
${e?node_name}
${e?parent?node_name}
${e?parent?parent?node_name}
${e?parent?parent?parent?node_name}
```

produces this output:

```
para
chapter
book
@document
```

The last line in the example reaches the root of the DOM tree, the `document` node. This is not an element, therefore it appears differently in the output.

To return to the `document` node, use the `root` built-in as shown in the following example:

```
<#assign e = doc.book.chapter[0].para[0]>
${e?root?node_name}
${e?root.book.title}
```

produces this output:

```
@document
Test Book
```

For a complete list of built-ins you can use to navigate in the DOM tree, see *“Chapter 5, “Built-in Reference”*.

Using XPath expressions

If a hash key used with a node variable cannot be interpreted otherwise (see the next section for the precise definition), then it will be interpreted as an XPath expression. For more information about XPath, please visit *http://www.w3.org/TR/xpath*.

For example, to list the para elements of the chapter called *Ch1*:

```
<#list doc["book/chapter[title='Ch1']/para"] as p>
  <p>${p}
</#list>
```

produces this output:

```
  <p>p1.1
  <p>p1.2
  <p>p1.3
```

The rule for sequences of length one (explained in an earlier section) applies to XPath results as well. That is, if the resulting sequence contains exactly one node, it also acts as the node itself. For example, to print the first paragraph of chapter *Ch1*:

```
${doc["book/chapter[title='Ch1']/para[1]"]}
```

The above example produces the same output as:

```
${doc["book/chapter[title='Ch1']/para[1]"][0]}
```

The context node of the XPath expression is the node (or sequence of nodes) whose hash sub-variable is used to issue the XPath expression. Thus, the following example produces the same output as the previous one:

```
${doc.book["chapter[title='Ch1']/para[1]"]}
```

Also note that XPath indexes sequence items from one. Thus, to select the first chapter, the XPath expression is "`/book/chapter[1]`" as shown in the following example:

```
<#assign currentTitle = "Ch1">
<#list doc["book/chapter[title=$currentTitle]/para"] as p>
...
```

Note that `$currentTitle` is an XPath expression, not an RPL interpolation, as it is not enclosed in curly braces ({ }).

The result of some XPath expressions is not a node set, but a string, a number, or a boolean. For those XPath expressions, the result is an RPL string, number, or boolean variable, respectively. For example, the following code counts the total number of `para` elements in the XML document, so the result is a number:

```
${x["count(//para)"]}
```

and the output is:

```
5
```

About XML namespaces

By default, when you write `doc.book`, RPL selects the element with the name `book` that does not belong to any XML namespace (similarly to XPath). To select an element inside an XML namespace, you must register a prefix and use that prefix. For example, if the `book` element is in the XML namespace *`http://example.com/ebook`*, you have to associate a prefix with it at the top of the template with the `ns_prefixes` parameter of the `rpl` directive as shown in the following example:

```
<#rpl ns_prefixes={"e":"http://example.com/ebook"}>
```

Now, you can write expressions such as `doc["e:book"]`. Note that in this case, the usage of square brackets ([]) is required.

As the value of `ns_prefixes` is a hash, you can register multiple prefixes as shown in the following example:

```
<#rpl ns_prefixes={
    "e":"http://example.com/ebook",
    "f":"http://example.com/form",
    "vg":"http://example.com/vectorGraphics"}
>
```

The `ns_prefixes` parameter affects the entire RPL namespace. This means that the prefixes you have registered in the main template will be visible in all *<#include ...>-d* templates, but not in *<#imported ...>-d* templates (often referred to as RPL libraries). An RPL library can register XML namespace prefixes for its own use, without interfering with prefix registrations of the main template and other libraries.

Note that you can set a default namespace for an input document. In this case, if you do not use a prefix, as in `doc.book`, RPL selects the element that belongs to the default namespace. You set the default namespace using the reserved prefix `D`, for example:

```
<#rpl ns_prefixes={"D":"http://example.com/ebook"}>
```

Now, the expression `doc.book` will select the book element that belongs to the XML namespace *http://example.com/ebook*.

Note that XPath does not support default namespaces. Thus, in XPath expressions, element names without a prefix always apply to the elements that do not belong to any XML namespace.

However, to access elements in the default namespace, you can use the prefix `D`, for example:

```
doc["D:book/D:chapter[title='Ch1']"]
```

Note that when you use a default namespace, you can select elements that do not belong to any node namespace using the reserved prefix `N`, for example:

```
doc.book["N:foo"].
```

The above example does not work for XPath expressions. The equivalent for an XPath expression is:

```
doc["D:book/foo"]
```

Escaping

In HTML, certain characters such as < and & are reserved. This means that to print plain text in HTML output, you must use escaping as shown in the following example:

```
<#escape x as x?html>
<#assign book = doc.book>
<h1>${book.title}</h1>
<#list book.chapter as ch>
  <h2>${ch.title}</h2>
  <#list ch.para as p>
    <p>${p}
  </#list>
</#list>
</#escape>
```

in the HTML output of the above example, the book title *Romeo & Julia* will be printed correctly:

```
...
<h1>Romeo &amp; Julia</h1>
...
```

Formal description

Every variable that corresponds to a single node in the DOM tree is a multi-type variable of type node and type hash. Thus, you can use the node built-ins with them. Hash keys are interpreted as XPath expressions, with the exception of the special keys shown in the table below. Some node variables have a string type as well, so you can use them as string variables (they implement TemplateScalarModel).

Node type (?node_type)	Node name (?node_name)	String value (e.g. <p>${node})	Special hash keys
"document"	"@document"	No string value. (Error when you try to use it as string.)	"elementName", "prefix:elementName", "*", "**", "@@markup", "@@nested_markup", "@@text"
"element"	"name": the name of the element. This is the local name (i.e. name without namespace prefix).	If it has no element children, the text of all text node children concatenated together. Error otherwise, when you try to use it as string.	"elementName", "prefix:elementName", "*", "**", "@attrName", "@prefix:attrName", "@@", "@*", "@@start_tag", "@@end_tag", "@@attributes_markup", "@@markup", "@@nested_markup", "@@text", "@@qname"
"text"	"@text"	The text itself.	"@@markup", "@@nested_markup", "@@text"
"pi"	"@pi$target"	The part between the target name and the ?>.	"@@markup", "@@nested_markup", "@@text"
"comment"	"@comment"	The text of the comment, without the delimiters <!-- and -->.	"@@markup", "@@nested_markup", "@@text"
"attribute"	"name": the name of the attribute. This is the local name (i.e. name	The value of the attribute.	"@@markup", "@@nested_markup", "@@text", "@@qname"

Node type (?node_type)	Node name (?node_name)	String value (e.g. <p>${node})	Special hash keys
	without namespace prefix).		
"document_type"	"@document_type$name": name is the name of the document element.	No string value. (Error when you try to use it as string.)	"@@markup", "@@nested_markup", "@@text"

NOTES:

- There is no CDATA type. CDATA nodes are transparently considered as text nodes.
- Variables do not support ?keys and ?values.
- Element and attribute node names are local names, that is, they do not contain the namespace prefix. The URI of the namespace to which a node belongs can be queried with the ?node_namespace built-in.
- Variables are visible with XPath variable references (e.g. `doc["book/chapter[title=$currentTitle]"]`).

Meaning of special hash keys:

"elementName", "prefix:elementName"
Returns the sequence of child nodes that are elements of `elementName`. The selection is XML namespace-aware, unless the XML document was parsed with an XML parser that was not namespace-aware. In XML namespace-aware mode, names without a prefix (for example, `elementName`) select only elements that do not belong to any XML namespace (unless you have registered a default XML namespace), and names with a prefix (for example *`prefix`*`:elementName`) select only elements that belong to the XML namespace denoted by the prefix. You register prefixes and set the default XML namespace using the `ns_prefixes` parameter of the `rpl` directive.

"*"
Returns the sequence of all child element nodes. The sequence will contain the elements in the document order, that is, in the order in which the first character of the XML representation of each node occurs (after expansion of general entities).

"**"
Returns the sequence of all descendant element nodes. The sequence will contain the elements in the document order.

"@attName", "@prefix:attrName"
Returns the attribute `attName` of the element as a sequence of size one that contains the attribute node, or as an empty sequence if the attribute does not exist. To check whether an attribute exists, use `foo.@attName[0]??`, not `foo.@attName??`. As with the special key "elementName", if the length of the sequence is 1, then it also acts as its first sub-variable. If no prefix is used, then it returns only the attribute that does not use XML namespace (even if you have set a default XML namespace). If a prefix is used, it returns only the attribute that belongs to the XML namespace associated with the prefix. The registration of prefixes is done with the `ns_prefixes` parameter of the rpl directive.

"@@" or "@*"
Returns the sequence of attribute nodes belonging to the parent element. This is the same as XPath @*.

"@@qname"
Returns the fully qualified name of the element (such as e:book, in contrast to the local name returned by `?node_name`) . The prefix used is chosen based on the prefix registered in the current namespace with the `ns_prefixes` parameter of the `rpl` directive, and is not influenced by the prefix used in the source XML document. If you have set a default XML namespace, the prefix D is used for nodes that use that namespace. For nodes that do not belong to an XML namespace, no prefix is used even if a default namespace is set. If there is

no prefix registered for the namespace of the node, the result is a non-existent variable (node.@@qname?? is false).

"@@markup"

Returns the full XML markup of a node, as a string. Full XML markup means that it also contains the markup of the child nodes, and the markup of the children of the child nodes, and so on. The markup is not necessary the same as the markup in the source XML file, but it is semantically identical. Note that CDATA sections will be converted to plain text. Also note that depending on how the original XML document is enclosed with RPL, comment or processing instruction nodes might be removed and will be missing from the output. The first outputted start tag will contain `xmlns:prefix` attributes for each XML namespace used in the outputted XML fragment, and those prefixes will be used in the outputted element and attribute names. These prefixes will be the same as the prefixes registered with the `ns_prefixes` parameter of the `rpl` directive (no prefix will be used for D, as it will be registered as the default namespace with an `xmlns` attribute). If no prefix was assigned for a XML namespace, an arbitrarily chosen unused prefix will be used.

"@@nested_markup"

This is similar to `@@markup`, but it returns the XML markup of an element without its opening and closing tags. For the document node, it returns the same as `@@markup`. For other node types, it returns an empty string. Unlike with `@@markup`, no xmlns:prefix attributes will be placed into the output. The rules regarding the prefixes used in element and attribute names are the same as for `@@markup`.

"@@text"

Returns the value of all text nodes that occur within the node (all descendant text nodes, not only direct children), concatenated into a single string. If the node has no text node children, the result is an empty string.

"@@start_tag"
Returns the markup of the start tag of the element node. As with `@@markup`, the output is the semantic equivalent of the original XML document. For XML namespaces (xmlns:prefix attributes in the output, etc.), the rules are the same as for "@@markup".

"@@end_tag"
Returns the markup of the end tag of the element node. As with `@@markup`, the output is the semantic equivalent of the original XML document.

@@attributes_markup
Returns the markup of the attributes of the element node. As with `@@markup`, the output is the semantic equivalent of the original XML document.

About node sequences

Many of the special hash keys described in the list above and XPath expressions that result in node sets (see the XPath recommendation) return a sequence of nodes.

If these sequences store only one sub-variable, they also act as the sub-variable. For example, `${book.title[0]}` is the same as `${book.title}` if the `book` element has only one `title` element.

Returning an empty node sequence is common. For example, if the element `book` has no child element chapter, then `book.chapter` results in an empty node sequence. Note that this means that an invalid element, for example `book.chap` will also return an empty node sequence, and will produce an error.
Also, `book.chaptre??` (note the typo) will return true because the empty sequence exists, so you have to use `book.chaptre[0]??` instead.

Node sequences that store 0 or more than 1 node, also support the following hash keys:

"elementName", "prefix:elementName"

"@attrName", "@prefix:attrName"

"@@markup", "@@nested_markup"

"@@text"

"*", "**"

"@@", "@*"

When you apply one of those special keys on a node sequence that contains more than one or zero nodes, the special key is applied for each node in the same way as for single nodes, and the result is concatenated to form the final result. The result is concatenated in the order in which the nodes occur in the node sequence. The nodes are concatenated based on the type of the result. For example, if the special key would return a string for a single node, then the result for multiple nodes is a single string; if the special key would return a sequence for a single node, then result for multiple nodes is a single sequence. If you apply a special key to a sequence with zero nodes, the string result is an empty string or an empty sequence.

Note that you can use XPath expressions with node sequences.

Working with declarative XML processing

With *declarative processing*, you define how to handle the different types of nodes, and RPL walks the DOM tree and calls the handlers you defined.

This approach is useful for complex XML schemas, where the same element can occur as the child of many other elements. Examples of such schemas are XHTML and XDocBook.

With declarative approach, you use the `recurse` directive in most cases. This directive gets a node variable as a parameter, and visits all its children nodes, one after the other, starting with the first child. *Visiting* a node means that the directive calls a user-defined directive (like a macro) that has the same name as the name of the child node (`?node_name`). Then, that the user-defined directive handles the node which is available as special variable `.node`. For example, the following code:

```
<#recurse doc>

<#macro book>
  I'm the book element handler, and the title is: ${.node.title}
</#macro>
```

produces this output:

```
I'm the book element handler, and the title is: Test Book
```

If you call `recurse` without a parameter, then it uses `.node`. This means that it visits all children nodes of the node being handled. For example, the following code:

```
<#recurse doc>

<#macro book>
  Book element with title ${.node.title}
    <#recurse>
  End book
</#macro>

<#macro title>
  Title element
</#macro>

<#macro chapter>
  Chapter element with title: ${.node.title}
</#macro>
```

produces this output:

```
Book element with title Test Book
Title element
Chapter element with title: Ch1
Chapter element with title: Ch2
End book
```

The examples above show how to define handlers for element nodes, but not for text nodes. Since the name of the handler is the same as the name of nodes it handles, and the name of all text nodes is `@text`, you define the handler for text nodes as shown in the following example:

```
<#macro @text>${.node?html}</#macro>
```

`?html` is necessary because you have to HTML-escape the text, since you generate output of HTML format.

The following example shows a template that transforms the XML to complete HTML:

```
<#recurse doc>

<#macro book>
  <html>
    <head>
      <title><#recurse .node.title></title>
    </head>
    <body>
      <h1><#recurse .node.title></h1>
      <#recurse>
    </body>
  </html>
</#macro>

<#macro chapter>
  <h2><#recurse .node.title></h2>
  <#recurse>
</#macro>

<#macro para>
  <p><#recurse>
</#macro>

<#macro title>
  <#--
    We have handled this element imperatively,
    so we do nothing here.
  -->
</#macro>

<#macro @text>${.node?html}</#macro>
```

produce this output:

```
<html>
  <head>
    <title>Test Book</title>
  </head>
  <body>
    <h1>Test Book</h1>

    <h2>Ch1</h2>

    <p>p1.1

    <p>p1.2

    <p>p1.3

  <h2>Ch2</h2>

    <p>p2.1

    <p>p2.2

  </body>
</html>
```

Note that you can substantially reduce the amount of superfluous whitespace in the output by using the trim directives such as `<#t>`.

The declarative approach works well with XML schemas where any element might occur anywhere. For example, you need to changes text color to red in any element such as a title or a paragraph. To do this, add a macro that creates an element called *mark* as shown in the following example:

```
<#macro mark><font color=red><#recurse></font></#macro>
```

Now, you can use `<mark>...</mark>` anywhere.

For certain XML schemas, declarative XML processing results in shorter and much clearer RPL than imperative XML processing. It's up to you to decide which approach to use; don't forget that you can mix the two approaches. For example, you can use imperative approach in an element handler to process the contents of that element.

About default handlers

For some XML node types, RPL provides a default handler. This handler will handle the node if you do not define a handler for the node (i.e. if there is no user-defined directive available with the name identical to the node name). The following table lists the node types that the default handler supports and describes how it handles each node type:

Node type	Default Handler Action
Text node	Prints the text as it. Note that in most cases, this approach does not work well, because you should escape the text before sending it to the output (with `?html`, `?xml` or `?rtf`, etc. depending on the output format).
Processing instruction node	Calls a handler called `@pi` if you have created such user-defined directive. Otherwise, ignores the node.
Comment node Document type node	Ignores the node.
Document node	Calls the `recurse` directive, that is visits all children of the document node.

Element and attribute nodes are handled according to the usual XML-independent mechanism. That is, `@node_type` will be called as the handler. If `@node_type` is not defined, then an error terminates template processing. For element nodes, this means that you must define a macro or another kind of user-defined directive called `@element` to catch all element nodes. If you do not define an `@element` handler, you must define a handler for all possible elements.

Attribute nodes are not visited by the `recurse` directive, so you do not need to write handlers for them.

Visiting a single node

You can visit a single node instead of its children with the `visit` directive.

About XML namespaces

The name of the handler user-defined directive must be the fully qualified name of the element: *`prefix:elementName`*. The rules regarding the usage of prefixes is the same as with imperative processing. For example, the user-defined `book` directive handles only the book element that does not belong to any XML namespace (unless you specified a default XML namespace). If the example, it uses the XML namespace *http://example.com/ebook*:

```
<book xmlns="http://example.com/ebook">
...
```

The RPL will be:

```
<#rpl ns_prefixes={"e":"http://example.com/ebook"}>

<#recurse doc>

<#macro "e:book">
  <html>
    <head>
      <title><#recurse .node["e:title"]></title>
    </head>
    <body>
      <h1><#recurse .node["e:title"]></h1>
      <#recurse>
    </body>
  </html>
</#macro>

<#macro "e:chapter">
  <h2><#recurse .node["e:title"]></h2>
  <#recurse>
</#macro>

<#macro "e:para">
  <p><#recurse>
</#macro>

<#macro "e:title">
  <#--
    We have handled this element imperatively,
    so we do nothing here.
  -->
</#macro>

<#macro @text>${.node?html}</#macro>
```

Alternately, you can define a default XML namespace, then use `recurse` and the `book` macro as shown in the following example:

```
<#rpl ns_prefixes={"D":"http://example.com/ebook"}>

<#recurse doc>

<#macro book>
...
```

In this case, keep in mind that in XPath expressions, you must access the default XML namespace with an explicit D:. This is because in XPath, names without a prefix always refer to nodes with no XML namespace in XPath. Also note that as with imperative XML processing, the name of handlers for elements that have no XML namespace is `N:elementName` only if a default XML namespace is defined. However, for nodes that are not of type element (such as text nodes), you never use the N prefix in the handler name, because those nodes do not use XML namespaces. For example, the handler for text nodes is always just `@text`.

For more detailed information, see the `recurse` and visit `directives` in *"Chapter 6. Directive Reference"*.

XSL transformation processing

We recommend that you use XSL if you are familiar with its concepts and/or when more advanced processing is needed. Otherwise, imperative and declarative models of RPL suffice for most cases.

XSL transformation is used to to transform XML data into either other XML formats, HTML, or text.

XSL uses XPath as its matching engine, which is another standard in the industry. XSL is, in its most basic form, a matching engine of XML patterns directed by XPath selectors.   For more information on XPath, visit *http://www.w3.org/TR/xpath*.

XSL basics

XSL is written in XML format with use and support of namespaces. The style of writing XSL is by entering `xsl:template` elements that match a given criteria (an XML path, a condition for an XML path, etc.). The flow of code is directed by `xsl:apply-templates` elements placed within templates, in a parent-child manner. This is similar to the declarative XML processing where the `xsl:template` is equivalent to `<#macro…>` elements, while `xsl:apply-templates` is somewhat equivalent to `<#recurse>`. XSL is more powerful than the declarative model because it allows recursion in which the path selection is defined in a more specific way.

The XSL language supports `xsl:for-each`, `xsl:choose` and `xsl:if` as programmatic constructs. These constructs resemble the `<#list>`, `<#if>` and `<#switch>` directives. Note that XSL functionality are somewhat limited.

XSL processing in RPL

You use the `xslt` method for XSL processing. The `xslt` method receives two XML nodes, which you need to previously initialize with `parsexml`, and perhaps load, as described in the section *"Putting the XML into the data model"*. The first parameter is the node of the XML to be transformed. The second parameter is the node of the XSL transformation (which is, by definition, XML as well).

The following example assumes that the node `doc` has been initialized:

```
<#assign
transform=parsexml(load("cms://contentlibrary/transforms/transform.x
ml"))>
<#assign result=xslt(doc, transform)>
${result}
```

The following *transform.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" />

  <xsl:template match="/">
    <xsl:value-of select="book/title"/>
    <xsl:apply-templates select="book/chapter" />
  </xsl:template>

  <xsl:template match="chapter">[<xsl:value-of
select="title"/>]</xsl:template>

</xsl:stylesheet>
```

produces this output:

```
Test Book[Ch1][Ch2]
```

The first template matches the root of the XML. That template first obtains the title of the book, then applies further templates that match the chapters of the book. Another template that matches the chapter extracts the title of the chapter.

Working with XML output

Sometimes you need to transform from one XML format into another XML format. Consider the following example:

```
<#assign transform=parsexml(load("cms://transforms/transform.xml"))>
<#assign result=xslt(source, transform)>
${result}
```

with the following *transform.xml:*

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes"/>

  <xsl:template match="/book">
    <publication>
      <xsl:attribute name="name"><xsl:value-of
select="title"/></xsl:attribute>
      <index>
      <xsl:apply-templates select="chapter" />
      </index>
    </publication>
  </xsl:template>

  <xsl:template match="chapter">
    <section><xsl:value-of select="title"/></section>
    <xsl:apply-templates select="para" />
  </xsl:template>
  <xsl:template match="para">
    <section><xsl:value-of select="."/></section>
  </xsl:template>
</xsl:stylesheet>
```

produces this output:

```
<publication name="Test Book">
<index>
<section>Ch1</section>
<section>p1.1</section>
<section>p1.2</section>
<section>p1.3</section>
<section>Ch2</section>
<section>p2.1</section>
<section>p2.2</section>
</index>
</publication>
```

This example:

1. Matches the book at the root.
2. Encloses the output in a *<publication>* tag with an attribute called `name` with the contents of the `title` element.
3. Looks for all chapters and places them inside an `<index>` tag.
4. Puts a `<section>` tag, with the contents of the chapter title into the `chapters` template.
5. Rather than enclosing the children elements into another tag, the example places them at the same level as the chapter, and thus simply selects the `para` elements.
6. In the `para` template, creates a new `<section>` tag that encloses the value of the paragraph by obtaining the value of the `para` element (using the period syntax).

Note that the example RPL outputs the value of the `result` variable.

To process this as a node instead, with the familiar RPL syntax, you can modify the above example as follows:

```
<#assign
transform=parsexml(load("cms://transforms/booktransform.xml"))>
<#assign result=parsexml(xslt(source, transform))>
Publication Title: ${result.publication.@name}
<#list result.publication.index.section as sec>
${sec_index+1}. - ${sec}
</#list>
```

produces this output:

```
Publication Title: Test Book
1. - Ch1
2. - p1.1
3. - p1.2
4. - p1.3
5. - Ch2
6. - p2.1
7. - p2.2
```

This time, the resulting string is parsed XML with the `parsexml` method. After this, we are free to use the imperative processing model on the resulting XML.

Including additional transformation templates

XSL supports an `<xsl:include href="path"/>` instruction to add extra transformation templates as defined in an external file. For example, the following file called *root.xml:*

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="text"/>

  <xsl:include href="cms://transforms/included.xml"/>

</xsl:stylesheet>
```

includes an additional file called *included.xml*, which contains the following code:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

  <xsl:template match="/book ">[<xsl:value-of
select="title"/>]</xsl:template>

</xsl:stylesheet>
```

This is similar to an RPL `include` directive. In this mode, the `xsl:include` instruction allows full paths to `cms://`, `http://`, and `https://` sources.

This case supports only full paths, unlike the RPL `include` directive which supports both relative and absolute paths.

Chapter 5. Built-in Reference

Built-ins are divided into the following categories:

- Built-ins for strings
- Built-ins for numbers
- Built-ins for dates
- Built-ins for booleans
- Built-ins for sequences
- Built-ins for hashes
- Built-ins for XML nodes
- Additional built-ins

Built-ins for strings

base64

```
expr?base64
```

The string encoded in base64 format. This can be useful in obfuscating personal information to be used in a URL's query string.

Base64 is an encoding scheme that represents binary data in an ASCII string format by converting non-printable characters into printable characters. Base 64 strings are usually 30% longer that their source (16 bits get converted to 24 bits).

Example:

```
"Person"?base64
```

produces this output:

```
UGVyc29u
```

boolean

```
exp?boolean(true-string)
or
exp?boolean(true-string, false-string)
```

Converts the given expression string into its boolean counterpart.

If only `true-string` is given, any value that is not identical to it is considered false. If both `true-string` and `false-string` are given, the string should match either one, otherwise an error occurs.

This comparison is case-sensitive.

If the string is not in the appropriate format, an error terminates template processing when you try to access this built-in.

Example:

```
<#if isVip?boolean("true")>
We love to offer you this great discount!
</#if>
```

If `isVip` is `true`, the output is:

```
We love to offer you this great discount!
```

If the value is anything other than `true`, the returned value is false.

If the strings are stored as “yes” and “no”, you can use the following example:

```
<#if isVip?boolean("yes", "no")>
We love to offer you this great discount!
</#if>
```

In this last case, if the `isVip` value is neither “yes” nor “no”, an error occurs.

cap_first

```
exp?cap_first
```

The string with the very first word capitalized. For the exact meaning of “word”, see “*word_list built-in*”.

Example:

```
${"  green mouse"?cap_first}
${"GreEN mouse"?cap_first}
${"- green mouse"?cap_first}
```

produces this output:

```
  Green mouse
GreEN mouse
- green mouse
```

In the case of *- green mouse*, the first word is -.

capitalize

```
exp?capitalize
```

The string with the first letter of each word capitalized. For the exact meaning of "word", see "*word_list built-in*".

Example:

```
${"  green  mouse"?capitalize}
${"GreEN mouse"?capitalize}
```

produces this output:

```
  Green Mouse
Green Mouse
```

chop_linebreak

```
exp?chop_linebreak
```

If the string includes a line break, returns the string without the line break; otherwise returns the unchanged string.

contains

```
exp?contains(string-expr)
```

Returns if the string specified as the parameter occurs in the string.

Example:

```
<#if "piceous"?contains("ice")>It contains "ice"</#if>
```

produces this output:

```
It contains "ice"
```

date, time, datetime

```
exp?date
exp?time
exp?datetime
```

The string converted to a date value. We recommended including a parameter that specifies the format.

Example:

```
<#assign test1 = "10/25/1995"?date("MM/dd/yyyy")>
<#assign test2 = "15:05:30"?time("HH:mm:ss")>
<#assign test3 = "1995-10-25 03:05 PM"?datetime("yyyy-MM-dd hh:mm
a")>
${test1}
${test2}
${test3}
```

produces output similar to the following, depending on the output locale and other settings:

```
Oct 25, 1995
3:05:30 PM
Oct 25, 1995 3:05:00 PM
```

Note that the dates were converted back to string according to the `date_format`, `time_format`, and `datetime_format` settings. For more information about converting dates to strings, see *"String built-in for dates, date interpolations"*. It does not matter what format you used when you converted the strings to dates.

If you know what the default date/time/datetime format will be when the template is processed, you do not have to use the `format` parameter.

Example:

```
<#assign test1 = "Oct 25, 1995"?date>
<#assign test2 = "3:05:30 PM"?time>
<#assign test3 = "Oct 25, 1995 03:05:00 PM"?datetime>
${test1}
${test2}
${test3}
```

If the string is not in the appropriate format, an error terminates template processing when you try to access this built-in.

debug

```
string-exp?debug
```

Embeds debug information in a message.

If the `environment.debug` flag is on, the builtin returns the string provided. If the `environment.debug` flag is off, the builtin returns an empty string.

The `environment.debug` flag is true only for test and preview launches, and when the debug option was set in the Email Message Designer.

Example:

```
${(launch.type+':')?debug}Welcome to ShopCo
```

For a standard launch, regardless of the campaign debug flag setting, the example produces the following output:

```
Welcome to ShopCo.
```

For a proof launch, if the debug flas is set to “on” in the campaign, the example produces the following output:

```
proof:Welcome to ShopCo
```

For a preview launch, if the debug flag is set to “on” in the campaign, the example produces output similar to the following:

```
preview:Welcome to ShopCo
```

For more information, see `environment.debug` flag.

ends_with

```
exp?ends_with(string-expr)
```

Returns `true` if the string ends with the specified substring. For example `"redhead"?ends_with("head")` and `"head"?ends_with("head")` return boolean `true`.

eval

```
expr?eval
```

Evaluates the string in the expression. This built-in is similar to the `exec` built-in, except that it does not allow directives and executes the string as an expression. The result of the evaluation is of the proper type.

Example:

```
"1+2"?eval
```

produces this output:

```
3
```

exec

```
expr?exec
```

Uses the script specified in the expression and executes it like a template. This built-in is similar to the `eval` built-in, except that it allows directives. Expressions must be enclosed in `${}`. The result of this built-in is always a string.

Example:

```
<#assign x=3>
${r"${x+2}"?exec?number + 4>
```

produces this output:

```
9
```

The example first assign a variable x to the value of 3. It then uses a raw string so that it can use `${…}` without interpreting it. This is the script, in this case: `${x+2}`. The example then executes the script with the `exec` built-in. The result of this execution is the string 5, since it is the result of the interpolation. The example then converts the string to a number, and adds 4 to return 9.

groups

```
expr?groups[index]
```

Used only with the result of the `matches` built-in. For more information, see `matches` built-in.

hex

```
expr?hex
```

Converts a string into a hex-encoded string. For each character in the string, a two character representation of the byte in hexadecimal format is produced. This doubles the number of characters needed to represent the originating string.

Example:

```
“a sample string”?hex
```

produces this output:

```
612073616D706C6520737472696E67
```

To convert a number to hex encoding as described in this section, perform a two-step process as follows:

```
7371?string?hex
```

produces this output:

```
37333731
```

Note that this built-in applies to strings, and this mechanism is called *hex encoding*, as opposed to the hex representation of a number.

index_of

```
exp?index_of(string-expr)
```

Returns the index of the first occurrence of the specified substring within the string.

Example:

```
"abcabc"?index_of("bc")
```

Returns 1. This is because the index of the first character is 0.

You can also start the search from a specific index. For example:

```
"abcabc"?index_of("bc", 2)
```

Returns 4. There is no restriction on the numerical value of the second parameter. If it is negative, it is treated as 0. If it is greater than the length of the string, it is treated as equal to the length of this string. Decimal values are truncated to integers. If the substring does not occur in the string (starting from the given index, if you use the second parameter), the built-in returns -1.

html

```
exp?html
```

Replaces a string as HTML markup as shown below:

- < is replaced with &lt;
- > is replaced with &gt;
- & is replaced with &amp;
- " is replaced with &quot;

Note that to insert an attribute value securely, you must quote the attribute value in the HTML template with double quotation mark (") as shown in the following example:

```
<input type=text name=user value="${user?html}">
```

In HTML pages, you should use this built-in for all interpolations. This can spare a lot of typing and reduces the chance of mistakes using the `escape` directive.

interpret

```
exp?interpret
```

Interprets a string as an RPL template and returns a user-defined directive that - when applied to any block - executes the template as if it was included at that point. For example:

```
<#assign x=["a", "b", "c"]>
<#assign templateSource = r"<#list x as y>${y}</#list>">
<#-- Note: That r was needed so that the ${y} is not interpreted
above -->
<#assign inlineTemplate = templateSource?interpret>
<@inlineTemplate />
```

produces this output:

```
abc
```

`inlineTemplate` is a user-defined directive that, when executed, runs the template that was generated on-the-fly using `interpret`.

You can also apply this built-in to a two-element sequence. In this case, the first element of the sequence is the template source, and the second element is a name for the inline template. This can be useful to give an explicit name to the inline template for debugging purposes. For example, in the above template, you can write the following:

```
<#assign inlineTemplate = [templateSource,
"myInlineTemplate"]?interpret>
```

Note that giving the inline template a name has no immediate effect. It is useful only as extra information if an error occurs.

isnull (when used with a string value)

```
exp?isnull
```

Returns true if the string is an empty string (“”), otherwise it returns false. When dealing with database fields, NULL and empty string are equivalent. For this reason, this built-in is useful for expressions representing database fields.

Example:

```
<#if ""?isnull>Empty String is NULL<#else>Empty String is NOT
NULL</#if>
```

poduces this output:

```
Empty String is NULL
```

j_string

```
exp?j_string
```

Escapes the string with the escaping rules of Java string literals, making it safe to insert the value into a string literal. Note that the built-it does not add quotation marks around the inserted value, you should use it inside a string literal.

All characters under UCS code point 0x20 are escaped. When the characters have no dedicated escape sequence in Java (such as \n, \t, etc.), they will be replaced with a UNICODE escape (\uXXXX).

Example:

```
<#assign beanName = 'The "foo" bean.'>
String BEAN_NAME = "${beanName?j_string}";
```

produces this output:

```
String BEAN_NAME = "The \"foo\" bean.";
```

js_string

```
exp?js_string
```

Escapes the string with the escaping rules of JavaScript string literals, making it safe to insert the value into a string literal. Note that the built-it does not add quotation marks around the inserted value, you should use it inside a string literal.

Both the single quote (") and the double quote (') are escaped. In addition, the built-in escapes > as \> (to avoid </script>).

All characters under UCS code point 0x20 are escaped. When the characters have no dedicated escape sequence in JavaScript (such as \n, \t, etc.), they will be replaced with a UNICODE escape (\uXXXX).

Example:

```
<#assign user = "Big Joe's \"right hand\"">
<script>
  alert("Welcome ${user?js_string}!");
</script>
```

produces this output:

```
<script>
  alert("Welcome Big Joe\'s \"right hand\"!");
</script>
```

json_string

```
exp?json_string
```

Escapes the string with the escaping rules of JSON language string literals, making it safe to insert the value into a string literal. Note that the built-it does not add quotation marks around the inserted value, you should use it inside a string literal.

The built-in does not escape ' characters, since JSON strings must be quoted with ". However, the built-in does escape the slash (/)as \/ where they occur directly after a < to avoid </script>. The built-in also escape the > characters as

\u003E where they occur directly after `]]` to avoid exiting an XML CDATA section.

All characters under UCS code point 0x20 are escaped. When the characters have no dedicated escape sequence in JSON (such as \n, \t, etc.), they will be replaced with a UNICODE escape (\uXXXX).

last_index_of

```
exp?last_index_of(string-expr)
```

Returns the index of the last (rightmost) occurrence of the first character of the specified substring within a string.

Example:

```
"abcabc"?last_index_of("ab")
```

Returns 3.

You can specify the index from which to start the search.

Example:

```
"abcabc"?last_index_of("ab", 2)
```

Returns 0.

Note that the second parameter indicates the maximum index of the start of the substring. There is no restriction on the numerical value of the second parameter. If it is negative, it has the same effect as if it were zero, and if it is greater than the length of this string, it has the same effect as if it were equal to the length of the string. Decimal values are truncated to integers.

If the first parameter does not occur as a substring in the string (before the given index, if you use the second parameter), the built-in returns -1.

left_pad

```
exp?left_pad(numeric-expr)
```

```
or
exp?left_pad(numeric-expr, string-expr)
```

When used with one parameter, the built-in inserts spaces at the beginning of the string until the string reaches the length specified by the parameter. For example:

```
[${""?left_pad(5)}]
```

inserts 5 spaces, making the string 5 characters long.

The following example:

```
[${"a"?left_pad(5)}]
```

inserts 4 spaces before the a, making the string 5 characters long.

If the string is already as long or longer than the specified number, the built-in does nothing. For example:

```
[${"abcdefgh"?left_pad(5)}]
```

produces this output:

```
 [abcdefgh]
```

If you specify two parameters, the built-in inserts the characters specified by the second parameter at the beginning of the string until the string reaches the length specified by the first parameter.

The following example:

```
[${""?left_pad(5, "-")}]
```

inserts 5 dashes (-), making the string 5 characters long.

The following example:

```
[${"a"?left_pad(5, "-")}]
```

inserts 4 dashes before the a, making the string 5 characters long.

If the string is already as long as or longer than the specified number, the built-in does nothing. For example:

```
[${"abcde"?left_pad(5, "-")}]
```

produces this output:

```
 [abcde]
```

If the second parameter is longer than one character, the following example:

```
 [${""?left_pad(8, ".oO")}]
```

produces this output:

```
 [.oO.oO.o]
```

length

```
exp?length
```

The number of characters in the string.

matches

```
expr?matches(regex-string-expr)
expr?groups[index]
```

NOTE: Use this built-in only if you are familiar with regular expressions.

Determines whether the string matches the pattern exactly and returns the list of matching sub-strings. The return value is a multi-type value:

- Boolean: `true`, if the string exactly matches the pattern; `false` otherwise. For example, `"fooo"?matches('fo*')` is true, but `"fooo bar"?matches('fo*')` is false.
- Sequence: the list of matched substrings of the string. Possibly a 0 length sequence.

Example:

```
<#if "fxo"?matches("f.?o")>Matches.<#else>Does not match.</#if>

<#assign res = "foo bar fyo"?matches("f.?o")>
<#if res>Matches.<#else>Does not match.</#if>
Matching sub-strings:
<#list res as m>
- ${m}
</#list>
```

produces this output:

```
Matches.

Does not match.
Matching sub-strings:
- foo
- fyo
```

If the regular expression contains groups (parentheses), you can access them with the `groups` built-in. For example:

```
<#assign res = "aa/rx; ab/r;"?matches("(\\w[^/]+)/([^;]+);")>
<#list res as m>
- ${m} is ${m?groups[1]} per ${m?groups[2]}
</#list>
```

produces this output:

```
- aa/rx; is aa per rx
- ab/r; is ab per r
```

This built-in accepts an optional second parameter, *`flags`*. Note that it does not support the flag `f`, and ignores the flag `r`. For more information, see “Common flags for sting built-ins”.

number

```
expr?number
```

The string converted to numerical value. The number must be in the same format as the numerical values specified directly in RPL: it must be in the locale-independent format, where the decimal separator is a dot. In addition, the built-in recognizes scientific notation (e.g. "1.23E6", "1.5e-8").

If the string is not in the appropriate format, an error occurs that terminates template processing when you try to access this built-in.

lower_case

```
exp?lower_case
```

The lower case version of the string.

Example:

```
"GrEeN MoUsE"?lower_case
```

produces this output:

```
"green mouse".
```

replace

```
expr?replace(string-exp-to-find, string exp-new-string)
expr?replace(string-exp-to-find, string exp-new-string, flag)
```

Replaces all occurrences of a substring with another string.

This built-in does not handle word boundaries.

Example:

```
${"this is a car acarus"?replace("car", "bulldozer")}
```

produces this output:

```
this is a bulldozer abulldozerus
```

The replacing occurs left-to-right. This means that the following example:

```
${"aaaaa"?replace("aaa", "X")}
```

produces this output:

```
Xaa
```

If the first parameter is an empty string, all occurrences of the empty string are replaced. For example:

```
 "foo"?replace("","|")
```

is replaced with:

```
 "|f|o|o|"
```

This built-in accepts an optional parameter, *flags*, as a third parameter. For more information, see “Common flags for sting built-ins”.

right_pad

```
exp?left_pad(numeric-expr)
or
exp?left_pad(numeric-expression, string-expr)
```

Inserts a specified number of spaces or characters at the end of a string.

If you specify one parameter, the built-in inserts spaces at the end of the string until the string reaches the length specified by the parameter. For example:

```
[${""?right_pad(5)}]
```

inserts 5 spaces, which makes the string 5 characters long.

This example:

```
[${"a"?right_pad(5)}]
```

inserts 4 spaces after the a, making the string 5 characters long.

If the string is already as long or longer than the specified number, the built-in does nothing. For example, this code:

```
[${"abcdefgh"?right_pad(5)}]
```

produces this output:

```
 [abcdefgh]
```

If you specify two parameters, the built-in inserts the characters specified by the second parameter at the end of the string until the string reaches the length specified by the first parameter.

The second parameter must be a string value, and must be at least one character long.

Example:

```
[${""?right_pad(8, ".oO")}]
```

produces this output:

```
 [.oO.oO.o]
```

rtf

```
expr?rtf
```

The string as Rich text (RTF text). That is, the string with all:

- \ replaced with \\
- { replaced with \{
- } replaced with \}

split

```
expr?split(string-expr)
```

Splits a string into a sequence of strings.

The built-in assumes that the separator always occurs before a new item. For example, this code:

```
<#list "some,,test,text,"?split(",") as x>
- "${x}"
</#list>
```

produces this output:

```
- "some"
- ""
- "test"
- "text"
- ""
```

The built-in accepts an optional flags parameter, `flags`, as a second parameter. For more information, see *“Common flags for sting built-ins”*.

Example:

```
<#list "someMOOtestMOOtext"?split("MOO") as x>
- ${x}
</#list>
```

produces this output:

```
- some
- test
- text
```

starts_with

```
expr?starts_with(string-expr)
```

Checks whether a string begins with the specified substring.

The built-in returns `true` if the string begins with the substring.

For example, both of the following return `true`:

```
 "redhead"?starts_with("red")
 "red"?starts_with("red")
```

skip

```
expr?skip
or
expr?skip(message-str)
```

A skip is the action of ignoring the current recipient during personalization. This built-in checks whether the value of the field has a null or empty string value, and if it does, causes a skip. In a skip, the email is not sent to the current recipient. The processing of the current record is stopped immediately and control returns to the personalization engine.

Example:

Assume that the *profile.firstname* field in the database contain the value of ""

```
Hello ${profile.firstname?skip}

Thanks for your recent visit to our site.
```

Since the value of *profile.firstname* is empty, the current recipient record is skipped. This is a short form of the skip directive. The previous example could also be written as:

```
<#if profile.firstname == "">
  <#skip "String is empty">
</#if>
Hello ${profile.firstname}

Thanks for your recent visit to our site.
```

string (when used with a string value)

```
expr?string
```

Returns the string as is. The exception is that if the value is a multi-type value (e.g. both a string and a sequence), the resulting value is a simple string, not a multi-type value. This can be utilized to prevent the effects of multi-typing.

substring

```
exp?substring(from, toExclusive)
or
exp?substring(from)
```

Returns a substring of the string, starting with the character specified by the `from` parameter and ending with the character before the `toExclusive` parameter.

`from` must be a number that is at least 0 and less than or equal to `toExclusive`; otherwise, an error occurs that terminates template processing.

`toExclusive` is the index of the character immediately after the last character to retrieve. It must be a number that is at least 0 and less than or equal to the length of the string; otherwise and error occurs that terminates template processing. If the `toExclusive` is omitted, it defaults to the length of the string. If a parameter is a number that is not an integer, only the integer part of the number is used.

Example:

```
- ${'abc'?substring(0)}
- ${'abc'?substring(1)}
- ${'abc'?substring(2)}
- ${'abc'?substring(3)}

- ${'abc'?substring(0, 0)}
- ${'abc'?substring(0, 1)}
- ${'abc'?substring(0, 2)}
- ${'abc'?substring(0, 3)}

- ${'abc'?substring(0, 1)}
- ${'abc'?substring(1, 2)}
- ${'abc'?substring(2, 3)}
```

produces this output:

```
- ab c
- bc
- c
-

-
- a
- ab
- abc

- a
- b
- c
```

trim

```
expr?trim
```

The string without leading and trailing white space. For example:

```
  (${"   green mouse   "?trim})
```

produces this output:

```
  (green mouse)
```

uncap_first

```
exp?uncap_first
```

Un-capitalizes the first word of the string.

upper_case

```
expr?upper_case
```

Returns the upper case version of the string. For example:

```
"GrEeN MoUsE"?upper_case
```

produces this output:

```
"GREEN MOUSE".
```

url

```
expr?url
```

Returns the string after URL escaping. This means that all non-US-ASCII and reserved URL characters are escaped with %XX. For example:

```
<#assign x = 'a/b c'>
${x?url}
```

Assuming that the charset used for escaping is an US-ASCII compatible charset, the example produces this output:

```
a%2Fb%20c
```

The built-in escapes all reserved URL characters (/, =, &, ...etc.), so this encoding can be used for encoding query parameter values, for example:

```
<a href="foo.cgi?x=${x?url}&y=${y?url}">Click here...</a>
```

Note that in the above example, no HTML encoding (?htm) was needed, because URL escaping escapes all reserved HTML characters. You should always quote the attribute value with double quotes ("), never with single quotes (') because the single quote is not escaped by URL escaping.

You must select a charset to be used for calculating the escaped parts (%XX). If you do not select a charset, RPL uses a default charset. To set a charset, specify it in the `url_escaping_charset` setting that can be set in template execution time. For example:

```
<#--
  This will use the charset specified by the system
  before the template execution has started.
-->
<a href="foo.cgi?x=${x?url}">foo</a>

<#-- Use UTF-8 charset for URL escaping from now: -->
<#setting url_escaping_charset="UTF-8">

<#-- This will surely use UTF-8 charset -->
<a href="bar.cgi?x=${x?url}">bar</a>
```

In addition, you can specify a charset explicitly for a single URL escaping as a parameter:

```
<a href="foo.cgi?x=${x?url('ISO-8895-2')}">foo</a>
```

If you do not specify the parameter, the built-in uses the charset specified as the value of the `url_escaping_charset` setting as set by the system.

word_list

```
expr?word_list
```

Returns a sequence that contains all words of the string in the order they appear in the string.

Words are continual character sequences that contain any character except white space. For example:

```
<#assign words = "   a bcd, .   1-2-3"?word_list>
<#list words as word>[${word}]</#list>
```

produces this output:

```
 [a][bcd,][.][1-2-3]
```

xhtml

```
expr?xhtml
```

The string as XHTML text. That is, the string with all:

< is replaced with &lt;

> is replaced with &gt;

& is replaced with &amp;

" is replaced with &quot;

' is replaced with &#39;

NOTE: The only difference between this built-in and the `xml` built-in is that the `xhtml` built-in escapes ' as &#39; instead of as &apos;. This is because some older browsers do not interpret &apos; correctly.

xml

```
expr?xml
```

The string as XML text. That is, the string with all:

< is replaced with &lt;

> is replaced with &gt;

& is replaced with &amp;

" is replaced with &quot;

' is replaced with &apos;

Common flags

Many string built-ins accept an optional string parameter, called `flag`.

Each letter in the flag affects a certain aspect of built-in behaviour. For example, the letter *i* means that the built-in should not differentiate between the lower and upper-case variation of the same letter.

You may use the flags in any order.

Supported flags

The following table lists all supported flags:

Flag	Description
i	Case insensitive: do not differentiate the lower and upper-case variation of the same letter.
f	First only. That is, replace/find/etc. only the first occurrence of something.
r	The substring to find is a regular expression. RPL uses the variation of regular expressions described below.
m	Multi-line mode for regular expressions. In multi-line mode the expressions ^ and $ match just after or just before, respectively, a line terminator or the end of the string. By default these expressions only match at the beginning and the end of the entire string. Note that ^ and

Flag	Description
	$ do not match the line-break character itself.
s	Enables dot-all mode for regular expressions (same as Perl single-line mode). In dot-all mode, the expression . matches any character, including a line terminator. By default, this expression does not match line terminators.
c	Permits whitespace and comments in regular expressions.

Example:

```
<#assign s = 'foo bAr baar'>
${s?replace('ba', 'XY')}
i: ${s?replace('ba', 'XY', 'i')}
if: ${s?replace('ba', 'XY', 'if')}
r: ${s?replace('ba*', 'XY', 'r')}
ri: ${s?replace('ba*', 'XY', 'ri')}
rif: ${s?replace('ba*', 'XY', 'rif')}
```

produces this output:

```
foo bAr XYar
i: foo XYr XYar
if: foo XYr baar
r: foo XYAr XYr
ri: foo XYr XYr
rif: foo XYr baar
```

Supported flags by built-in

The following table lists all built-ins that support flags and shows which flags each one supports.

Built-in name	Flags					
	c	f	i	m	r	s
replace	Only with r	Yes	Yes	Only with r	Yes	Only with r
split	Only with r	No	Yes	Only with r	Yes	Only with r
match	Yes	No	Yes	Yes	Ignored	Yes

Built-ins for numbers

c

```
expr?c
```

Converts a number to a string which is independent of any locale and number format settings of RPL.

This built-in is crucial because by default, numbers are converted to strings with the locale-specific number formatting. When the number is not meant for users (for example, for a database record ID used as the part of an URL, or as an invisible field value in an HTML form), you must use this built-in to print the number (i.e., use ${x?c} instead of ${x}); otherwise the output might be incorrect, depending on the current number formatting settings and locale, and the value of the number.

The built-in always uses the dot as decimal separator.

The built-in never uses any of the following:

- Grouping separators (such as 3,000,000)
- Exponential form (such as 5E20)
- Superfluous leading or trailing zeros (such as 03 or 1.0)
- Plus sign (for example +1)

The built-in prints at most 16 digits after the decimal dot, thus numbers whose absolute value is less than 1E-16 will be shown as 0.

hex

```
expr?hex
```

Converts a number to its hexadecimal representation, without padding. To add padding, convert the number to a string, then use the `left_pad` built-in for strings.

Example:

```
<#assign x=32>
${x}
${x?hex}
0x${x?hex?string?left_pad(8, "0")}
```

produces this output:

```
32
20
0x00000020
```

isnull (when used with a numerical value)

```
exp?isnull
```

Useful only for expressions representing fields from the database.

When a numeric field in the database is NULL, RPL internally converts it to a zero, but temporarily retains the fact that this field came from a NULL value. This built-in returns true if the number came from a NULL value, otherwise returns false. For non-database fields, the built-in always returns false. When an operation that involves a database field is performed (for instance adding a field to a number), the result does not retain whether the value is NULL in the database.

The following example shows a field in the database that results in a NULL:

```
<#-- NULL is remembered -->
<#if profile.nullnumber?isnull>
     profile.nullnumber is NULL
<#else>
     profile.nullnumber is NOT NULL
</#if>

<#-- but zero is assumed, so zero plus one is one -->
${profile.nullnumber+1}

<#-- the result is no longer null -->
<#if (profile.nullnumber+0)?isnull>
     profile.nullnumber+0 is NULL
<#else>
     profile.nullnumber+0 is NOT NULL
</#if>
```

produces this output:

```
profile.nullnumber is NULL
1
profile.nullnumber+0 is NOT NULL
```

round, floor, ceiling

```
expr?round
expr?floor
expr?ceiling
```

Converts a number to a whole number using the specified rounding rule:

`round`

Rounds to the nearest whole number. If the number ends with .5, then it rounds up (i.e., towards positive infinity).

`floor`

Rounds the number down (i.e., towards negative infinity).

`ceiling`

Rounds the number up (i.e., towards positive infinity).

These built-ins are useful for pagination and similar operations. If you want to display numbers in rounded form, use the `string` built-in or the `number_format` setting.

Example:

```
<#assign testlist=[
  0, 1, -1, 0.5, 1.5, -0.5,
  -1.5, 0.25, -0.25, 1.75, -1.75]>
<#list testlist as result>
    ${result} ?floor=${result?floor} ?ceiling=${result?ceiling}
?round=${result?round}
</#list>
```

produces this output:

```
     0 ?floor=0 ?ceiling=0 ?round=0
     1 ?floor=1 ?ceiling=1 ?round=1
     -1 ?floor=-1 ?ceiling=-1 ?round=-1
     0.5 ?floor=0 ?ceiling=1 ?round=1
     1.5 ?floor=1 ?ceiling=2 ?round=2
     -0.5 ?floor=-1 ?ceiling=0 ?round=0
     -1.5 ?floor=-2 ?ceiling=-1 ?round=-1
     0.25 ?floor=0 ?ceiling=1 ?round=0
     -0.25 ?floor=-1 ?ceiling=0 ?round=0
     1.75 ?floor=1 ?ceiling=2 ?round=2
     -1.75 ?floor=-2 ?ceiling=-1 ?round=-2
```

string (when used with a numerical value)

```
expr?string
expr?string(mask-expr)
```

Converts a number to a string.

This built-in uses the default format specified by the system. You can also specify a number format explicitly.

RPL supports four predefined number formats: computer, currency, number, and percent. The exact meaning of currency, number, and percent is locale-specific, and is controlled by the Java platform installation rather than by RPL. The computer format uses the same formatting as the `c` built-in. You can use these predefined

formats as shown in the following example:

```
<#assign x=42>
${x}
${x?string}  <#-- the same as ${x} -->
${x?string.number}
${x?string.currency}
${x?string.percent}
${x?string.computer}
```

If your locale is US English, the example produces the following output:

```
42
42
42
$42.00
```

```
4,200%
42
```

The output of the first three expressions is identical because the first two expressions use the default format, which is "number" here. You can change this default using a setting:

```
<#setting number_format="currency">
<#assign x=42>
${x}
${x?string}  <#-- the same as ${x} -->
${x?string.number}
${x?string.currency}
${x?string.percent}
```

Because the default number format was set to "currency", the example now produces this output:

```
$42.00
$42.00
42
$42.00
4,200%
```

In addition to the three predefined formats, you can use an arbitrary number format patterns written in the standard decimal number format syntax as:

```
<#assign x = 1.234>
${x?string("0")}
${x?string("0.#")}
${x?string("0.##")}
${x?string("0.###")}
${x?string("0.####")}

${1?string("000.00")}
${12.1?string("000.00")}
${123.456?string("000.00")}

${1.2?string("0")}
${1.8?string("0")}
${1.5?string("0")} <-- 1.5, rounded towards even neighbor
${2.5?string("0")} <-- 2.5, rounded towards even neighbor

${12345?string("0.##E0")}
```

produces this output:

```
1
1.2
1.23
1.234
1.234

001.00
012.10
123.46

1
2
2 <-- 1.5, rounded towards even neighbor
2 <-- 2.5, rounded towards even neighbor

1.23E4
```

Following the financial and statistics practice, RPL rounds according the "half-even" rule. This means rounding towards the nearest ``neighbor'', unless both neighbors are equidistant. If both neighbors are equidistant, RPL rounds to the even neighbor. This rule is illustrated in the above example: 1.5 and 2.5 are both rounded to 2 is even, but 1 and 3 are odds.

In addition to the standard decimal syntax patterns, you can use `${aNumber?string("currency")}` and similar formatting. This produces the same result as `${aNumber?string.currency}`, etc.

As with predefined formats, you can set the default number formatting in the template. For example:

```
<#setting number_format="0.##">
${1.234}
```

produces this outputs:

```
1.23
```

Note that the number formatting is locale sensitive. For example:

```
<#setting locale="en_US">
In US they write:        ${12345678?string(",##0.00")}
<#setting locale="hu">
In Hungary they write: ${12345678?string(",##0.00")}
```

produces this output:

```
In US they write:        12,345,678.00
In Hungary they write: 12 345 678,00
```

Built-ins for dates

date, time, datetime (when used with a date value)

```
expr?date
expr?time
expr?datetime
```

You can use these built-ins to specify which parts of the date variable are in use:

`date`

Only the year, month and day parts are used.

`time`

Only the hour, minute, second and millisecond parts are used.

`datetime`

Both the date and the time parts are used.

Due to technical limitations, RPL cannot always determine which parts of the date are in use. If RTL has to execute an operation where this information is needed, such as displaying the date as text, but cannot determine which parts are in use, it will produce an error that terminates template processing. In such cases, you have to use these built-ins.

The following example illustrates a potentially problematic variable, *openingTime*:

```
<#assign x = openingTime> <#-- no problem can occur here -->
${openingTime?time} <#-- without ?time it would fail -->
<#-- For the sake of better understanding, consider this: -->
<#assign openingTime = openingTime?time>
${openingTime} <#-- this will work now -->
```

Additionally, you might use these built-ins to truncate dates. For example:

```
Last updated: ${lastUpdated} <#-- assume that lastUpdated is a date-
time value -->
Last updated date: ${lastUpdated?date}
Last updated time: ${lastUpdated?time}
```

produces this output:

```
Last updated: 04/25/2003 08:00:54 PM
Last updated date: 04/25/2003
Last updated time: 08:00:54 PM
```

If the left side of the ? is a string, then these built-ins convert strings to date variables.

isnull (when used with a date value)

```
exp?isnull
```

This built-in is useful only for expressions representing fields from the database.

When a date field in the database is NULL, RPL converts it to a date representing January 1st, 1800 in the current time zone, but the fact that this field came from a NULL value is temporarily retained. This built-in returns true if the date came from a NULL value, otherwise returns false. For non-database fields, the built-in always returns false. When an operation that involves a database field is performed (for instance adding a day with `dayadd`) the result does not retain whether it came from a NULL value from the database.

The default value for a NULL data field is January 1st, 1800 in the **current timezone** of the template execution.

Example:

The following field in the database results in a NULL:

```
<#-- NULL is remembered -->
<#if profile.nulldate?isnull>
      profile.nulldate is NULL
<#else>
      profile.nulldate is NOT NULL
</#if>

<#-- 1800-01-01 00:00:00 is the default-->
${dayadd(profile.nulldate,1)?string("yyyy-MM-dd HH:mm:ss")}

<#-- the result is no longer null -->
<#if dayadd(profile.nullnumber,0)?isnull>
      profile.nulldate+0 is NULL
<#else>
      profile.nulldate+0 is NOT NULL
</#if>
```

produces this output:

```
profile.nulldate is NULL
1800-01-02 00:00:00
profile.nulldate+0 is NOT NULL
```

iso_...

```
expr?iso...
```

Converts a date, time or date-time value to a string that follows ISO 8601 "extended" format. The name is constructed from the following words, in the order shown, separated by a _:

1. `iso` (required)

2. Either `utc` or `local` (required, except when used with parameters as described below).

   Specifies whether you want to print the date according to UTC or the current time zone. The current time zone is determined by the `time_zone` RPL setting and is normally configured by the system outside the templates. Note that the current time zone can also be set in a template, for example `<#setting time_zone="America/New_York">`.

   Omitting utc or local

   Instead of specifying `utc` or `local`, you can specify the time zone as a parameter.

   The parameter can also be a java.util.TimeZone object (which might be a return value of a Java method), or might be in the data-model.

   If RPL cannot interpret the time zone parameter, an error will occur that terminates template processing.

3. Either `h`, `m`, or `ms` (optional)

   The precision of the time part. When omitted, defaults to seconds (for example 12:30:18). `h` means hours precision (in this case 12), `m` means minutes precision (in this case, 12:30), and `ms` means milliseconds precision (in this case, 12:30:18.25, for 250 milliseconds). Note that when using `ms`, the milliseconds are displayed as a fraction of a second and trailing zeroes are omitted. This means that if the millisecond is 0, the millisecond will be omitted (for example, 12:30:18). The fraction is always separated with a dot to follow the Web conventions and the XML Schema date/time format.

4. `nz` (optional)

   When present, the time zone offset , such as +02:00 or -04:30 or Z, will not be displayed. Otherwise, the time zone offset will be displayed except for date-only values (because dates with zone offset does not appear in ISO 8601:2004).

Since ITL 2.3.19, the offset always contains the minutes for XML Schema date/time format compliance. For example:

```
<#assign aDateTime = .now>
<#assign aDate = aDateTime?date>
<#assign aTime = aDateTime?time>

Basic formats:
${aDate?iso_utc}
${aTime?iso_utc}
${aDateTime?iso_utc}

Different accuracies:
${aTime?iso_utc_ms}
${aDateTime?iso_utc_m}

Local time zone:
${aDateTime?iso_local}
```

produces output similar to this, depending on current time and time zone:

```
Basic formats:
2011-05-16
21:32:13Z
2011-05-16T21:32:13Z

Different accuracies:
21:32:13.868Z
2011-05-16T21:32Z

Local time zone:
2011-05-16T23:32:13+02:00
```

The following example illustrates the use of the built-in with `utc` or `local` omitted:

```
<#assign aDateTime = .now>
${aDateTime?iso("UTC")}
${aDateTime?iso("GMT-02:30")}
${aDateTime?iso("Europe/Rome")}

The usual variations are supported:
${aDateTime?iso_m("GMT+02")}
${aDateTime?iso_m_nz("GMT+02")}
${aDateTime?iso_nz("GMT+02")}
```

produces output similar to this, depending on current time and time zone:

```
2011-05-16T21:43:58Z
2011-05-16T19:13:58-02:30
2011-05-16T23:43:58+02:00

The usual variations are supported:
2011-05-16T23:43+02:00
2011-05-16T23:43
2011-05-16T23:43:58
```

long

```
expr?long
```

You can use this built-in with date, time and date-time values to get the number of milliseconds since January 1, 1970, 00:00:00 GMT (also known as the *unix epoch date*).

string (when used with a date value)

```
expr?string
expr?string.short
expr?string.medium
expr?string.long
expr?string.full
…
expr?string(mask-expr)
```

Converts a date to a string with the specified formatting.

Tip: You do need to use this built-in if you want to use the default format specified by the `date_format`, `time_format`, and `datetime_format` settings. The format can be one of the predefined formats, or you can specify the formatting pattern explicitly.

The predefined formats are *`short`*, *`medium`*, *`long`*, and *`full`*. These formats define how verbose the resulting text will be. For example, if the locale of the output is U.S. English, and the time zone is the U.S. Pacific Time zone, the following example:

```
${openingTime?string.short}
${openingTime?string.medium}
${openingTime?string.long}
${openingTime?string.full}

${nextDiscountDay?string.short}
${nextDiscountDay?string.medium}
${nextDiscountDay?string.long}
${nextDiscountDay?string.full}

${lastUpdated?string.short}
${lastUpdated?string.medium}
${lastUpdated?string.long}
${lastUpdated?string.full}
```

produces this output:

```
12:45 PM
12:45:09 PM
12:45:09 PM CEST
12:45:09 PM CEST

4/20/07
Apr 20, 2007
April 20, 2007
Friday, April 20, 2007

4/20/07 12:45 PM
Apr 20, 2007 12:45:09 PM
April 20, 2007 12:45:09 PM CEST
Friday, April 20, 2007 12:45:09 PM CEST
```

The exact meaning of short, medium, long, and full depends on the current locale (language). Furthermore, it is specified by the Java platform implementation on which you run RPL.

For dates that contain both a date and a time, you can specify the length of the date and time parts independently, as shown in the following example:

```
${lastUpdated?string.short_long} <#-- short date, long time -->
${lastUpdated?string.medium_short} <#-- medium date, short time -->
```

produces this output:

```
4/8/03 9:24:44 PM PDT
Apr 8, 2003 9:24 PM
```

Note that `?string.short` is the same as `?string.short_short`, `?string.medium` is the same as `?string.medium_medium`, etc.

WARNING: Due to technical limitations, in some cases RPL cannot determine whether the variable stores only date (year, month, day), only time (hour, minute, second, millisecond), or both. In such cases, RPL will stop with an error when you write something similar to `${lastUpdated?string.short}` or `${lastUpdated}`. To prevent such errors, we recommend that you always use the `?date`, `?time`, and `?datetime` built-ins, for example: `${lastUpdated?datetime?string.short}`.

Instead of using the predefined formats, you can specify the formatting pattern explicitly with `?string(`*`pattern_string`*`)`. The pattern uses the Java date format syntax explained in the *"About the* date format*"* section. For example:

```
${lastUpdated?string("yyyy-MM-dd HH:mm:ss zzzz")}
${lastUpdated?string("EEE, MMM d, ''yy")}
${lastUpdated?string("EEEE, MMMM dd, yyyy, hh:mm:ss a '('zzz')'")}
```

produces this output:

```
2003-04-08 21:24:44 Pacific Daylight Time
Tue, Apr 8, '03
Tuesday, April 08, 2003, 09:24:44 PM (PDT)
```

NOTE: With explicitly given patterns, you do not need to use `?date`, `?time`, and `?datetime`. This is because the pattern specifies which parts of the date to show. However, note that you can show "noise" if you display parts that are not stored in the variable. For example, `${openingTime?string("yyyy-MM-dd hh:mm:ss a")}`, where `openingTime` stores only time, will display 1970-01-01 09:24:44 PM.

The pattern string also can be "short", "medium", ..., "short_medium", ...etc. These are the same as using the predefined formats with the dot syntax: `someDate?string("short")` and `someDate?string.short` are equivalent.

Built-ins for booleans

string (when used with a boolean value)

```
expr?string
expr?string(true-expr, false-expr)
```

Converts a boolean to a string. You can use it in two ways:

foo?string
Converts the boolean to string using the default strings for representing true and false values. By default, true is rendered as `true` and false is rendered as `false`. To change these default strings, you can use the `boolean_format` setting. Note that if the variable is a multi-type variable that is both boolean and string, the string value of the variable is returned.

foo?string("yes", "no")
Returns the first parameter (here: "yes") if the boolean is true; otherwise the second parameter (here: "no"). Note that the return value is always a string. If the parameters are numbers, they are first converted to strings.

Built-ins for sequences

chunk

```
expr?chunk(number-expr)
expr?chunk(number-expr, string expr)
```

Splits a sequence into multiple sequences of the size given with the first parameter to the built-in (for example, `mySeq?chunk(3)`). The result is the sequence of these sequences. The last sequence might be shorter than the given size, unless the second parameter is given (for example, `mySeq?chunk(3, '-')`), that is the item used to make up the size of the last sequence to the given size.

For example:

```
<#assign seq = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']>

<#list seq?chunk(4) as row>
  <#list row as cell>${cell} </#list>
</#list>

<#list seq?chunk(4, '-') as row>
  <#list row as cell>${cell} </#list>
</#list>
```

produces this output:

```
  a b c d
  e f g h
  i j

  a b c d
  e f g h
  i j - -
```

This built in is used mostly for outputting sequences in tabular/columnar format. When used with HTML tables, the second parameter is often "\xA0" (that is the code of the no-break space character, also known as *nbsp*), so the border of the empty TDs will not be missing.

The first parameter must be a number that is at least 1. If the number is not an integer, it will be rounded down to integer (i.e. both 3.1 and 3.9 will be rounded to 3). The second parameter can be of any type and value.

first

```
expr?first
```

Returns the first sub-variable of the sequence.

If the sequence is empty, an error occurs that terminates template processing.

join

```
expr?join(separator-expr
or
expr?join(separator-expr, last-separator-expr)
or
expr?join(separator-expr, last-separator-expr, when-empty-expr)
or
expr?join(separator-expr, last-separator-expr, when-empty-expr,
after-last-expr)
```

Joins a sequence into a string by using a string separator. Optionally, you can include an additional separator between the next-to-last and last elements, plus an additional argument to use when the sequence has no elements, as well as a suffix to add after the last element has been appended.

Example 1: Join a list with a separator:

```
${["a", "b", 1]?join(",")}
```

produces this output:

```
a,b,1
```

Note that the number one was forcibly transformed to a string.

Example 2: Join a list with two separators:

```
<#assign list=["red", "blue", "cyan"]>
The colors are ${list?join(", ", " and ")}
```

produces this output:

```
The colors are red, blue and cyan
```

Example 3: Join a list with separators and an optional empty string:

```
<#assign list=["red", "blue", "cyan"]>
The colors are ${list?join(", ", ", ", "empty")}
<#assign list=[]>
The colors are ${list?join(", ", ", ", "empty")}
```

produces this output:

```
The colors are red, blue, cyan
The colors are empty
```

Example 4: Join a list with an optional empty string and an optional suffix

```
<#assign list=["red", "blue", "cyan"]>
The colors are ${list?join(", ", ", ", "empty", " - " +
list?size?string)}
<#assign list=[]>
The colors are ${list?join(", ", ", ", "empty", " - " +
list?size?string)}
```

produces this output:

```
The colors are red, blue, cyan - 3
The colors are empty
```

The optional suffix is appended only if the item had any elements, as in the first assignment. The suffix is composed of a separator (" - ") plus the list size, in string format. Note that the size is converted to a string to enable the concatenation. Otherwise, RPL would have attempted to add a string with a number, which would result in an error.

last

```
expr?last
```

Returns the last sub-variable of the sequence.

If the sequence is empty, an error occurs that terminates template processing.

reverse

```
expr?reverse
```

Returns the sequence in reversed order.

seq_contains

```
expr?seq_contains(expr)
```

Determines whether the sequence contains the specified value.

The built- in has one parameter, the value to find.

NOTE: The `seq_` prefix is required in the built-in name to differentiate it from the `contains` built-in that searches a substring in a string (since a variable can be both a string and a sequence on the same time).

Example:

```
<#assign x = ["red", 16, "blue", "cyan"]>
"blue": ${x?seq_contains("blue")?string("yes", "no")}
"yellow": ${x?seq_contains("yellow")?string("yes", "no")}
16: ${x?seq_contains(16)?string("yes", "no")}
"16": ${x?seq_contains("16")?string("yes", "no")}
```

produces this output:

```
"blue": yes
"yellow": no
16: yes
"16": no
```

To find the value, the built-in uses RPL's comparison rules (as if you were using the `==` operator). The difference is that comparing two values of different types or of types for which RPL does not support will not cause an error; instead, the values will be evaluated as not equal. Thus, you can use this built-in to find only scalar values (i.e. string, number, boolean or date/time values). For other types, the result will always be `false`.

For fault tolerance, this built-in also works with collections.

seq_index_of

```
exp?seq_index_of(expr)
exp?seq_index_of(expr, index-expr)
```

Returns the index of the first occurrence of a value in the sequence, or -1 if the sequence does not contain the specified value. The value to find is specified as the first parameter.

Optionally, you can specify the index where the searching should begin as a second parameter. This might be useful if the same item can occur multiple times in the same sequence. There is no restriction on the numerical value of the second parameter: if the value is negative, it evaluates to zero; if it is greater than the length of the sequence, it evaluates to the length of the sequence. Decimal values will be truncated to integers.

To find the value, the built-in uses RPL's comparison rules (as if you were using == operator). The difference is that using this built-in, comparing two values of different or unsupported types will not cause an error; instead, the values will be evaluated as not equal. Thus, you can use it only to find scalar values (i.e. string, number, boolean or date/time values). For other types of values, the result will always be -1.

NOTE: `seq_` prefix is required in the built-in name to differentiate it from the `index_of` built-in that searches a substring in a string, since a variable can be both a string and a sequence at the same time.

Example:

```
<#assign colors = ["red", "green", "blue"]>
${colors?seq_index_of("blue")}
${colors?seq_index_of("red")}
${colors?seq_index_of("purple")}
```

produces this output:

```
2
0
-1
```

Example:

```
<#assign names = ["Joe", "Fred", "Joe", "Susan"]>
No 2nd param: ${names?seq_index_of("Joe")}
-2: ${names?seq_index_of("Joe", -2)}
-1: ${names?seq_index_of("Joe", -1)}
 0: ${names?seq_index_of("Joe", 0)}
 1: ${names?seq_index_of("Joe", 1)}
 2: ${names?seq_index_of("Joe", 2)}
 3: ${names?seq_index_of("Joe", 3)}
 4: ${names?seq_index_of("Joe", 4)}
```

produces this output:

```
No 2nd param: 0
-2: 0
-1: 0
 0: 0
 1: 2
 2: 2
 3: -1
 4: -1
```

seq_last_index_of

```
expr?seq_last_index_of(expr)
expr?seq_last_index_of(expr, index-expr)
```

Returns the index of the last occurrence of a value in the sequence, or -1 if the sequence does not contain the specified value. This built-in is the same as `seq_index_of`, except that it searches backward, starting from the last item of the sequence.

Optionally, you can specify the index where the searching should begin as a second parameter. This might be useful if the same item can occur multiple times in the same sequence. There is no restriction on the numerical value of the second parameter: if the value is negative, it evaluates to zero; if it is greater than the length of the sequence, it evaluates to the length of the sequence. Decimal values will be truncated to integers.

NOTE: The `seq_ prefix` is required in the built-in name to differentiate it from the `last_index_of` built-in that searches for a substring in a string (since a variable can be both string and sequence at the same time).

For example:

```
<#assign names = ["Joe", "Fred", "Joe", "Susan"]>
No 2nd param: ${names?seq_last_index_of("Joe")}
-2: ${names?seq_last_index_of("Joe", -2)}
-1: ${names?seq_last_index_of("Joe", -1)}
 0: ${names?seq_last_index_of("Joe", 0)}
 1: ${names?seq_last_index_of("Joe", 1)}
 2: ${names?seq_last_index_of("Joe", 2)}
 3: ${names?seq_last_index_of("Joe", 3)}
 4: ${names?seq_last_index_of("Joe", 4)}
```

produces this output:

```
No 2nd param: 2
-2: -1
-1: -1
 0: 0
 1: 0
 2: 2
 3: 2
 4: 2
```

size

```
expr?size
```

Returns the number of sub-variables in sequence as a numerical value. The highest possible index in a sequence is `s?size - 1` (since the index of the first sub-variable is 0) assuming that the sequence has at least one sub-variable.

sort

```
expr?sort
```

Returns the sequence sorted in ascending order.

To sort in descending order, use this built-in, then the `reverse` built in.

This built-in works only if all sub-variables are of the same type. If the sub-variables are strings, it uses locale (language) specific lexical sorting (which is usually not case sensitive).

Example:

```
<#assign ls = ["whale", "Barbara", "zeppelin", "aardvark",
"beetroot"]?sort>
<#list ls as i>${i} </#list>
```

produces this output with the US locale:

```
aardvark Barbara beetroot whale zeppelin
```

sort_by

```
expr?sort_by(key-expr)
```

Returns the sequence of hashes sorted by the given hash sub-variable in ascending order.

To sort in descending order, use this built-in, then the `reverse` built in.

The rules are the same as with the `sort` built-in, except that the sub-variables of the sequence must be hashes, and you must give the name of a hash sub-variable that will determine the order.

For example:

```
<#assign ls = [
  {"name":"whale", "weight":2000},
  {"name":"Barbara", "weight":53},
  {"name":"zeppelin", "weight":-200},
  {"name":"aardvark", "weight":30},
  {"name":"beetroot", "weight":0.3}
]>
Order by name:
<#list ls?sort_by("name") as i>
- ${i.name}: ${i.weight}
</#list>

Order by weight:
<#list ls?sort_by("weight") as i>
- ${i.name}: ${i.weight}
</#list>
```

produces this output with the US locale:

```
Order by name:
- aardvark: 30
- Barbara: 53
- beetroot: 0.3
- whale: 2000
- zeppelin: -200

Order by weight:
- zeppelin: -200
- beetroot: 0.3
- aardvark: 30
- Barbara: 53
- whale: 2000
```

If the sub-variable you want to use for the sorting is nested at a deeper level (that is, if it is a sub-variable of a sub-variable), you can use a sequence as a parameter that specifies the names of the sub-variables that lead down to the desired sub-variable. For example:

```
<#assign members = [
    {"name": {"first": "Joe", "last": "Smith"}, "age": 40},
    {"name": {"first": "Fred", "last": "Crooger"}, "age": 35},
    {"name": {"first": "Amanda", "last": "Fox"}, "age": 25}]>
Sorted by name.last:
<#list members?sort_by(['name', 'last']) as m>
- ${m.name.last}, ${m.name.first}: ${m.age} years old
</#list>
```

produces this output with the US locale:

```
Sorted by name.last:
- Crooger, Fred: 35 years old
- Fox, Amanda: 25 years old
- Smith, Joe: 40 years old
```

unique

Returns a new sequence where the elements are unique.

Built-ins for hashes

keys

```
expr?keys
```

Returns a sequence that contains all the lookup keys in the hash. Note that not all hashes support this.

Example:

```
<#assign h = {"name":"mouse", "price":50}>
<#assign keys = h?keys>
<#list keys as key>${key} = ${h[key]}; </#list>
```

produces this output:

```
name = mouse; price = 50;
```

Because hashes do not define an order for their sub-variables in general, the order in which key names are returned can be arbitrary. However, some hashes maintain a meaningful order. For example, hashes created with the above {...} syntax preserve the same order as specified by the sub-variables.

values

```
expr?values
```

Returns a sequence that contains all variables in the hash.

Because hashes do not define an order for their sub-variables in general, the order in which key names are returned can be arbitrary. However, some hashes maintain a meaningful order. For example, hashes created with the above *{...}* syntax preserve the same order as specified by the sub-variables.

Built-ins for XML nodes

Note that the variables returned by these built-ins are generated by the node variable implementation with which the built-ins are used. This means that the returned variables might have extra features in addition to what it stated here. For example, with the XML DOM nodes, the sequence retuned by the `children` built-in also can be used as hash and maybe as string, as described in the section about XML processing.

ancestors

```
expr?ancestors
```

Returns a sequence that contains all the node's ancestors, starting with the immediate parent and ending with the root node. The result of this built-in is also a method, by which you can filter the result with the fully-qualified name of the node. For example, use `node?ancestors("section")` to get the sequence of all ancestors with name *`section`*.

children

```
expr?children
```

Returns a sequence that contains all the node's child nodes (i.e. immediate descendant nodes).

XML: This is almost the same as special hash key *, except that it returns all nodes, not only elements. This means that the possible children are element nodes, text nodes, comment nodes, processing instruction nodes, etc. but not attribute nodes. Attribute nodes are excluded from the sequence.

parent

```
expr?parent
```

Returns the node's immediate parent node. The root node has no parent node, so for the root node, the expression `node?parent??` evaluates to false.

XML: Note that the value returned by this built-in is also a sequence (same as the result of XPath expression .., when you write someNode[".."]). Also note that for attribute nodes, the built-in returns the element to which the attribute belongs, despite the fact that attribute nodes are not counted as children of the element.

root

```
expr?root
```

Returns the root node of the tree to which this node belongs.

XML: According to W3C, the root of an XML document is not the topmost element node. Rather, the document, which is the parent of the topmost element, is the root node. For example, if you want to get the topmost element of the XML (the "document element" not "document') called *foo*, you have to write `someNode?root.foo`. If you write `someNode?root`, you get the document itself, not the document element.

node_name

```
expr?node_name
```

Returns the string used to determine which user-defined directive to invoke to handle this node when it is visited. For more information, see the `visit` and `recurse` directives.

XML: If the node is an element or attribute, the string will be the local (prefix free) name of the element or attribute. Otherwise, the name usually starts with @ followed by the node type. Note that the node name is not the same as the node name returned in the DOM API; the goal of RPL node names is to give the name of the used-defined directive that will process the node.

node_namespace

```
expr?node_namespace
```

Returns the namespace string of the node. RPL does not define the exact meaning of the node namespace; it depends on what your node variables are modeling. It is possible that a node does not have a node namespace defined. In this case, the built-in should evaluate to an undefined variable (i.e. `node?node_namespace??` is false), so you cannot use the returned value.

XML: In the case of XML, the built-in returns the XML namespace URI (such as "http://www.w3.org/1999/xhtml"). If an element or attribute node does not use an XML namespace, this built-in evaluates to an empty string. For other XML nodes, the built-in always returns an undefined variable.

node_type

```
expr?node_type
```

Returns a string that describes the node type. RPL does not define the meaning of a node type; it depends on what your variables are modeling. It is possible that a node does not support a node type. In this case, the built-in evaluates to an undefined value, so you cannot use the returned value. However, you can check whether a node supports the type property with `node?node_type??`.

XML: The possible values are: "attribute", "text", "comment", "document_fragment", "document", "document_type", "element", "entity", "entity_reference", "notation", "pi". Note that a there is no "cdata" type, because CDATA is considered a plain text node.

Additional built-ins

is_...

```
expr?is_string
expr?is_number
expr?is_boolean
expr_is_datee
expr?is_hash
expr?is_hash_ex
expr?is_sequence
expr?is_directive
expr?is_node
```

These built-ins check the type of a variable and return true or false, depending on the type.

The following table lists the `is_...` built-ins.

This built-in	Returns true if the value is a ...
is_string	string
is_number	number
is_boolean	boolean
is_date	date (all types: date-only, time-only and date-time)
is_hash	hash
is_hash_ex	extended hash (i.e. supports ?keys and ?values)
is_sequence	sequence
is_directive	any kind of directive
is_node	node

number_to_date, number_to_time, number_to_datetime

```
expr?number_to_date
expr?number_to_time
expr?number_to_datetime
```

Use these built-ins to convert a number (usually a Java long) to a date, time or date-time, respectively. These built-ins work in the same way as java.util.Date(long) in Java. That is, the number is interpreted as the milliseconds passed since the epoch. The number can be anything and of any type, as long as its value fits into a long. If the number is not a whole number, it will be rounded to a whole number using the half-up rule. This conversion is not automatic.

Example:

```
${1305575275540?number_to_datetime}
${1305575275540?number_to_date}
${1305575275540?number_to_time}
```

produces output similar to this, depending on the current locale and time zone:

```
May 16, 2011 3:47:55 PM
May 16, 2011
3:47:55 PM
```

About regular expressions

The following table provides a summary of regular expression constructs.

Construct	Matches
Characters	
x	The character x
\\	The backslash character
\0*n*	The character with octal value 0*n* (0 <= *n* <= 7)
\0*nn*	The character with octal value 0*nn* (0 <= *n* <= 7)
\0*mnn*	The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
\x*hh*	The character with hexadecimal value 0x*hh*
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')

\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\cx	The control character corresponding to x
Character classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character classes	
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
POSIX character classes (US-ASCII only)	
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]

\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}\x20]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]
Classes for Unicode blocks and categories	
\p{InGreek}	A character in the Greek block (simple block)
\p{Lu}	An uppercase letter (simple category)
\p{Sc}	A currency symbol
\P{InGreek}	Any character except one in the Greek block (negation)
[\p{L}&&[^\p{Lu}]]	Any letter except an uppercase letter (subtraction)
Boundary matchers	
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input
Greedy quantifiers	
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{*n*}	X, exactly *n* times

X{*n*,}	X, at least *n* times
X{*n*,*m*}	X, at least *n* but not more than *m* times
Reluctant quantifiers	
X??	X, once or not at all
X*?	X, zero or more times
X+?	X, one or more times
X{*n*}?	X, exactly *n* times
X{*n*,}?	X, at least *n* times
X{*n*,*m*}?	X, at least *n* but not more than *m* times
Possessive quantifiers	
X?+	X, once or not at all
X*+	X, zero or more times
X++	X, one or more times
X{*n*}+	X, exactly *n* times
X{*n*,}+	X, at least *n* times
X{*n*,*m*}+	X, at least *n* but not more than *m* times
Logical operators	
XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group
Back references	
\n	Whatever the *nth* capturing group matched
Quotation	
\\	Nothing, but quotes the following character
\\Q	Nothing, but quotes all characters until \\E
\\E	Nothing, but ends quoting started by \\Q

Special constructs (non-capturing)	
(?:X)	X, as a non-capturing group
(?idmsux-idmsux)	Nothing, but turns match flags i d m s u x on - off
(?idmsux-idmsux:X)	X, as a non-capturing group with the given flags i d m s u x on - off
(?=X)	X, via zero-width positive lookahead
(?!X)	X, via zero-width negative lookahead
(?<=X)	X, via zero-width positive lookbehind
(?<!X)	X, via zero-width negative lookbehind
(?>X)	X, as an independent, non-capturing group

About the date format

Date and time formats are specified by the date and time pattern strings. Within the date and time pattern strings, unquoted letters from 'A' to 'Z' and from 'a' to 'z' are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (') to avoid interpretation. "''" represents a single quote. All other characters are not interpreted, they are simply copied into the output string during formatting or matched against the input string during parsing.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2

Letter	Date or Time Component	Presentation	Examples
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

Text
For formatting, if the number of pattern letters is 4 or more, the full form is used. Otherwise, a short or abbreviated form is used, if available. For parsing, both forms are accepted, regardless of the number of pattern letters.

Number
For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it is needed to separate two adjacent fields.

Year

If the formatter uses the Gregorian calendar, the following rules apply:

- For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits. Otherwise, it is interpreted as a number.
- For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. This means that using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D.
- For parsing with the abbreviated year pattern ("y" or "yy"), SimpleDateFormat must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the SimpleDateFormat instance is created. For example, using a pattern of "MM/dd/yy" and a SimpleDateFormat instance created on Jan 1, 1997, the string "01/11/12" is interpreted as Jan 11, 2012 while the string "05/04/64" is interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by Character.isDigit(char), are parsed into the default century. Any other numeric string is interpreted literally. This means that "01/02/3" or "01/02/003" are parsed, using the same pattern, as Jan 2, 3 AD. Likewise, "01/02/-3" is parsed as Jan 2, 4 BC.

Otherwise, calendar-system specific forms are applied. For both formatting and parsing, if the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar specific short or abbreviated form is used.

Month

If the number of pattern letters is 3 or more, the month is interpreted as text. Otherwise, it is interpreted as a number.

General time zone

Time zones are interpreted as text if they have names. For time zones representing a GMT offset value, the following syntax is used:

GMTOffsetTimeZone:

GMT *Sign Hours* : *Minutes*

Sign: one of

+ -

Hours:

Digit

Digit Digit

Minutes:

Digit Digit

Digit: one of

0 1 2 3 4 5 6 7 8 9

Hours must be between 0 and 23, and minutes must be between 00 and 59. The format is locale-independent and digits must be taken from the Basic Latin block of the Unicode standard.

For parsing, RFC 822 time zones are also accepted.

RFC 822 time zone

For formatting, the RFC 822 4-digit time zone format is used:

RFC822TimeZone:

Sign TwoDigitHours Minutes

TwoDigitHours:

Digit Digit

TwoDigitHours must be between 00 and 23. Other definitions are the same as for general time zones.

Examples:

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific Time time zone.

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

Chapter 6. Directive Reference

assign

```
<#assign name=value>
or
<#assign name1=value1 name2=value2 ... nameN=valueN>
or
<#assign same as above... in namespacehash>
or
<#assign name>
  capture this
</#assign>
or
<#assign name in namespacehash>
  capture this
</#assign>
```

Parameter	Description
name	The name of the variable. This is not an expression. However, it can be written as a string literal. This can be useful if the variable name contains reserved characters, for example `<#assign "foo-bar" = 1>`. Note that this string literal does not expand interpolations (as "${foo}").
value	The value to store. This is an expression.
namespacehash	A hash that was created for a namespace (by import). This is an expression.

Use this directive to create a new variable or replace an existing variable. Note that you can create and replace only top-level variables (i.e. you cannot create/replace some_hash.subvar, only some_hash).

Example: Create a variable called `seasons` that stores a sequence:

```
<#assign seasons = ["winter", "spring", "summer", "autumn"]>
```

Example: Increment the numerical value stored in variable called `test`:

```
<#assign test = test + 1>
```

As a convenience feature, you can do more assignments with one assign tag. The following example does the same as the two previous examples:

```
<#assign
  seasons = ["winter", "spring", "summer", "autumn"]
  test = test + 1
>
```

You can use this directive to create variables in namespaces. Normally, the directive creates the variable in the current namespace. However, if you use `namespacehash`, you can create/replace a variable in another namespace. For example, the following code creates/replaces a variable called `bgColor` in the namespace used for */contentlibrary/mylib.htm*:

```
<#import "/contentlibrary/mylib.htm" as my>
<#assign bgColor="red" in my>
```

You can also use this directive to capture the output generated between its start and end tags. That is, everything printed between the tags will not be shown on the page, but will be stored in the variable. For example:

```
<#macro myMacro>foo</#macro>
<#assign x>
  <#list 1..3 as n>
    ${n} <@myMacro />
  </#list>
</#assign>
Number of words: ${x?word_list?size}
${x}
```

produces this output:

```
Number of words: 6
    1 foo
    2 foo
    3 foo
```

IMPORTANT: Do not to use the following to insert variables into strings:

```
<#assign x>Hello ${user}!</#assign> <#-- BAD PRACTICE! -->
```

Instead, use:

```
<#assign x="Hello ${user}!">
```

attempt, recover

```
<#attempt>
  attempt block
<#recover>
  recover block
</#attempt>
```

Parameter	Description
attempt block	Template block with any content. This will be always executed, but if an error occurs during execution, all output from this block is rolled back, and the recover block will be executed.
recover block	This parameter is required. Template block with any content. This will be executed only if an error occurs during the execution of the attempt block. You can use the recover block, for example, to print an error message. `attempt` and `recover` can be nested into other `attempt` blocks or `recover` blocks.

Use these directives to successfully output the page when execution of a part of the page fails. If an error occurs during the execution of the `attempt` block, all output from this block is rolled back, and the `recover` block is executed. If no error occurs during the execution of the attempt block, the recover block is ignored.

For example:

```
Primary content
<#attempt>
  Optional content: ${missinghash.thisMayFails}
<#recover>
  Ops! The optional content is not available.
</#attempt>
Primary content continued
```

if `thisMayFails` variable does not exist, the output is:

```
Primary content
  Ops! The optional content is not available.
Primary content continued
```

if `thisMayFails` variable exists and its value is *123*, the output is:

```
Primary content
  Optional content: 123
Primary content continued
```

The attempt block is either printed in its entirety (if there is no error) or not at all (if an error occurs). In other words, if an error occurs anywhere within the block, error-free parts will not be printed. This is implemented with the aggressive buffering of the output inside the attempt block. Not even the flush directive will send the output to the client.

To prevent misinterpretation, do not use attempt/recover for handling undefined variables (instead, use missing value handler operators). It can handle all types of errors that occur when the block is executed except syntactical errors, which are detected earlier. These directives are meant to enclose bigger template fragments, where errors can occur at various points. For example, a section of your template prints ads, but that is not the primary content of the page. In this case, you can put the ad printing section in the `attempt` block so that the rest of the page will print if an error occurs when printing ads.

A error message is available inside the `recover` block with the .error special variable (references to special variable begin with a dot (for example: `${.error}`).

Errors occurring during template execution are always logged, even if they occur inside the `attempt` block.

compress

```
<#compress>
  ...
</#compress>
```

This directive is useful for removing superfluous white space when you use a white space-insensitive format (e.g. HTML or XML).

The directive captures the output generated between its start and end tags, and reduces all unbroken white space sequences to a single white space character. If the replaced sequence contains line breaks, the inserted character is a line break. Otherwise, the inserted character a space. The very first and very last unbroken white space sequences are removed.

Example:

```
<#assign x = "    moo  \n\n   ">
(<#compress>
  1 2  3   4    5
  ${moo}
  test only

  I said, test only

</#compress>)
```

produces this output:

```
 (1 2 3 4 5
moo
test only
I said, test only)
```

content, filter, break

```
<#content datasource as hash-name option=value-expr ...>
  <#filter key1 operator value-expr1 [&& key2 operator value-expr2]
...>
  ...
</#content>
```

Content	
Parameter	**Description**
datasource	The name of the datasource alias, as defined in the Campaign Workbook. Not an expression. Note that this is an alial of the datasource, not its name.
hash-name	The name of the structure that can be used in the enumeration of records. The hash name will be used to obtain the values for each of the resulting values. It will also be used as a prefix to the special variables that start with the hash name and have a suffix of `_index`, and `_has_next`.
option	It can be: limit=numeric-expression Limits the number of records returned from the execution of the expression. If the limit is greater than the existing number of records, it is ignored. The numeric expression is any expression that returns a number, including any numeric constants. Non-integer numbers get rounded off. IMPORTANT: An account has a maximum allowable number of rows (default is 20); providing a limit bigger than that number produces an error.

Filter	
Parameter	**Description**
key1, key2, …	The alias names of the columns to be used to perform the filter, as defined in the Datasources tab of the Campaign Workbook. Not an expression.
operator	Either one of the following: **=, ==** Equals operation **gte** greater than or equals to **gt** greater than **lte, <=** less than or equals to **lt, <** less than
value-expr-1, value-expr-2, …	The values to use to search for data. Can be expressions according to the data type in the datasource. WARNING: Template execution will fail if the wrong types are provided. Ask the campaign designer for the proper types.
&&, &	Represents the “and” operator. Must be used between (key, operator, value) sets.

This directive searches content in an optimized fashion, applying the supplied predicates (expression set). The directive creates a hash, which is used in the body of the directive to retrieve and output the content by field name. The hash name is given in the `content` directive, while the predicates are given in the `filter` directive.

Unlike the `data` directive, this directive is optimized to retrieve content only as needed. For instance, if an audience of a million records is launched and there are only a thousand variants of the content, Oracle Responsys internally caches and keeps the data for future requests in an efficient manner. It also deals with images in the content by changing paths so that the files can be accessible from the Internet.

Two special loop variables are available inside the data loop:

*hash-name*_**index**
This is a numerical value that contains the index of the current item being stepped over in the loop.

*hash-name*_**has_next**
Boolean value that tells if the current item the last in the datasource query or not.

You can use the `break` instruction inside the loop body to exit the iteration. For more information, see the `list` directive.

IMPORTANT: The results are not sorted in any order. If ordering is required, create an in-memory list of hashes in the loop using RPL, and use the `?sort_by` builtin.

Example:

```
<#assign today=.now>
<#assign oneweek=dayadd(today, 7)>
<#content Events as event limit=5>
  <#filter REGION_NAME=PROFILE.REGION && DATE gte today && DATE lt
oneweek>
  ${event.DESCR}-${event.CATEGORY}-${event.VENUE}-
${event.DATE?string("yyyyMMdd")}
</#content>
```

produces output similar to this:

```
Eagles in concert-CONCERT-HP Pavillion-20161028
Knicks vs. Golden State-NBA-Oracle Arena-20161024
Lakers vs. Golden State-NBA-Oracle Arena-20161028
Trailblazers vs. Kings-NBA-Sleep Train Arena-20161030
```

Limiting the number of records

To limit the number of records, use the `limit=x` option in the `content` declaration. The limit expression can be any numeric expression. See the example in this section.

Two limits are configured in your account:

Default Limit

When the limit is not specified in the `content` directive, the default is 20. This default limit can be configured by the account administrator.

Maximum Limit

To ensure efficiency, when the limit is specified, it cannot exceed this number of results. The default limite is also. Entering a limit greater than the allowed maximum produces an error.

Conditionally breaking out of the looping

It is possible to loop through the found records until all records are reached, according to the `filter` and `limit` declarations. However, in some cases you might want to stop the iteration by using the `break` directive. The `break` directive is usually placed within an `if` construct.

Obtaining context information within the loop

In addition to the `namespace` variable, two other variables are also declared, to give context about the execution of the loop: `hash-name_has_next`, and `hash-name_index`.

In the above example, these new variables are *`event_has_next`* and *`event_index`*.

Handling images

If the content table contains strings that begin with the sequence `/contentlibrary/`, they are automatically converted to a valid path for the internet through the use of the Content Delivery Network (CDN). This way, you can create content that is related to images in the Content Library. For example, the image is created at the path

`campaigns/campaign10242016/welcome1.jpg`.

To properly convert the image to a CDN address during a launch, you must store the file name in the content table as

`/contentlibrary/campaigns/campaign10242016/welcome1.jpg`.

With this mechanism. you can manage images stored in the Content Library using meta data and paths in supplemental tables. Upload your images to the Content Library, and fill the table with paths related to each specific piece of content.

You can also store and manage content other than images (e.g. PDFs) in this way.

data, filter, fields, break

```
<#data datasource as hash-name option=value-expr ...>
  <#filter key1=value-expr-1 key2=value-expr-2 ...>
  <#fields field-name1 field-name2 ...>
  ...
</#data>
```

Parameter	Description
datasource	The name of the datasource alias, as defined in the Message Designer for Email. Not an expression. Please note that this is not the name of the actual datasource. Datasource aliases are defined in the Message Designer for Email's datasources tab.
hash-name	The name of the structure that can be used in the enumeration of records.
option	One of the following values:

Parameter	Description
	limit A numeric expression Limits the number of records returned from the execution of the expression. If the limit is greater than the existing number of records, it is ignored. This option is invalid when using the singlekey option. The numeric expression is any expression that returns a number, including any numeric constants. Non-integer numbers are rounded off. **singlekey** A boolean-expression Specifies that only one row should be returned for each key that matches the filter. This option is restricted to the case when there is only one key. This option does not allow limiting the number of records returned (see the limit option above); it is an error to specify both options at the same time. The boolean expression is any expression that results on true or false, including the constants `true` and `false`.

Filter	
Parameter	**Description**
key1, key2, ...	The alias names of the columns to use for the filter, as defined in the Datasources tab of Message Designer for Email. Not an expression.
value-expr-1, value-expr-2, ...	The values to use to look up the data. The values can be expressions according to the data type in the datasource. If the data types are different from the ones in the data source, template execution will fail. If you do not know the value types in the data source, ask the campaign designer for the proper types to use.
Fields	
Parameter	**Description**

field-name1, field-name2, etc.	The alias names of the fields to use. These appear in the hash specified by `hash-name` as hash values. The alias names are specified in the Datasources section of the Message Designer for Email.

This directive selects records from a given datasource with the given filter using the specified fields. A new hash is created with the given fields. The records are filtered as specified in the `<#filter …>` construct. The hash is created with the `<#fields…>` construct.

There are two special loop variables available inside the data loop:

hash-name_index
This is a numerical value that contains the index of the current item being stepped over in the loop.

hash-name_has_next
A boolean value that tells whether or not the current item the last in the datasource query.

The `break` instruction can be used inside the loop body to exit the iteration. For more information, see the `list` directive.

Select based on one value for one or multiple lookup fields
To use this option, specify multiple field/-value pairs in the filter specification. Each value must be a scalar.

The data query performed has the AND semantics.

Example:

```
<#data events as event limit=10>
  <#filter category="NBA Tickets" region="NY">
  <#fields event_id description venue city state event_date>
Event:
- ${event.event_id}, ${event.description}, ${event.venue},
  ${event.city}, ${event.state} ${event.event_date}
</#data>
```

produces output similar to this:

```
Event:
- 15, Knicks vs. Raptors, Madison Square Garden,
  New York, NY, 4/30/2013
Event:
- 16, Knicks vs. Clippers, Madison Square Garden,
  New York, NY, 5/1/2013
```

This option will look for records whose category is *NBA Tickets* and the region is *NY*.

In this modality, you can also specify the limit option as shown. If there were more records in the database table then the specified limit, the response would be limited to the first 10 elements.

In addition to retrieving records with one value (AND semantics), it is possible to specify multiple values for those fields.

This introduces a lower-level OR semantics to the selection of fields, for example:

```
<#data events as event limit=10>
  <#filter category=["NBA Tickets", "MLB Tickets"] region="NY">
  <#fields event_id description venue city state event_date>
Event:
- ${event.event_id}, ${event.description}, ${event.venue},
  ${event.city}, ${event.state} ${event.event_date}
</#data>
```

Notice that the category now has a sequence. The meaning of this expression is "obtain the records whose category is either *NBA Ticket*s or *MLB Tickets*, but whose region has the value of *NY*." This would get tickets for the NBA or the MLB in the NY region.

The `limit` option is also valid in this context. To use this option, specify a *s*ingle field in the `filter` specification and provide a sequence of scalars (or a single value). All scalars must be of the same type and in accordance with the datasource specification.

You can add the `singlekey` option with a value of *`true`* to the data directive. The `singlekey` option only works with a single field.

The data query performed has the OR semantics. The sequence provides the possible values that the field can take. When a single value is provided, it has the EQUALITY semantics.

IMPORTANT: The filter will return at most only one value for each key.

Example:

```
<#data events as event singlekey=true>
  <#filter category=["NBA Tickets", “MLB Tickets”, “NFL Tickets”]>
  <#fields category event_id description venue city state
event_date>
Event:
- ${event.category}, ${event.event_id}, ${event.description},
${event.venue}
  ${event.city}, ${event.state}, ${event.event_date}
</#data>
```

produces output similar to this:

```
Event:
- MLB Tickets, 1, Mets vs Cardinals, Shea Stadium,
  New York, NY, 5/25/2013
Event:
- NBA Tickets, 15, Knicks vs. Raptors, Madison Square Garden,
  New York, NY, 4/30/2013
Event:
- NFL Tickets, 12, Jets vs. Raiders, MetLife Stadium,
  New York, NY, 3/1/2013
```

If two records match, for example *NBA Tickets,* then only the first one is returned. This behavior is specified with the addition of the `singlekey` option.

IMPORTANT: It is not currently possible to query multiple keys with multiple values with th `singlekey` semantics.

Using the `limit` option will cause an error. Also, using a single value as opposed to a sequence will return zero, or at most, one record.

This option is useful when you want to retrieve, for example, all the event types in the NY region as shown in the following example:

```
<#assign categories=[]>
<#data events as event singlekey=true>
  <#filter category=["NBA Tickets", "MLB Tickets", "NFL Tickets"]>
  <#fields category>
  <a href="#${event.category}">${event.category}</a>
  <#assign categories=categories + [ event.category ]>
</#data>

<#list categories as category_name>
  <a name="${category_name}" />
  <h3>${category_name}</h3>
  <#data events as event limit=5>
    <#filter category=${category_name}>
    <#fields description venue city state event_date>
  ${event.description} at ${event.venue} in
  ${event.city}, ${event.state} on ${event.event_date}
  </#data>
</#list>
```

Limiting the number of records

To get a maximum number of records, use the `limit=x` option in the data declaration. This causes the number of records obtained be cut if there are more records than the specified limit. The `limit` can be any numeric expression.

This option is not valid with the `singlekey` option.

Conditionally breaking out of the loop

It is possible to loop through the found records until all records are reached, according to the `filter` and `limit` declarations. However, you can stop the iteration by using the `break` directive, usually placed within an `if` construct.

Obtaining Context information within the loop.

In addition to the namespace variable, two other variables, `hash_has_next` and `hash_index` are also declared, to give context about the execution of the loop. In our example, these new variables are `offer_has_next` and `offer_index`.

escape

```
<#escape identifier as expression>
  ...
  <#noescape>...</#noescape>
  ...
</#escape>
```

When you surround a part of the template with an `escape` directive, interpolations (${...}) that occur inside the block are automatically combined with the escaping expression. This is a convenience method for avoiding writing similar expressions several times. It does not affect interpolations in string literals (as in <#assign x = "Hello ${user}!">) or numerical interpolations (#{...}).

Example:

```
<#escape x as x?html>
  First name: ${firstName}
  Last name: ${lastName}
  Maiden name: ${maidenName}
</#escape>
```

is equivalent to:

```
  First name: ${firstName?html}
  Last name: ${lastName?html}
  Maiden name: ${maidenName?html}
```

Note that it is irrelevant what identifier you use in the directive: it serves as a formal parameter to the escaping expression.

When calling macros or the `include` directive, it is important to understand that `escape` affects only interpolations that occur between the *`<#escape ...>`* and *`</#escape>`* tags in the template text. That is, it will not escape anything before *`<#escape ...>`* or after *`</#escape>`* in the text, not even if it is called from inside the escaped section. For example:

```
<#assign x = "<test>">
<#macro m1>
  m1: ${x}
</#macro>
<#escape x as x?html>
  <#macro m2>m2: ${x}</#macro>
  ${x}
  <@m1/>
</#escape>
${x}
<@m2/>
```

produces this output:

```
  &lt;test&gt;
  m1: <test>
<test>
m2: &lt;test&gt;
```

The effect of the escape directive are applied at template parsing time rather than at template processing time. This means that if you call a macro or include another template from within an escape block, it will not affect the interpolations in the macro/included template, since macro calls and template includes are evaluated at template processing time. On the other hand, if you surround one or more macro declarations (which are evaluated at template parsing time) with an escape block, the interpolations in those macros will be combined with the escaping expression.

If you need to temporarily turn off escaping for one or two interpolations in an escape block, you can close and later reopen the escape block, but then you have to write the escaping expression twice. Instead, you can instead use the noescape directive as shown in the following example:

```
<#escape x as x?html>
  User: ${user.Name}
  Text: ${user.Text}
  <#noescape>Title: ${user.title}</#noescape>
  ...
</#escape>
```

the above example is equivalent to:

```
  User: ${user.Name?html}
  Text: ${user.Text?html}
  Title: ${user.title}
```

You can nest escapes, although you should do it only in rare circumstances. Therefore, you can write something like the following example:

```
<#escape x as x?html>
  Customer Name: ${customerName}
  Items to ship:
  <#escape x as itemCodeToNameMap[x]>
    ${itemCode1}
    ${itemCode2}
    ${itemCode3}
    ${itemCode4}
  </#escape>
</#escape>
```

which is equivalent to:

```
  Customer Name: ${customerName?html}
  Items to ship:
    ${itemCodeToNameMap[itemCode1]?html}
    ${itemCodeToNameMap[itemCode2]?html}
    ${itemCodeToNameMap[itemCode3]?html}
    ${itemCodeToNameMap[itemCode4]?html}
```

When using the `noescape` directive in a nested escape block, it undoes only a single level of escaping. Therefore, to completely turn off escaping in a two-level deep escaped block, use two nested `noescape` directives.

fail

```
<#fail>
or
<#fail reason>
```

Parameter	Description
reason	Informative message about the reason for termination. Expression evaluates to a string.

Stops the processing of the entire launch. After this directive executes, no other instructions are executed.

A failed run appears in the Live Report as a launch failure.

function, return

```
<#function name param1 param2 ... paramN>
  ...
  <#return returnValue>
  ...
</#function>
```

Parameter	Description
name	The name of method variable. Not an expression.
param1, param2, ...etc.	The name of the local variables store the parameter values (not an expression), optionally followed by = and the default value (an expression).
paramN	The last parameter, can optionally include a trailing ellipsis (...), which indicate that the macro takes a variable number of parameters. Local variable `paramN` will be a sequence of the extra parameters.
returnValue	The expression that calculates the value of the method call.

Creates a method variable in the current namespace. This directive works in the same way as the `macro` directive, except that the `return` directive must have a parameter that specifies the return value of the method, and that attempts to write to the output will be ignored. If the `</#function>` is reached (i.e. there was no return `returnValue`), the return value of the method is an undefined variable.

You can use the `return` directive anywhere and any number of times between the <#function ...> and </#function>.

Parameters without default values must precede parameters with default values (paramName=defaultValue).

Example: Creating a method that calculates the average of two numbers:

```
<#function avg x y>
  <#return (x + y) / 2>
</#function>
${avg(10, 20)}
```

produces this output:

```
15
```

Example: Creating a method that calculates the average of multiple numbers:

```
<#function avg nums...>
  <#local sum = 0>
  <#list nums as num>
    <#local sum = sum + num>
  </#list>
  <#if nums?size != 0>
    <#return sum / nums?size>
  </#if>
</#function>
${avg(10, 20)}
${avg(10, 20, 30, 40)}
${avg()!"N/A"}
```

produces this output:

```
15
25
N/A
```

global

```
<#global name=value>
or
<#global name1=value1 name2=value2 ... nameN=valueN>
or
<#global name>
  capture this
</#global>
```

Parameter	Description
name	The name of the variable. Not a expression. However, it can be written as a string literal, which is useful if the variable name contains reserved characters, for example `<#assign "foo-bar" = 1>`. Note that this string literal does not expand interpolations (as "${foo}").
value	The value to store. This is an expression.

This directive is similar to `assign`, but the variable created will be visible in all namespaces, and will not be inside any namespace as if you created or replaced a variable of the data model. Hence, the variable is global. If a variable with the same name already exists in the data model, it will be hidden by the variable created with this directive. If a variable with the same name already exists in the current namespace, it will hide the variable created with this directive.

For example, you create a variable with `<#global x = 1>` that is visible as x in all namespaces, unless another variable called x hides it (for example a variable created as `<#assign x = 2>`). In this case, you can use the special variable `.globals`, as `${.globals.x}`. Note that with `.globals`, you see all globally accessible variables, including the variables of the data model.

if, else, elseif

```
<#if condition>
  ...
<#elseif condition2>
  ...
<#elseif condition3>
  ...
...
<#else>
  ...
</#if>
```

Parameter	Description
condition, condition2, …etc.	Expression evaluates to a boolean value.

Use these directives to conditionally skip a section of the template. The conditions must evaluate to a boolean value. Otherwise, an error occurs that terminates template processing. The `elseif`(s) and `else`(s) must occur between the start and end tags of `if`. `if` can contain any number of `elseif`(s) and, optionally, one `else` at the end.

Examples:

`if` with no `elseif` and no `else`:

```
<#if x == 1>
  x is 1
</#if>
```

`if` with no `elseif` and an `else`:

```
<#if x == 1>
  x is 1
<#else>
  x is not 1
</#if>
```

`if` with 2 `elseif` and no `else`:

```
<#if x == 1>
  x is 1
<#elseif x == 2>
  x is 2
<#elseif x == 3>
  x is 3
</#if>
```

`if` with 3 `elseif` and an `else`:

```
<#if x == 1>
  x is 1
<#elseif x == 2>
  x is 2
<#elseif x == 3>
  x is 3
<#elseif x == 4>
  x is 4
<#else>
  x is not 1 nor 2 nor 3 nor 4
</#if>
```

You can nest `if` directives, as shown in the following example:

```
<#if x == 1>
  x is 1
  <#if y == 1>
    and y is 1 too
  <#else>
    but y is not
  </#if>
<#else>
  x is not 1
  <#if y < 0>
    and y is less than 0
  </#if>
</#if>
```

IMPORTANT: To test whether x is greater than 1, write either `<#if (x > 1)>` or `<#if x &gt; 1>`. Note that `<#if x > 1>` is incorrect, as RPL will interpret the first > as the end of the tag.

list, break

```
<#list sequence as item>
    ...
</#list>

or

<#list sequence as item>
    ...
    <#break>
    ...
</#list>
```

Parameter	Description
sequence	Expressions evaluates to a sequence or collection.
item	Name of the loop variable (not an expression).

Use this directive to process a section of the template for each variable of a sequence. The code between the start and end tags will be processed for the first sub-variable, then for the second sub-variable, etc. until it passes the last one. For each such iteration, the loop variable will contain the current sub-variable.

Two special loop variables are available inside the list loop:

item_index

This is a numerical value that contains the index of the current item being stepped over in the loop.

item_has_next

A boolean value that specifies whether or not the current item the last in the sequence.

Example:

```
<#assign seq = ["winter", "spring", "summer", "autumn"]>
<#list seq as x>
  ${x_index + 1}. ${x}<#if x_has_next>,</#if>
</#list>
```

produces this output:

```
  1. winter,
  2. spring,
  3. summer,
  4. autumn
```

You can use this directive to count between numbers, using a numerical range sequence expression, for example:

```
<#assign x=3>
<#list 1..x as i>
  ${i}
</#list>
```

produces this output:

```
  1
  2
  3
```

Note that the above example will not work as you may expect if x is 0, as it will print 0 and -1.

You can leave the `list` loop before it passes the last sub variable of the sequence using the `break` directive. For example, to print only *winter* and *spring*:

```
<#list seq as x>
  ${x}
  <#if x = "spring"><#break></#if>
</#list>
```

You can also use this directive to create a comma separated list manually. Note that the `join` builtin can produce the same list with fewer lines. For example, if the sequence animals contains [“giraffe”, “lion”, “eagle”], then:

```
<#assign readable=””>
<#list animals as animal>
  <#assign readable=readable+animal
  <#if animal_has_next>
    <#assign readable=readable + “, ”:
  <#else>
    <#assign readable=readable + “, and”>
  <#/if>
<#/list>
```

produces this output:

```
giraffe, lion, and tiger
```

The example above illustrates using the list directive. The same output can be achieved with the `join` builtin as follows:

```
<#assign readable=animals?join(", ", ", and ")>
```

import

```
<#import path as hash>
```

Parameter	Description
path	The path of a template. This is an expression that evaluates to a string.
hash	The unquoted name of hash variable by which you can access the namespace. Not an expression.

Imports a library. The directive creates a new empty namespace, then executes the template given with the `path` parameter in that namespace so the template populates the namespace with variables (macros, functions, ...etc.). Then, the directive makes the newly created namespace available to the caller with a hash variable. The hash variable will be created as a plain variable in the namespace used by the caller of `import` (as if you would create it with the `assign` directive), with the name given with the `hash` parameter.

Calling `import` multiple times with the same path creates the namespace and runs the template only for the first call. Subsequent calls will create a hash by which you can access the same namespace.

The template is executed to populate the namespace with variables, not to write to the output. Therefore, the output produced by the imported template will be ignored (will not be inserted at the place of importing).

Example:

```
<#import "/contentlibrary/libs/mylib.htm" as my>

<@my.copyright date="1999-2002"/>
```

The `path` parameter can be a relative path such as "foo.htm" and "../foo.htm", or an absolute path such as "/contentlibrary/foo.htm". Relative paths are relative to the directory of the template that uses the directive.

NOTE: Always use the forward slash (/) to separate path components, never the back slash (\). If you are loading templates from your local file system and it uses backslashes, RPL will convert them automatically.

Similarly to the `include` directive, you can use acquisition and localized lookup for resolving the path.

For information about namespaces, see *"Chapter 8: Namespace Reference"*.

include

```
<#include path>
or
<#include path options>
```

Parameter	Description
path	The path of the file to include; an expression that evaluates to a string.
options	One or more of the following: **encoding=encoding** Expression evaluates to a string **parse=parse** Expression evaluates to Boolean **cleanup=cleanup** Expression evaluates to Boolean **cleanupmode=cleanupmode** Expression evaluates to a string For more information about these options, see *"Supported Options"* further in this section.

You can use this directive to insert the RPL template file specified by the `path` parameter into the template.

The output from the included template is inserted at the point where the include tag occurs. The included file shares the variables with the including template, similarly as if it was copy/pasted into it.

The `include` directive does not replace the content of the included file, it just processes the included file each time RPL reaches the include directive during template processing. For example, if you use `include` inside a listloop, you can specify different file names in each cycle.

The `path` parameter can be a relative path such as "foo.htm" and "../foo.htm", or an absolute path such as "cms://contentlibrary/foo.htm". Relative paths are relative to the directory of the template that uses the import directive.

NOTE: Always use the forward slash (/) to separate path components, never the back slash (\).

Example:

If /common/copyright.htm contains:

```
Copyright 2001-2002 ${me}<br>
All rights reserved.
```

then the following example:

```
<#assign me = "Juila Smith">
<h1>Some test</h1>
<p>Yeah.
<hr>
<#include "cms://contentlibrary/common/copyright.htm">
```

produces this output:

```
<h1>Some test</h1>
<p>Yeah.
<hr>
Copyright 2001-2002 Juila Smith
All rights reserved.
```

Supported options

parse

If `true`, the included file will be parsed as RPL. Otherwise, the file will be considered as simple text. If you omit this option, the default is `true`.

encoding

The included file inherits the encoding (charset) of the including template, unless you specify an encoding with this option. Encoding names are the same as those supported by java.io.InputStreamReader (as of Java API 1.3: MIME-preferred charset names from the IANA Charset Registry). Examples of valid names: ISO-8859-2, UTF-8, Shift_JIS, Big5, EUC-KR, GB2312.

cleanup

In some cases, the contents of the included file is a complete HTML file, with

HTML and body tags in it. Inserting such a file into the destination stream might result in a file with double HTML and body tags. The effect of this double body becomes apparent when the footers are inserted after personalization: the footer might be placed right on top of the wrong closing body tag. http/https included files are usually retrieved with contents in this way. The `cleanup` flag allows the extraction of all content between the included file's body tags, thus eliminating any extra HTML, head, or body tags. Note that an inclusion of a file with no body tags will result in empty content. For this reason, use this flag only when you are sure that the content contains a body tag.

cleanupmode
In some cases, the content of the included file does not contain an html <body> tag. In such cases, if the `cleanup` parameter is set to `true`, it will return an empty document by default. If you want to return either the original content or empty content, use the `cleanupmode` parameter which can take either of two values: "empty" or "original".

If `cleanupmode` is not specified, the default value is "empty".

If "empty" is specified, or no `cleanupmode` is specified, `include` returns an empty document if the content does not have a <body> tag.

If "original" is specified, the original content is returned if the content does not have a <body> tag.

Example:

```
<#include "cms://contentlibrary/common/navbar.html" parse=false
encoding="Shift_JIS" cleanup=true cleanupmode="original">
```

Using acquisition
Acquisition allows you to place commonly included files in a parent directory, and place them in sub-directories as needed. The including template will then acquire the template to include from the parent directory.

To use acquisition, use an asterisk (*) to represent a directory and any of its parents. For example, if the template is located in /contentlibrary/foo/bar/template.htm , this line:

```
<#include "*/footer.htm">
```

searches for the template in the following locations, in the order shown:

1. /contentlibrary/foo/bar/footer.htm
2. /contentlibrary/foo/footer.htm
3. /contentlibrary/footer.htm

Note that you can specify not only a template name to the right of the asterisk, but a sub-path as well. For example, this line:

```
<#include "*/commons/footer.htm">
```

searches for the template in following locations, in the order shown:

1. /contentlibrary/foo/bar/commons/footer.htm
2. /contentlibrary/foo/commons/footer.htm
3. /contentlibrary/commons/footer.htm

Finally, * does not have to be the first element in the path, as shown in the following example:

```
<#include "commons/*/footer.htm">
```

This searches for the template in following locations, in the order shown:

1. /contentlibrary/foo/bar/commons/footer.htm
2. /contentlibrary/foo/bar/footer.htm
3. /contentlibrary/foo/footer.htm
4. /contentlibrary/footer.htm

The path can include only one asterisk. Specifying more than one asterisk will result in the template not being found.

Localized lookup

When a template is loaded, it is assigned a locale. A locale is a language and an optional country or dialect identifier. A template is typically loaded by and the locale is chosen based on the profile.

When a template includes another template, it attempts to load a template with the same locale. For example, your template was loaded with locale en_US, which means U.S. English. When you include another template:

```
<include "footer.htm">
```

the engine will look for several templates, in the order shown:

1. footer_en_US.htm
2. footer_en.htm
3. footer.htm

When you use both acquisition and localized template lookup, the template with a more specific locale in a parent directory takes precedence over template with a less specific locale in a child directory. For example, if you use the following include from /foo/bar/template.htm:

```
<#include "*/footer.htm">
```

RPL will look for these templates, in the order shown:

1. /foo/bar/footer_en_US.htm
2. /foo/footer_en_US.htm
3. /footer_en_US.htm
4. /foo/bar/footer_en.htm
5. /foo/footer_en.htm
6. /footer_en.htm
7. /foo/bar/footer.htm
8. /foo/footer.htm
9. /footer.htm

Loading from the Content Library

In most cases, the templates are stored in the Content Library. You can include additional documents in other scenarios as described below.

Using the full path referencing the document

In this case, specify your import as:

```
<#import "cms://full-content-library-path">
```

The full content path is described using the root as /contentlibrary/ and thus the previous example will look this:

```
<#import "cms://contentlibrary/sub-content-library-path">
```

Using a document that includes a document in the same directory, a subdirectory, or a directory right above it.
In this case, you can import the document specifying a root or the cms:// prefix. RPL will ensure that the document is located using the path of the current template. For example:

```
<#import "document2.htm">
Or
<#import "subdir/document3.htm">
Or
<#import "../siblingdir/document4.htm">
```

The nomenclature for .. means one directory above the current directory.

Using a document including a document in an unspecified subdirectory
See the section of content acquisition above.

Loading from urls
To be able to download content from urls, your account must be enabled to download from http and https.

In cases when content is stored content remotely, i.e. in a web server outside of the system, use the http:// or https:// prefix and the full path name of the remote template.

We recommend that you do not store content on your web site because it can open your campaign to malicious attacks. In addition, you can cause a "denial-of-service" attack to your web server by repeatedly requesting a template during a launch with many records.

To prevent malicious attacks, RPL tries to optimize access to remote sites by following the commonly used HTTP/1.1 protocol. It is very important that the web site is configured correctly to avoid repeated calls. You can optimize content retrieval using either the *expiration model* or the *validation model*.

Expiration model
In the expiration model, content remains valid until a specified time. RPL will not try to retrieve such content until the expiration time has been reached, keeping the content available for personalization.

Validation model
The validation model works after prior content has been retrieved, the expiration time is reached, or the expiration time has not been set. RPL will send the last retrieval time (or the last modified date as specified by the web server), and the web server can respond with new content or notify RPL that the current content has not been modified. When using the validation model without expiration, the same warnings apply regarding “denial-of-service”.

It is always recommended to ensure that the web server correctly sends the expiration time. As a safeguard, when no expiration time is returned, the system will assume one hour, which is appropriate for most launch scenarios. If one hour is not sufficient for your needs, make sure that your web server sets the expiration to the time you need.

In addition, when the web server does not send the last modified time, for validation purposes, RPL will request the content to the remote repository with the time that the last content was retrieved.

Note that for url-based content, it is often recommended to use the `cleanup` flag to remove any html, head, or body tags from the source content, but only if the content contains a proper body tag.

join

```
<#join>
…
</join>
or
<#join separator-expression>
…
</#join>
```

Parameter	Description
separator-expression	A string expression that specifies the string to be placed instead of new lines.

Strips new line from the output of the block that it encompasses. This is very useful when dealing with blocks that result in text-only output, and proper formatting of RPL is desired. It is very helpful in conjunction with the `compress` directive.

Example 1:

```
<#assign sequence=["a", "b", "c", "d", "e", "f"]>
<#join>
<#compress>
<#list sequence as item>
  ${item}
</#list>
</#compress>
</#join>
```

produces this output:

```
abcdef
```

Example 2:

```
<#assign sequence=["a", "b", "c", "d", "e", "f"]>
<#join "-">
<#compress>
<#list sequence as item>
  ${item}
</#list>
</#compress>
</#join>
```

produces this output:

```
a-b-c-d-e-f
```

local

```
<#local name=value>
or
<#local name1=value1 name2=value2 ... nameN=valueN>
or
<#local name>
  capture this
</#local>
```

Parameter	Description
name	The name of the variable. This is not an expression. However, you can write it as a string literal, which is useful if the variable name contains reserved characters, for example `<#assign "foo-bar" = 1>`. Note that this string literal does not expand interpolations (as `"${foo}"`).
Value	The value to store. This is an expression.

This directive is similar to the `assign` directive, but it creates or replaces local variables. This directive works only inside macro and function definitions.

macro, nested, return

```
<#macro name param1 param2 ... paramN>
  ...
  <#nested loopvar1, loopvar2, ..., loopvarN>
  ...
  <#return>
  ...
</#macro>
```

Parameter	Description
name	The name of the macro variable. This is not an expression. However, it can be written as a string literal, which is useful if the macro name contains reserved characters, for example `<#macro "foo-bar">`. Note that this string literal does not expand interpolations (as `"${foo}"`).
param1, param2, ...etc.	The name of the local variables that store the parameter values (not an expression), optionally followed by = and the default value (an expression). The default value can be another parameter, for example `<#macro section title label=title>`.
paramN	Optionally, the last parameter can include a trailing ellipsis (...). The ellipsis indicate that the macro takes a variable number of parameters. If called using named parameters, `paramN` will be a hash containing all undeclared key/value pairs passed to the macro. If called using positional parameters, `paramN` will be a sequence of the extra parameters.
loopvar1, loopvar2, ...etc.	Optional. The values of loop variables that the nested directive should create for the nested content. These are expressions.

Creates a macro variable in the current namespace. For more information about macros, see *“Defining Your Own Directives”*.

A macro variable stores a template fragment (called *macro definition body*) that can be used as user-defined directive. The variable also stores the name of allowed parameters to the user-defined directive. When using the variable as a directive, you must give value for all parameters that do not have a default value. The default value will be used only if you do not provide a value for the parameter when calling the macro.

The `return` and `nested` directives are optional and can be used anywhere and any number of times between the <#macro ...> and </#macro>.

Parameters without a default value must precede parameters with a default value (paramName=defaultValue).

The variable will be created at the beginning of the template regardless of where the macro directive is placed in the template. For example:

```
<#-- call the macro; the macro variable is already created: -->
<@test/>
...

<#-- create the macro variable: -->
<#macro test>
  Test text
</#macro>
```

However, if the macro definitions are inserted with the `include` directive, they will not be available until RPL has executed the include directive.

Example: Macro without parameters:

```
<#macro test>
  Test text
</#macro>
<#-- call the macro: -->
<@test/>
```

produces this output:

```
  Test text
```

Example: Macro with parameters:

```
<#macro test foo bar baaz>
  Test text, and the params: ${foo}, ${bar}, ${baaz}
</#macro>
<#-- call the macro: -->
<@test foo="a" bar="b" baaz=5*5-2/>
```

produces this output:

```
  Test text, and the params: a, b, 23
```

Example: Macro with parameters and default parameter values:

```
<#macro test foo bar="Bar" baaz=-1>
  Test text, and the params: ${foo}, ${bar}, ${baaz}
</#macro>
<@test foo="a" bar="b" baaz=5*5-2/>
<@test foo="a" bar="b"/>
<@test foo="a" baaz=5*5-2/>
<@test foo="a"/>
```

produces this output:

```
  Test text, and the params: a, b, 23
  Test text, and the params: a, b, -1
  Test text, and the params: a, Bar, 23
  Test text, and the params: a, Bar, -1
```

Example: A more complex macro:

```
<#macro list title items>
  <p>${title?cap_first}:
  <ul>
    <#list items as x>
      <li>${x?cap_first}
    </#list>
  </ul>
</#macro>
<@list items=["mouse", "elephant", "python"] title="Animals"/>
```

produces this output:

```
  <p>Animals:
  <ul>
      <li>Mouse
      <li>Elephant
      <li>Python
  </ul>
```

Example: A macro with support for a variable number of named parameters:

```
<#macro img src extra...>
  <img src="/context${src?html}"
  <#list extra?keys as attr>
    ${attr}="${extra[attr]?html}"
  </#list>
  >
</#macro>
<@img src="/images/test.png" width=100 height=50 alt="Test"/>
```

produces this output:

```
  <img src="/context/images/test.png"
    alt="Test"
    height="50"
    width="100"
  >
```

nested

Executes the template fragment between the start and end tags of a user-defined directive. The directive can contain anything that is valid in templates, such as interpolations and directives.

This directive is executed in the context from where the macro is called, rather than in the context of the macro definition body. Thus, for example, the local variable of the macro does not appear in the nested part. If you do not call the `nested` directive, the part between the start and end tags of the user-defined directive will be ignored.

Example:

```
<#macro do_twice>
  1. <#nested>
  2. <#nested>
</#macro>
<@do_twice>something</@do_twice>
```

produces this output:

```
  1. something
  2. something
```

The nested directive can create loop variables for the nested content, for example:

```
<#macro do_thrice>
  <#nested 1>
  <#nested 2>
  <#nested 3>
</#macro>
<@do_thrice ; x>
  ${x} Anything.
</@do_thrice>
```

produces this output:

```
  1 Anything.
  2 Anything.
  3 Anything.
```

A more complex example:

```
<#macro repeat count>
  <#list 1..count as x>
    <#nested x, x/2, x==count>
  </#list>
</#macro>
<@repeat count=4 ; c, halfc, last>
  ${c}. ${halfc}<#if last> Last!</#if>
</@repeat>
```

produces this output:

```
  1. 0.5
  2. 1
  3. 1.5
  4. 2 Last!
```

return

Allows you to leave a macro or function definition body anywhere. For example:

```
<#macro test>
  Test text
  <#return>
  Will not be printed.
</#macro>
<@test/>
```

produces this output:

```
  Test text
```

noparse

```
<#noparse>
  ...
</#noparse>
```

Prevents RPL from searching RPL tags, interpolations and other special sequences within the body of this directive. For example:

```
<#noparse>
  <#list animals as being>
  <tr><td>${being.name}<td>${being.price} Euros
  </#list>
</#noparse>
```

produces this output:

```
  <#list animals as being>
  <tr><td>${being.name}<td>${being.price} Euros
  </#list>
```

Note that HTML readers such as web browsers might not render RPL properly. This is caused by the fact that the HTML reader expects only HTML, and RPL is not properly formed HTML. Specifically, it has been observed that the readers usually omit closing RPL tags such as (</#list>).

To properly display RPL in an HTML reader, you can write the previous example as:

```
<#assign unparsed>
<#noparse>
  <#list animals as being>
  <tr><td>${being.name}<td>${being.price} Euros
  </#list>
</#noparse>
</#assign>
${unparsed?html}
```

This example uses `assign` to keep the output that will be displayed in a variable called *unparsed*. It then encodes and outputs this variable as HTML.

nt

```
<#nt>
```

Disables white space stripping in the line where it occurs. It also disables the effect of other trim directives occurring in the same line (the effect of t, rt, lt).

rpl

```
<#rpl param1=value1 param2=value2 ... paramN=valueN>
```

Parameter	Description
param1, param2, ...etc.	Name of the parameter. This is not an expression. See the *"Supported parameters"* table below for valid parameters.
value1, value2, ...etc.	The value of the parameter. This must be a constant expression (as true, or "ISO-8859-5", or {x:1, y:2}). It cannot use variables.

Supported parameters

Parameter	**Description**
encoding	Specifies the encoding (charset) of the template in the template file. This will be the encoding setting of the newly created template, and not even the encoding parameter to Configuration.getTemplate can override it. Note however, that RPL will try to find and interpret the RPL directive first with the automatically guessed encoding. After that, if the RPL directive dictates something different, RPL will re-read the template with the new encoding. This means that the template must be valid RPL until the end of the RPL tag with the encoding tried first. The valid values of this parameter are MIME-preferred charset names from the IANA Charset Registry, such as ISO-8859-5, UTF-8 or Shift_JIS.
strip_whitespace	Enables or disables white space stripping. Valid values are `true` and `false`, and strings "yes", "no", "true", "false"). When this parameter is not specified, the default value is `true`.
strip_text	When enabled, all top-level text in a template is removed when the template is parsed. This does not affect text within macros,

Parameter	Description
	directives, or interpolations. Valid values are `true` and `false`. When this parameter is not specified, the default value is `false`.
ns_prefixes	A hash that associates prefixes with node namespaces. For example: `{"e":"http://example.com/ebook", "vg":"http://example.com/vektorGraphics"}`. This is used mostly with XML processing where the prefixes can be used in XML queries, but it also influences the working of the `visit` and `recurse` directives. There is a one-to-one relation between prefixes and node namespaces. This means that only one prefix can be registered for the same node namespace; otherwise an error will occur. Prefixes D and N are reserved. If you register prefix D, then in addition to associating the node namespace with prefix D, you are setting the default node namespace. You cannot register prefix N because it is used to denote nodes with no node namespace in certain places when prefix D is registered. For more information, see *"Chapter 4. XML Processing"* and the `visit` and `recurse` directives. The effect of `ns_prefixes` is limited to a single RPL namespace, namely, to the RPL namespace that was created for the template. This also means that `ns_prefixes` is used only when an RPL namespace is created for the template that contains it; otherwise, this parameter has no effect. An RPL namespace is created for a template when: the template is the main template, that is, it is not invoked as a result of an <#include ...>) OR the template is invoked directly with <#import ...>.
Attributes	A hash that associates arbitrary attributes (name/value pairs) to the template. The values of the attributes can be of any type.

This directive provides information about the template for RPL and other tools, and helps programs to automatically detect whether a text file is an RPL file.

IMPORTANT: This directive, if present, must be the very first thing in the template. Any white space before this directive will be ignored.

The settings given here have the highest precedence. This means that they will be used for the template regardless of any other RPL configuration settings.

setting

```
<#setting name=value>
```

Parameter	Description
name	The name of the setting. This is not an expression.
value	New value of the setting. This can be an expression.

Sets values that influence RPL behavior for further processing. The new value will be present only in the template processing where it was set, and does not affect the template itself. The initial value is set by the system.

Supported settings

Setting	Description
locale	The locale (language) of the output. The locale can influence the presentation format of numbers, dates, etc. The value is a string which consists of a language code (lowercase two-letter ISO-639 code) plus an optional county code (uppercase two-letter ISO-3166 code) separated from the language code by an underscore. If you specify the country, you can also specify an optional variant code (not standardized) separated from the country by an underscore. For example: *en*, *en_US*, *en_US_MAC*. RPL will try to use the most specific available locale. This means that if you specify *en_US_MAC* and MAC is unknown, RPL will try *en_US*, then *en*, then the default locale of the computer (which is set by the system).
number_format	The number format to use to convert numbers to strings when no explicit format is specified. This can be one of predefined values: number (the default), computer, currency, or percent. Additionally, you can specify an arbitrary format pattern written in Java decimal number format syntax. For more information about format patterns, see the `string` built-in.

Setting	Description
boolean_format	The comma-separated pair of strings to use for representing *true* and *false* when converting booleans to strings with no explicitly specified format. The default value is "true,false".
date_format, time_format, datetime_format	The date/time format to use for converting dates to strings when no explicit format is specified, for example `${someDate}`. date_format affects the formatting of date-only dates (year, month, day). time_format affects the formatting of time-only dates (hour,minute, second, millisecond). datetime_format affects the formatting of date-time dates (year, month, day, hour, minute, second, millisecond). The possible values of the settings are similar to the parameters of string built-in of dates, for example, "short", "long_medium", "MM/dd/yyyy".
time_zone	The name of the time zone to use for formatting times for display. This can be any value that is accepted by Java TimeZone API. For example, "GMT", "GMT+2", "GMT-1:30", "CET", "PST", "America/Los_Angeles". By default, the system time zone is used.
url_escaping_charset	The charset to use for URL escaping (e.g. for ${foo?url}) to calculate the escaped (%XX) parts. The framework that encloses RPL usually sets the charset, so this setting is rarely used.

Example: Assuming that the initial locale of template is hu (Hungarian), the following example:

```
${1.2}
<#setting locale="en_US">
${1.2}
```

produces this output, because Hungary uses the comma as their decimal separator:

```
1,2
1.2
```

skip

```
<#skip>
or
<#skip reason>
```

Parameter	Description
reason	Informative message about the reason for termination. Expression evaluates to string.

Stops the personalization of a record in an email campaign and, if possible, continues with the next record. The directive will stop executing the record immediately and return control to the personalization engine.

Skip calls are reported as skip events and are added to the events database. The reason is provided in the event itself. For information about the event system, see Interact Help and training materials.

switch, case, default, break

```
<#switch value>
  <#case refValue1>
    ...
    <#break>
  <#case refValue2>
    ...
    <#break>
  ...
  <#case refValueN>
    ...
    <#break>
  <#default>
    ...
</#switch>
```

Parameter	Description
value, refValue1, etc.:	Expressions that evaluate to scalars of the same type.

IMPORTANT: We do not recommend using these directives, as they are error-prone because of the fall-through behavior. Unless you want to exploit the fall-through behavior, use elseif instead.

`switch` is used to choose a fragment of template depending on the value of an expression. For example:

```
<#switch being.size>
  <#case "small">
     This will be processed if it is small
     <#break>
  <#case "medium">
     This will be processed if it is medium
     <#break>
  <#case "large">
     This will be processed if it is large
     <#break>
  <#default>
     This will be processed if it is neither
</#switch>
```

One or more `<#case value>` is required inside the switch and, optionally, one `<#default>` after all case tags.

When RPL reaches a case where `refValue` equals the value, it processes that case and continues processing the template. If there is no `case` directive with an appropriate value, RPL continues processing at the default directive if that exists; otherwise, it continues the processing after the end tag of `switch`. Note that when RPL has chosen a `case` directive, it will continue processing until it reaches a `break` directive. That is, it will not automatically leave the switch directive when it reaches another `case` directive or the `<#default>` tag.

Example:

```
<#switch x>
  <#case x = 1>
    1
  <#case x = 2>
    2
  <#default>
    d
</#switch>
```

if x is 1, the output will be *1 2 d*; if x is 2 , the output will be *2 d*; if x is 3, the output will be d. The break tag instructs RPL to immediately skip past the `switch` end tag.

## t, lt, rt

```
<#t>

<#lt>

<#rt>

<#nt>
```

These directives instruct RPL to ignore certain whitespace in the line of the tag:

t (trim)

Ignores all leading and trailing white space in this line

lt (left trim)

Ignores all leading white space in this line

rt (right trim)

Ignores all trailing white space in this line

Leading white space is all space and tab characters (and other characters that are white space according to UNICODE, except line breaks) before the first non-white space character of the line.

Trailing white space is all space and tab characters (and other characters that are white space according to UNICODE, except line breaks) after the last non-white space character of the line, and the line break at the end of the line.

You can use these directives anywhere within the line.

These directives examine the template, not the output that the template generates when you merge it with the data model. This means that white space removal happens at parse time.

For example:

```
--
  1 <#t>
  2<#t>
  3<#lt>
  4
  5<#rt>
  6
--
```

produces this output:

```
--
1 23
  4
  5   6
--
```

user-defined directive

```
<@user_def_dir_exp param1=val1 param2=val2 ... paramN=valN/>
(Note the XML-style / before the >)
or if you need loop variables (more details...)
<@user_def_dir_exp param1=val1 param2=val2 ... paramN=valN ; lv1,
lv2, ..., lvN/>

Or the same as the above two but with end-tag (more details...):

<@user_def_dir_exp ...>
  ...
</@user_def_dir_exp>
or
<@user_def_dir_exp ...>
  ...
</@>

Or all above but with positional parameter passing (more
details...):

<@user val1, val2, ..., valN/>
...etc.
```

Parameter	Description
user_def_dir_exp	Expression that evaluates to an user-defined directive such as a macro that will be called.
param1, param2, ...etc.	Parameter names. These are not expressions.
val1, val2, ...etc.	Parameter values. These are expressions.
lv1, lv2, ...etc.	Loop variable names. These are not expressions.

Calls a user-defined directive, for example a macro. The meaning of parameters and the set of supported and required parameters depend on the directive.

The following rules apply to parameters:

- The call can have zero parameters.
- Parameters can be specified in any order, unless you use positional parameter passing.
- Parameter names must be unique.
- Parameter names are case sensitive.

Example 1: Call the directive stored in the variable *html_escape*:

```
<@html_escape>
  a < b
  Romeo & Juliet
</@html_escape>
```

produces this output:

```
  a &lt; b
  Romeo &amp; Juliet
```

Example 2: Call a macro with parameters:

```
<@list items=["mouse", "elephant", "python"] title="Animals"/>
...
<#macro list title items>
  <p>${title?cap_first}:
  <ul>
    <#list items as x>
      <li>${x?cap_first}
    </#list>
  </ul>
</#macro>
```

produces this output:

```
  <p>Animals:
  <ul>
      <li>Mouse
      <li>Elephant
      <li>Python
  </ul>
```

End tag

You can omit the `user_def_dir_exp` in the end tag. That is, you can write </@> instead of </@anything>. This rule is useful mostly when the `user_def_dir_exp` expression is too complex, because you do not have to repeat the expression in the end tag. Furthermore, if the expression contains anything except simple variable names and dots, you are not allowed to repeat the expression. For example, <@a_hash[a_method()]>...</@a_hash[a_method()]> is an error; you must write <@a_hash[a_method()]>...</@>. But <@a_hash.foo>...</@a_hash.foo> is correct.

Loop variables

Some user-defined directives create loop variables, similarly to the `list` directive. As with the predefined directives, the name of loop variables is given when you call the directive, while the value of the variable is set by the directive itself. In the case of user-defined directives, the name of loop variables is given after a semicolon. For example:

```
<@myRepeatMacro count=4 ; x, last>
  ${x}. Something... <#if last> This was the last!</#if>
</@myRepeatMacro>
```

Note that the number of loop variable created by the user-defined directive and the number of loop variables specified after the semicolon does not need to match. For example, if you are not interested if the repetition is the last one, you can simply write:

```
<@myRepeatMacro count=4 ; x>
  ${x}. Something...
</@myRepeatMacro>
```

or:

```
<@myRepeatMacro count=4>
  Something...
</@myRepeatMacro>
```

Specifying more loop variables after the semicolon than the user-defined directive creates does not cause an error. In this case, the extra loop variables will not be created (i.e. those will be undefined in the nested content). Trying to use the undefined loop variables, however, will cause error unless you use built-ins such as `?default` because you will be trying to access a non-existing variable.

Positional parameter passing

Positional parameter passing is currently supported only for macros.

Positional parameter passing (such as `<@heading "Preface", 1/>`) is a shorthand form of named parameter passing (such as `<@heading title="Preface" level=1/>`) where you omit the parameter name. Use positional parameter passing if a user-defined directive has only one parameter, or if it is easy to remember the order of parameters for a frequently used user-defined directive. To use this form, you have to know the order in which the named parameters are declared. For example, if a heading was created as `<#macro heading title level>...`, then `<@heading "Preface", 1/>` is equivalent to `<@heading title="Preface" level=1/>` or `<@heading level=1 title="Preface"/>`.

visit, recurse, fallback

```
<#visit node using namespace>
or
<#visit node>
<#recurse node using namespace>
or
<#recurse node>
or
<#recurse using namespace>
or
<#recurse>
<#fallback>
```

Parameter	Description
node	Expression that evaluates to a node variable.
namespace	A namespace or a sequence of namespaces. A namespace can be given with the namespace hash (known as the *gate hash*), or with a string literal that stores the path of template that can be imported. Instead of namespace hashes, you can use plain hashes as well.

The `visit` and `recurse` directives are used for the recursive processing of trees. These are used mostly for processing XML.

visit

When you call `<#visit node>`, it looks for a user-defined directive (such as a macro) to invoke that has the name deducted from the node's name (node?node_name) and namespace (node?node_namespace).

Rules of name deduction

If the node does not support node namespaces (as text nodes in XML), then the directive name is simply the name of the node (node?node_name). A node does not support node namespaces if the `getNodeNamespace` method returns null.

If the node does support node namespaces (as element nodes in XML), then a prefix deduced from the node namespace may be appended before the node name with a colon (:) used as separator, for example *e:book*. The prefix, if one is used, depends on which prefixes have been registered with the `ns_prefixes` parameter of the `rpl` directive in the RPL namespace where `visit` looks for the handler directive. This namespace is not necessary the same as the RPL namespace where visit was called from. If a default namespace was not registered with `ns_prefixes`, no prefix is used for nodes that do not belong to any namespace. If default namespace was registered with `ns_prefixes`, prefix N is used for nodes that do not belong to any namespace, and no prefix is used for nodes that belong to the default node namespace. Otherwise, in both case, the prefix associated to the node namespace with the ns_prefixes is used. If a prefix is not associated to the node namespace of the node, `visit` behaves as if no directive was found with the proper name.

The node for which the user-defined directive was invoked is available for it as a special variable `.node`. For example:

```
<#-- Assume that nodeWithNameX?node_name is "x" -->
<#visit nodeWithNameX>
Done.
<#macro x>
   Now I'm handling a node that has the name "x".
   Just to show how to access this node: this node has
${.node?children?size} children.
</#macro>
```

produces output similar to this:

```
   Now I'm handling a node that has the name "x".
   Just to show how to access this node: this node has 3 children.
Done.
```

If one or more namespaces is specified using the optional `using` clause, then visit will look for the directives in those namespaces only, with the earlier specified namespaces in the list getting priority. If no `using` clause is specified, the namespace or sequence of namespaces specified with the using clause of the last uncompleted visit call is reused. If there is no such pending visit call, then the current namespace is used.

For example, if you execute this template:

```
<#import "n1.htm" as n1>
<#import "n2.htm" as n2>

<#-- This will call n2.x (because there is no n1.x): -->
<#visit nodeWithNameX using [n1, n2]>

<#-- This will call the x of the current namespace: -->
<#visit nodeWithNameX>

<#macro x>
  Simply x
</#macro>
```

and this is n1.htm:

```
<#macro y>
  n1.y
</#macro>
```

and this is n2.htm:

```
<#macro x>
  n2.x
  <#-- This will call n1.y, because it inherits the "using [n1, n2]"
from the pending visit call: -->
  <#visit nodeWithNameY>
  <#-- This will call n2.y: -->
  <#visit nodeWithNameY using .namespace>
</#macro>
<#macro y>
  n2.y
</#macro>
```

then this will print:

```
  n2.x
  n1.y
  n2.y

  Simply x
```

If `visit` does not find a user-defined directive in either RPL namespaces with the name identical to the name deduced with the rules described earlier, then it tries to find an user-defined directive with name @node_type. If the node does not support node type property (i.e. node?node_type returns undefined variable), then with name @default. For the lookup, it uses the same mechanism as was explained earlier. If it still does not find an user-defined directive to handle the node, then `visit` stops template processing with error. Some XML-specific node types have special handling in this regard.

```
<#-- Assume that nodeWithNameX?node_name is "x" -->
<#visit nodeWithNameX>

<#-- Assume that nodeWithNameY?node_type is "foo" -->
<#visit nodeWithNameY>

<#macro x>
Handling node x
</#macro>

<#macro @foo>
There was no specific handler for node ${node?node_name}
</#macro>
```

produces this output:

```
Handling node x

There was no specific handler for node y
```

recurse

The <#recurse> directive visits all children nodes of the node (and not the node itself). So, the following:

```
<#recurse someNode using someLib>
```

is equivalent to:

```
<#list someNode?children as child><#visit child using
someLib></#list>
```

However, `target node` is optional in the `recurse` directive. If the target node is unspecified, it simply uses the `.node`. Thus, the instruction <#recurse> is equivalent to:

```
<#list .node?children as child><#visit child></#list>
```

Note for those familiar with XSLT: <#recurse> is analogous to the <xsl:apply-templates/> instruction in XSLT.

fallback

The user-defined directive that handles the node may be searched in multiple RPL namespaces. The `fallback` directive can be used in a user-defined directive that was invoked to handle a node. It directs RPL to continue searching for the user-defined directive in further namespaces (that is, in the name spaces that are after the namespace of the currently invoked user-defined directive in the list of namespaces). If a handler for the node is found then it is invoked; otherwise `fallback` does nothing.

Typical usage of this to write a customization layer over a handler library that sometimes passes the handling to the customized library:

```
<#import "docbook.htm" as docbook>

<#--
  We use the docbook library, but we override some handlers
  in this namespace.
-->
<#visit document using [.namespace, docbook]>

<#--
  Override the "programlisting" handler, but only in the case if
  its "role" attribute is "java"
-->
<#macro programlisting>
  <#if .node.@role[0]!"" == "java">
    <#-- Do something special here... -->
    ...
  <#else>
    <#-- Just use the original (overidden) handler -->
    <#fallback>
  </#if>
</#macro>
```

Chapter 7. Method Reference

avg

```
avg(number1, number2, number3, …)
or
avg(numeric-list-expr)
```

Parameter	Description
number1, number2, number3, etc.	The numbers from which the average is computed.
numeric-list-expr	A sequence expression containing the numbers to be averaged.

Computes the average of the given numbers.

Example:

```
<#assign list=[1,73,22]>
${avg(list)}
${avg(1,73,22)}
```

produces this output:

```
32
32
```

NOTE: If you do not provide the numbers or if the sequence is empty `avg` produces an error, as the average is undefined. To catch that error, use `<#attempt>` and `<#recover>`.

bazaarvoiceauthstring

```
bazaarvoiceauthstring(key, query-string-expr)
```

Parameter	Description
key	A key/passphrase value that both the sending and receiving party know.
query-string-expr	The decoded and unencrypted query string.

Helps create links to the Bazaarvoice service.

To create a proper url, you need the base url and the key/passphrase. These will be provided to you in a document that encrypts the required key and query parameters.

Consult technical services for information about how to obtain these two elements, and for further details on how to utilize the Bazaarvoice service.

clickthrough

```
clickthrough(linkname)
or
clickthrough(linkname, parameter1, parameter2, parameter3, …)
```

Parameter	Description
linkname	An expression that identifies the link in the link table
parameter1, parameter2, parameter3, …	Additional values to be used in the tracking of the link, in one of the following formats: “datasourcealias.columnalias” For replacement of fields coming from the recipient record. “replacementname=value” For replacement of values coming from an expression. See below for an explanation of these values.

You can track clicks on links by first directing these links to a link tracking URL in Interact. Interact will redirect the request to the final URL. For more information, see *"Chapter 3. Working with Forms and Link Tracking"*.

In Mobile SMS campaigns, the `clickthrough` method returns a temporary shortened version of the link. The shortened URL is valid fo a short period of time. In email campaigns, this method returns a long URL.

Example:

Assume you want to track clicks to a link on the website *http://www.example.com*. To achieve this, you first need to set up and edit a link table for your campaign as follows:

LINK_NAME= Example
LINK_URL= http://www.example.com

This ensures that Interact is ready to track the link. Now, you need to point the document anchor to the link tracking URL in Interact using the `clickthrough` method.

The link in the user's email will look different from the original link. When a user clicks the link, `clickthrough` notifies Interact. Upon receiving the request, Interact redirects the user's browser to the original URL to get the content (in the example, the page located at *http://www.example.com*).

Use the following construct to create the link to Interact:

```
<a href="${clickthrough("Example")}">Click Here</a>
```

The `clickthrough` method creates a link that:

- Records the fact that a user clicked on it.
- Optionally, replaces fields in the link tracking table.
- Redirects the user to the link in the link table.

Sending additional information from content in the recipient record

Sending additional information from the recipient record to the receiving link is usually done on web sites that can receive parameters. For example, when a user clicks a link, you want to provide offers customized to the user's preferences. The link might look similar to the following:

http://www.example.com/offers?cid=<the-id-of-the-recipient>

To do this:

1. Create a link in the link tracking table with the following information:

   LINK_NAME=*Offers*
   LINK_URL=http://www.example.com/offers?cid=${profile.customerid}

2. Construct the `clickthrough` method as follows:

```
<a href="${clickthrough("Offers", "profile.customerid")}">Find
More Offers</a>
```

Note the use of quotes around the field name. This instructs `clickthrough` to use that field name to obtain the actual field name (in our example, this is CUSTOMER_ID_0 and its value). Without quotes, `clickthrough` would look for the field name in the *profile.customerid* field, an indirection form of the `clickthrough` method. In general you can use string expressions to specify the field name.

IMPORTANT: The field name resulting from the aliases used in `clickthrough` must match the field name in the LINK_URL.

Sending additional information from values in the template

To use a specific value from an expression in the template in the replacement of fields, set up the link table as follows:

LINK_NAME=Offers
LINK_URL=http://www.example.com/offers?cid=${RECOMMENDATION}

In this example, the replacement field is a user defined name that you must match in the `clickthrough` method, not a field in the database .

The following example illustrates how to recommend offers to the user based on whether the user has a recommendation in his record: if the user has a recommendation, use that recommendation; otherwise use the customer id.

The field with the recommendation is mapped as *pet.recommendationid* from a field RECOMMENDATION_ID in a profile extension table.

To do this, set up the clickthrough method as follows:

```
<#if pet.recommendationid != 0>
  <#assign value=pet.recommendationid>
<#else>
  <#assign value=profile.customerid>
</#if>
<a href="${clickthrough('Offers', 'RECOMMENDATION=' + value)}">Find
More Offers</a>
```

The expression:

'RECOMMENDATION=' + value

means that the replacement name is "RECOMMENDATION" (matching the field replacement in the LINK_URL). = indicates that the field has a value, and the value follows the =. The result of the expression looks similar to:

RECOMMENDATION=43853.

Now, the link will have the fields replaced. In the example, the resulting URL will be:

http://www.example.com/offers?cid=43853.

Tracking links to internal Interact forms

This section describes how to redirect to a form served by Interact. In the example, the form is called *userpreferences*.

For this example, the link table is set up as:

LINK_NAME= Preferences

LINK_URL=${form('userpreferences', {'usedb':true})}

The LINK_URL contains a `form` specification.

The template includes the following code:

```
<a href="${clickthrough('Preferences')}">User Preferences</a>
```

This `clickthrough` method creates a link to the click processor as in other cases. When the click is received, the `form` specification is executed and the user is redirected to the specified form, *userpreferences*.

Form parameters and implicit tracking of links

Assume the following link table setup:

LINK_NAME= Preferences

LINK_URL=${form('userpreferences', {}, "OPTION")}

This link loads the form *userpreferences* and sends the value of the OPTION parameter to the form. To do this, the `clickthrough` method must send the option to the `form` method as follows:

```
<a href="${clickthrough('Preferences', 'OPTION=o')}">User
Preferences</a>
```

The `clickthrough` method must send all required parameters in the `form` method, otherwise an error will occur. For more information about sending parameters, see *"Explicit external link tracking"*.

When the `form` method requires many parameters, keeping the `clickthrough` invocations in sync with the `form` method can become quite complex. You can simplify this by marking your personalization as requiring tracking, and then copying the LINK_URL and pasting it in the anchor element as follows:

```
<a href="${form('userpreferences', {}, 'profile.useroption')}">User
Preferences</a>
```

During personalization, the execution engine will recognize anchor tags and will replace this code:

```
${form('userpreferences', {}, 'profile.useroption')}
```

with this code:

```
${clickthrough('Preferences', 'profile.useroption')}
```

This mechanism of copy/pasting is used to simplify the maintenance of parameter matching. The `form` method is executed only during the click tracking process, it will not be executed during personalization.

converttimezone

```
converttimezone(date-expr, from-timezone, to-timezone)
```

Parameter	Description
date-expr	The base date that used for time zone conversion.
from-timezone	The string identifying the assumed timezone for the date expression.
to-timezone	The string identifying the target timezone for the conversion.

Converts a date to a target timezone. Dates do not assume a time zone. Conversions from a time zone to a destination time zone need to specify both the source and the destination time zones.

Due to technical limitations, RLP cannot always determine the date type. You can use `?date` or `?datetime` to properly specify the type of date that is used.

When using `?datetime`, or when the type is unknown, the timezone conversion will include hours. When using a date-only expression, the origination time will not have any time, but the result of the `converttimezone` invocation will include date and time. This resulting date-time will be adjusted according to the requested destination timezone. When using a time-only field, the conversion will cause an error. The following examples show the different variations.

Example:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${converttimezone(date, "America/Los_Angeles",
"Asia/Kolkata")?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-28 02:55:03
```

Example using a date-only expression:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${converttimezone(date, "America/Los_Angeles",
"Asia/Kolkata")?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-17 13:30:00
```

Note that parsing a date-only string ignores the time. The equivalent result date is then 2012-12-27 00:00:00. After that, the time is moved 12:30 hours forward.

The following example produces an error:

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${converttimezone(date, "America/Los_Angeles",
"Asia/Kolkata")?string("yyyy-MM-dd HH:mm:ss")}
```

dayadd

```
dayadd(date-expr, days-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
days-expr	The number of days to move forward or backwards. For backward offsets, use a negative number.

Adds or subtracts the number of days specified by `days-expr` to a base date specified by `date-expr`. To subtract days, specify a negative number in `days-expr`.

For example, adding 1.5 days is equivalent to adding one day and twelve hours.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`, `?date`, and `?time` to specify the date type.

Example: Advance one day forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${dayadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-28 13:25:03
2012-12-26 13:25:03
```

Example: Advance one day from a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-28 00:00:00
```

Example: Advance a time forward:

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${dayadd(date, 0.5)?string("HH:mm:ss")}
```

produces this output:

```
01:25:03
```

decrypt

```
decrypt(code)
or
decrypt(code, key)
```

Parameter	Description
code	The code previously produced by a call to encrypt.
key	The string that indicates the key that was used during encryption. This parameter is optional, but should match the key entered during encryption.

Decrypts text that was encrypted with the `encrypt` method.

Example:

```
${decrypt(encrypt("some text", "private-key"), "private-key")}
```

produces this output:

```
some text
```

emaildomain

```
emaildomain(email-expr)
```

Parameter	Description
email-expr	The email address.

A convenience method that extracts the string representing the domain from the email address.

Example:

```
${emaildomain("jamesbond@m5.com")
```

produces this output:

```
m5.com
```

encrypt

```
encrypt(text)
or
encrypt(text, key)
```

Parameter	Description
text	The expression or string constant for the text to encrypt.
key	The string that indicates the key used for encrypting. It is optional. The key same should be given for the equivalent `decrypt` method.

Encrypts the text that can only be decrypted by the equivalent `decrypt` method in a form.

Example:

```
${encrypt("some text")}
```

produces a string of the form:

```
E9uV7kzJm13p9SsE_SvvG8s
```

For security reasons, the string shown in the output above represents possible output for a text string, it is not the real encrypting of the given text string.

exists

```
exists(path-string)
```

Parameter	Description
path-string	The string path for the content whose existence to check. This is the full path to a Content Library file (such as html, text, or an image).

This method checks whether the content at the specified path exists in the Content Library.

The method returns Boolean `true` if the file at the path-string exists.

The path parameter must be an absolute path, for example:

```
cms://contentlibrary/foo.htm
```

If the path starts with the `http://` or `https://` prefix, the method assumes that the content exists and returns `true`. Any other value will result in error.

NOTE: You must always use the forward slash (/) to separate path components, do not use the backslash (\).

Example:

```
<#if exists("cms://contentlibrary/campaigns/promotion-2013-06-
27/promotion.htm")>
   The promotion file exists
<#else>
   The promotion file does not exist
<#/if>
```

The above example outputs the appropriate string whether or not the file exists at the specified path.

Example:

```
<#if exists("http://www.example.com/couponoftheday/assign"))>
   The coupon link exists
<#else>
   The coupon link does not exist
<#/if>
```

The above example outputs the string “The coupon link exists”.

facebookjoinus

```
facebookjoinus(link-name, link-url)
```

Parameter	Description
link-name	The name of the link to be used for Facebook link tracking. This is the name in the Link table. The link name must be an expression without commas. A link name and target URL (LINK_URL) are required.
link-url	The url of the link to be used as a Facebook destination, usually the URL of a page. This is the URL in the Link table. A link name and target URL (LINK_URL) are required.

Creates a URL that points to a traceable link to a Facebook page.

Example:

```
<a href="${facebookjoinus( 'FollowUsFacebook',
'https://www.facebook.com/cocacola')}"><img
src="/interact/ui/styles/images/findusonfacebook.PNG"></a>
```

produces this output, with a link that tracks and opens the Coca Cola page in Facebook:

Standard Images

The standard Facebook image is available in Interact via the image SRC path shown in the following table. When you use this path, Interact automatically updates the SRC path to the proper Akamai URL for the given Interact account.

Note that you must type these paths exactly as they appear in the table below (uppercase file extension).

Image Path	Image
/interact/ui/styles/images/findusinfacebook.png Example: `<img src="/interact/ui/styles/images/findusonfacebook.PNG">`	

facebooklike

```
facebooklike(button-type, link-name, button-verb, button-style,
description, thumbnail)
```

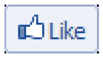
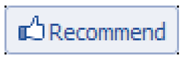
Parameter	Description
button-type	The type of like button: 0 = Like an email 1 = Like an offer
link-name	The name of the link for tracking. If buttonType = 0, this parameter preference is ignored. This is the name of the link as configured in the Link table. The link name must be a string expression without

Parameter	Description
	commas. A link-name and target URL (LINK_URL) for the offer, in the link table, is required when liking an offer.
button-verb	The verb shown on the Like button: 0 = Like 1 = Recommend The verb also appears in the individual's Facebook news feed, for example: Richard likes GiftCo's Sale. John recommends GiftCo's Sale.
button-style	The format of the Like button on the Like Landing Page: 0 = Standard On one line, the Like button followed by a text string, "X likes." 1 = Button Count A horizontal presentation of the button and number of Likes for the item. 2 = Box Count Stacked presentation of the button and number of Likes for the item.
description	The string shown on the Like Landing Page. This should be a tag line string of "what" is liked. If buttonType = 0, this parameter is ignored Dollar sign symbols must be escaped.
thumbnail	The image used for the Like Landing Page. Recommended size is 200x200 pixels.

Standard Facebook Images

The standard Facebook Like button and Facebook Recommend button images are available in Interact via the image SRC paths shown in the following table. When you use these paths, Interact automatically updates the SRC path to the proper Akamai URL for your Interact account.

Note that you must type these paths exactly as they appear in the table below (all lowercase).

Image Path	Image
/interact/ui/styles/images/likeonfacebook.png Example: `<img src="/interact/ui/styles/images/likeonfacebook.png">`	
/interact/ui/styles/images/recommendonfacebook.png Example: `<img src="/interact/ui/styles/images/recommendonfacebook.png">`	

facebookshare

```
facebookshare(title,  summary, image-url)
```

Parameter	Description
title	A string expression or constant that defines the title that will appear in the Facebook share dialog.
summary	A string expression or constant that defines the description for the link that is being shared in the Facebook dialog,
image-url	The URL to an image that will appear in the Facebook dialog. If you use an image from the Content library, this image will be published and a link to the externally published image will be used in its place. The recommended image is the logo of your organization.

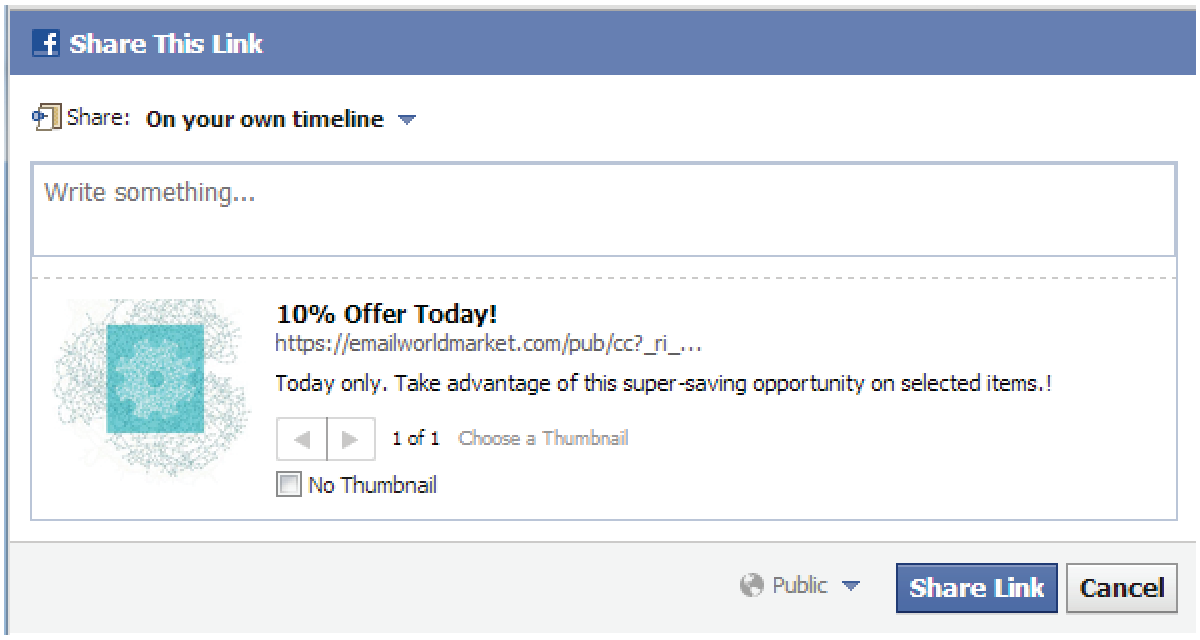
Creates a button that will give the user the ability to share a story on their Facebook page about the message or part of a message they received. For example, you can share stories about your organization, or stories about the promotion, or a coupon in the current message.

In addition to sharing the story, you can track two additional actions: when someone clicks on the share icon in the message, and when someone clicks the shared link on Facebook. To do this, you need to create two tracking entries in the link table. The link table entries do not need to be added manually. Upon launch, the system automatically adds and maintains the entries in the link table. The Link URLs are computed for both of the tracking actions and might not be URLs that you can easily recognize.

Example:

```
<a href="${facebookshare( '10% Offer Today', 'Today only. Take
advantage of this super-saving opportunity on selected items.'
,'/contentlibrary/media/logo.png')}">Share in Facebook</a>
```

produces this output:



10% Offer Today!

https://emailworldmarket.com/pub/cc?_ri_...

Today only. Take advantage of this super-saving opportunity on selected items.!

Standard image

The standard image for sharing is available in Interact via the image SRC path shown in the following table. When you use this path, Interact automatically updates the SRC path to the proper Akamai URL for your Interact account.

Note that you must type these paths exactly as they appear in the table below (uppercase file extension).

Image Path	Image
/interact/ui/styles/images/sharetofacebook.png Example: `<img src="/interact/ui/styles/images/sharetofacebook.PNG">`	

firstname

```
firstname(string-expr)
```

Parameter	Description
string-expr	The expression that contains the individual's full name. Valid formats are: First Last Last, First First M. Last Last, First M.

This convenience function obtains the first name on a small set of formats. If additional formats are required, we recommend creating a new function in a library. For more information , see `function`, `return` and `import` functions.

form

```
form(form-name)
or
form(form-name, options-hash)
or
form(form-name, parameter1, parameter2, parameter3, …)
or
form(form-name, options-hash, parameter1, parameter2, parameter3, …)
```

Paragraph	Description
form-name	An expression that provides the name to be used as a form. The form must exist either as a campaign or a form, since campaigns can be used as forms as it is the case in the "view in browser" use cases. When the `form` method is used in a link URL in a link table, the form name must be a string constant.
options-hash	An optional hash with the following options: **'usedb':boolean-expression** Specifies whether the form function is requesting that the values are retrieved from the database. The default is `false`. Setting this value to `true` will read the record from the database, slowing performance. Example: {'usedb':true} When `usedb` is set to true, the use of parameters will cause an error since all data comes from the database. **'format':string-expression** An expression or a constant value that returns one of the following values: **'H'** For a link to the HTML version of the desired form, or **'T'** For a link to the text-only version of the form. The default is to use HTML document if one exists for the form, text otherwise. When specifying the desired format, the forms processor checks whether the requested format is available for that form. If the format is not available, the forms processor return text in the available format. Example: {'format':'H'} Specific usage examples are provided in examples below. The hash is required only when you need options other than the default one, or when you are passing parameters. You can also use an empty hash (or {})].
parameter1, parameter2, parameter3, …	The parameters to append to the form for further value resolution. When `usedb` is false (the default), these are the values that will be used in the target form. When `usedb` is true, using any parameters will cause an error.

Paragraph	Description
	The values can be in one of the following forms: **datasourcealias.columnalias** Reads the given field from the current record and sends its value to the target form. The form will receive the field name of the record. fieldname=value Sends the given value to the receiving form as specified. The form will receive the field name of the record.

Provides a way to create links to other forms in the system. Forms are defined with a name, and in general both forms and campaigns can be used in this way. For more information, see *"Working with Forms and Link Tracking"*.

The `form` method invocation is usually linked to an HTML anchor tag ("<a href="" ... >".) It is flexible and allows for different ways to pass and retrieve information in the linked form as explained in the following scenarios.

In Mobile SMS campaigns, the `form` method returns a temporary shortened version of the link. The shortened URL is valid fo a short period of time. In email campaigns, this method returns a long URL.

Loading a form with data from the database

The following example shows how to pass overriding values as parameters. The example creates a link to a form called *usersettings,* requesting that the result be a form in default format. It will read the current user record:

```
<a href="${form('usersettings', {'usedb':true},)}">Unsubscribe</a>
```

Loading a form with the specified parameters

The following example links to a landing page:

```
<a href="${form('offer', {}, 'EMAIL_ADDRESS_=' + profile.email),
'MAX_AGE=55'}">Offers</a>
```

Assuming that `profile.email` is provided as an alias to EMAIL_ADRESS_ , this form link provides an email parameter in the receiving form with the email address of the current recipient.

It is worth noting that the value is obtained from the alias, while the email address is fully specified. These values are further concatenated for a string of the form:

```
EMAIL_ADDRESS_=john@example.com
```

The previous example can be simplified as follows:

```
<a href="${form('offer', {}, 'profile.email',
'MAX_AGE=55'}>Offers</a>
```

View form in a predefined format in a browser

The following example displays the current campaign in text format:

```
<a href="${form(campaign.name, {'format':'T'})}">View as text</a>
```

The example requests that the text version of the form be rendered. To request the HTML version, specify H for the `format` parameter. If you do not specify the format, the form will be rendered in the same format as the template being personalized.

IMPORTANT: When you specify an unavailable format, the available one is returned.

---

houradd

```
houradd(date-expr, hours-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
hours-expr	The number of hours to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of hours specified by `hours-expr` to a base date specified by `date-expr`. To subtract hours, specify a negative number in `days-expr`.

For example, adding 1.5 hours is equivalent to adding one hour and thirty minutes.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`,`?date`, and `?time` to specify the date type.

Example 1: Advance one hour forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${houradd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-27 14:25:03
2012-12-27 12:25:03
```

Example 2: Advance one hour on a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-27 00:00:00
```

Example 3: Advance a time forward:

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 0.5)?string("HH:mm:ss")}
```

produces this output:

```
13:55:03
```

load

```
load(url-string)
or
load(url-string, encoding)
```

Parameter	Description
url-string	The string path for the content being provided. Must be a full path to either a Content Library text file (such as html, text only, or xml) or a path to an external http or https resource.
encoding	The encoding that was used to save the file. This value must match the encoding of the file. Using an incorrect value can result on either an error (for invalid encodings) or incorrect output values when using international characters.

Loads the string of a text file, which can be used to assign to a variable or for other similar purposes. Refer to the *XML Processing Guide* for examples on how to use this method to load XML content into a variable.

NOTE: To work with remote sites, you must have an account enabled for http, https download.

IMPORTANT: This method does not process the content of the file being read as it makes no assumption of the type of contents being read. For example, it will not try to publish images, or encode clicks if the file being loaded was in html format. Loading html content with additional <head>, <title>, and/or <body> tags might cause undesirable results if the content of the file is outputted verbatim to the resulting stream. This behavior allows you to load any text into the system.

Example: Output the contents of the html version of the promotion:

```
<#assign promotion=load("cms://contentlibrary/campaigns/promotion-
2013-06-27/promotion.htm")>
${promotion}
```

Example: Output the coupon code, assuming that the loaded xml contains a sub-element called `<number>`:

```
<#assign
coupon=parsexml(load("http://www.example.com/couponoftheday/assign")
)>
${coupon.number}
```

IMPORTANT: A request to a URL, as in the second example above, might fail because of communications errors. For instance, RPL might fail to locate the remote site through no fault of its own, the remote site might throw an error under load, or the URL might return the proper content, but RPL fails to receive it because of a communication failure over the wire. Such errors are common, therefore when using load with remote URLs, it is imperative to use `<#attempt>` and `<#recover>` to account for them. With this mechanism, the RPL script might be able to assign a default content, skip, or take other action. Failure to use `<#attempt>`, `<#recover>` night cause the launch to fail; that is, one communication error that is not recovered will stop a launch of all records.

localtoutc

```
localtoutc(datetime)
or
localtoutc(date)
```

Parameter	Description
datetime	A value that contains both date and time. The date and time are considered local as per the `time_zone` setting on the template. For more information, see the `setting` directive.
date	A value that identifies the local date. The date is interpreted as midnight on the given date.

Converts the `datetime` or `date` parameter from local time to UTC datetime, based on the local time configured in the template.

Passing time-only variables will cause an error.

TIP: See `utlocal` for the inverse operation.

Example with date:

```
<#setting time_zone="America/Los_Angeles">
<#assign local="2016-09-24"?date("yyyy-MM-dd")>
<#assign utc=localtoutc(local)>
${local?string("yyyy-MM—dd HH:mm:ss")}/${utc?string("yyyy-MM-dd
HH:mm:ss")}
```

produces this output:

```
2016-09-24 00:00:00/2016-09-24 07:00:00
```

Example with date-time:

```
<#setting time_zone="America/Los_Angeles">
<#assign local="2016-9-24 13:00:00"?datetime("yyyy-MM-dd HH:mm:ss")>
<#assign utc=localtoutc(local)>
${local?string("yyyy-MM-dd HH:mm:ss")}/${utc?string("yyyy-MM-dd
HH:mm:ss")}
```

produces this output:

```
2016-09-24 13:00:00/2016-09-24 20:00:00
```

max

```
max(number1, number2, number3, …)
or
max(numeric-list-expr)
```

Parameter	Description
number1, number2, number3, etc.	The numbers from which the maximum is computed.
numeric-list-expr	A sequence expression containing the numbers to be compared.

Determines which number is the greatest of the given numbers.

Example:

```
<#assign list=[1,73,22]>
${max(list)}
${max(1,73,22)}
```

produces this output:

```
73
73
```

NOTE: If you do not provide any numbers, or if the sequence is empty, this method returns an error, as the maximum number is undefined. To catch this error, use `<#attempt>` and `<#recover>`.

---

messagedigest

```
messagedigest(string-expr)
or
messagedigest(string-expr, algorithm)
```

Parameter	Description
string-expr	The string to which the algorithm will be applied.
algorithm	Either “MD5”, “SHA”, or “SHA-256”, “SHA-384”, “SHA-512”. The default is “SHA”.

Generates a one-way digest by using two standard hashing algorithms. The hashing algorithm converts a string to a unique signature that identifies the message.

You can use this method to deliver encrypted information anywhere in a campaign or form. For example, you might use it to pass encrypted promotion codes in the query string of the link URL so that the destination web site can compare the code to a list of authorized codes.

min

```
min(number1, number2, number3, …)
or
min(numeric-list-expr)
```

Parameter	Description
number1, number2, number3, etc.	The numbers from which the minimum is computed.
numeric-list	A sequence expression with the numbers to be compared

Determines which of the given numbers is the smallest.

Example:

```
<#assign list=[1,73,22]>
${min(list)}
${min(1,73,22)}
```

produces this output:

```
1
1
```

NOTE: If you do not provide any numbers, or if the sequence is empty, this method returns an error, as the minimum number is undefined. To catch this error, use `<#attempt>` and `<#recover>`.

minuteadd

```
minuteadd(date-expr, minutes-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
minutes-expr	The number of minutes to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of minutes specified by `minutes-expr` to a base date specified by `date-expr`. To subtract minutes, specify a negative number in `minutes-expr.`

For example, adding 1.5 minutes is equivalent to adding one minute and thirty seconds.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`, `?date,` and `?time` to specify the date type.

Example 1: Advance one minute forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${minuteadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${minuteadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-27 13:26:03
2012-12-27 13:24:03
```

Example 2: Advance one second on a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output (losing the seconds):

```
2012-12-27 00:00:00
```

Example 3: Advance a time forward:

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("HH:mm:ss")}
```

produces this output:

```
13:26:03
```

monthadd

```
monthadd(date-expr, months-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
months-expr	The number of months to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of months specified by `months-expr` to a base date specified by `date-expr`. To subtract months, specify a negative number in `months-expr`.

You cannot use fractional numbers. Fractional numbers are rounded down. For example, 1.5 becomes one, and -1.5 becomes -1. This means that specifying 1.5 months adds one month.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`, `?date`, and `?time` to specify the date type.

Example 1: Advance one month forward and back

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${monthadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${monthadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2013-01-27 13:25:03
2012-11-27 13:25:03
```

Example 2: Advance one week on a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${monthadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output (losing the time portion):

```
2013-12-27 00:00:00
```

nonce

```
nonce()
or
nonce(size-number)
```

Parameter	Description
size-number	A value that determines the string size of the resulting nonce. Must be between 1 and 128. The default size

Parameter	Description
	if the parameter is omitted is 20. Numbers beyond the allowable range result in an error.

A random string of alphanumeric characters. Nonces are usually used in communication protocols such as Oauth 1, to uniquely identify a request.

Requests with nonces are retryable, if the receiving service is keeping track of the nonces, and thus can reject duplicates.

Example 1:

```
${nonce()}
```

produces output of of alphanumeric values. As described above, the output changes with every invocation:

```
0aPojhKmkk2VlmNTUjfz
```

Example 2:

```
${nonce(10)}
```

produces output of of alphanumeric values. As described above, the output changes with every invocation:

```
fJk4Ok7Yyz
```

nonemptyfields

```
nonemptyfields(field1, field2, field3, etc.)
or
nonemptyfields(sequence-expr)
```

Parameter	Description
field1, field2, field3, etc.	The names of the fields to be examined.
sequence-expr	A sequence or an expression that produces a sequence. This is useful, for instance, for use with the randomsubset method.

In some cases, the personalization record is set up so that some fields might be empty. This is done so that only the fields that actually contain values are used.

This method allows you to create dynamic content by specifying a complete set of potential values, but retrieving only the values appropriate for the current record.

To extract the field names, use the hash built-in `?keys`. To extract only the values, use the hash built-in `?values`.

This method returns a hash with the proper field names and values associated with it.

In the following example, the personalization record for Mary has values in the fields *Boots* and *Backpacks,* with the values *2 pair* and *1*, respectively:

```
<#assign populated=nonemptyfields(“Hats”, “Shirts”, “Shorts”,
“Boots”, “Backpacks”, “Tents”)>
<#list populated?keys as fieldName>
    - ${fieldName} - ${populated[fieldName]}
</#list>
```

produces this output:

```
Boots - 2 pair
Backpacks – 1
```

As shown in the example, when you want to know the field names use the hash built-in `?keys`. To extract only the values, use the hash built-in `?values`.

parsejson

```
parsejson(string-exp)
```

Parameter	Description
string-expr	The text that contains JSON.

Converts a JSON string into sequences, hashes, and primitives.

Example 1:

```
${parsejson('[1,"hello",true]')?join(",")}
```

produces this output:

```
1,hello,true
```

Example 2:

```
<#assign o=parsejson("{'a':1, 'b':'Hello','c':[2,'world',false]}")>

${o['a']}-${o['b']}-${o['c']?join(",")}
```

produces this output:

```
1-Hello-2,world,false
```

NOTE: Parsing a string null `${parsejson("null")}` prodces an error.

parsexml

```
parsexml(string-expr)
```

Parameter	Description
string-expr	The text that contains XML.

Converts a text string into a set of nodes.

The following example assumes that `xmlField` contains an XML string:

```
<#assign doc=parsexml(xmlField)>
```

rand

```
rand(value)
```

Returns a random numeric value between 0 and the given number.

randomsubset

```
randomsubset(list-expr, on-empty-expr, max-size-expr)
```

Parameter	Description
list-expr	The list from which to get the random subset.
on-empty-expr	The default value to return if the list has fewer elements than the maximum size specified.
max-size-expr	Is the maximum number of elements to return. If the list contains a number of elements that is less than these numbers, the on-empty-expr will be used.

Returns a subset of list elements. The `max-size-expr` parameter specifies the maximum number of elements to return.

Example:

```
<#assign fruits=["bananas", "oranges", "apples", "strawberries",
"pears">
<#list randomsubset(fruits, [], 3) as fruit>${fruit}</#list>
<#assign morefruits=["bananas", "oranges" >
<#list randomsubset(morefruits, fruits[0..2], 3) as
fruit>${fruit}</#list>
<#list randomsubset(fruits, fruits, 6) as fruit>${fruit}<#/list>
```

produces this output:

```
oranges strawberries pears
bananas oranges apples
bananas oranges apples strawberries pears
```

secondadd

```
secondadd(date-expr, seconds-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
seconds-expr	The number of seconds to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of seconds specified by `seconds-expr` to a base date specified by `date-expr`. To subtract seconds, specify a negative number in `seconds-expr`.

Example 1: Advance one second forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${secondadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${secondadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2012-12-27 13:25:04
2012-12-27 13:25:02
```

Example 2: Advance one second on a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output (losing the seconds):

```
2012-12-27 00:00:00
```

Example 3: Advance a time forward:

```
<#assign date="2012-12-27 13:25:03"?time("yyyy-MM-dd HH:mm:ss")>
${houradd(date, 1)?string("HH:mm:ss")}
```

produces this output:

```
13:25:04
```

shorturl

```
shorturl(link-url)
```

Parameter	Description
link-url	The URL to be shortened.

Attempts to shorten a given URL. A short URL is a smaller temporary representation of the provided URL. These short URLs are guaranteed to be valid for a short time frame after which they expire.

Currently only SMS campaigns return a shortened URL that allows you to keep the resulting message within the size limits of the SMS protocol. For email campaigns, this method does not shorten the URL and returns the same URL as provided.

Example:

```
${shorturl('http://www.google.com?q=sports+equipment')}
```

tracking

```
tracking(campaign-name)
or
tracking(campaign-name, parameter1, parameter2, parameter3, …)
```

Parameter	Description
campaign-name	An expression that identifies the campaign to be tracked.

Parameter	Description
parameter1, parameter2, parameter3,…	Additional values to use in the tracking of the campaign: • “datasourcealias.columnalias”—used for replacing fields from the recipient record using aliased fields • “FIELD_NAME” —used for replacing fields coming from the recipient record using the field name directly • “replacementname=value” —used for replacing values coming from an expression For descriptions of these values, see the method description below.

Returns a string with parameters representing the campaign, recipient, additional parameters, and other information needed for communication with Oracle Responsys from any system.

The string is formatted as:

```
_ri_=ENCODEDDATA&_ei_=ENCODEDDATA
```

The encoded data is kept in a form required for Oracle Responsys communications. This information is particularly useful when creating links from external subscription forms to the Oracle Responsys unsubscribe servlet endpoint. For more information, see “Using tracking parameters”.

To communicate back to Oracle Responsys, both parameters must be retruned from an external system in one of the ways described below.

Creating a tracking string without additional parameters

The following example creates a set of parameters for the current campaign. This is useful for external unsubscribe form usage. This link will allow an external form to be called with unsubscription information that can be used to track unsubscriptions in Oracle Responsys.

```
<a
href="http://www.example.com/unsubscribe?${tracking(campaign.name)}"
>Unsubscribe</a>
```

Creating a tracking string with additional parameters

Assuming that `profile.email` is provided as an alias to EMAIL_ADRESS_, the following tracking string provides an email parameter plus a parameter called MAX_AGE.

```
<a href="http://www.example.com/external?${tracking(campaign.name,
'EMAIL_ADDRESS_=' + profile.email), 'MAX_AGE=55'}">Weekly Add</a>
```

twitterjoinus

```
twitterjoinus(link-name, link-url)
```

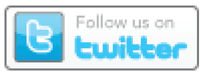
Parameter	Description
link-name	The name of the link to be used for Twitter link tracking. This is the name in the Link table. The link name must be an expression without commas. A link name and target URL (LINK_URL) are required.
link-url	The url of the link to be used as a Twitter destination. Usually the URL of a Twitter username, or a link to a search for a hash. This is the URL in the Link table. A link name and target URL (LINK_URL) are required.

Creates a URL that points to a traceable link to a Twitter user or hash.

Example:

```
<a href="${twitterjoinus( 'FollowUsTwitter',
'https://twitter.com/search?q=%23electricyty&src=hash')}"><img
src="/interact/ui/styles/images/findusontwitter.png"></a>
```

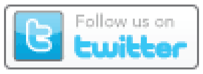
produces this output, with a link that will track and open the search for the indicated #electricity hash:



Standard Images

The standard Follow on Twitter image is available in Interact via the image SRC path shown in the following table. When you use this path, Interact automatically updates the SRC path to the proper Akamai URL for the given Interact account.

Note that you must type this path exactly as it appears in the table below (uppercase file extension).

Image Path	Image
/interact/ui/styles/images/findusintwitter.png Example: `<img src="/interact/ui/styles/images/findusontwitter.PNG">`	

twitterpost

```
twitterpost(text)
```

Parameter	Description
text	A string expression or constant that defines the proposed text of the tweet that will appear in the Twitter post dialog. It should not be longer than 106 characters, since a URL is added to the message. The added URL will not exceed 34 characters. This means that the text and the link will not exceed the Twitter maximum of 140 characters.

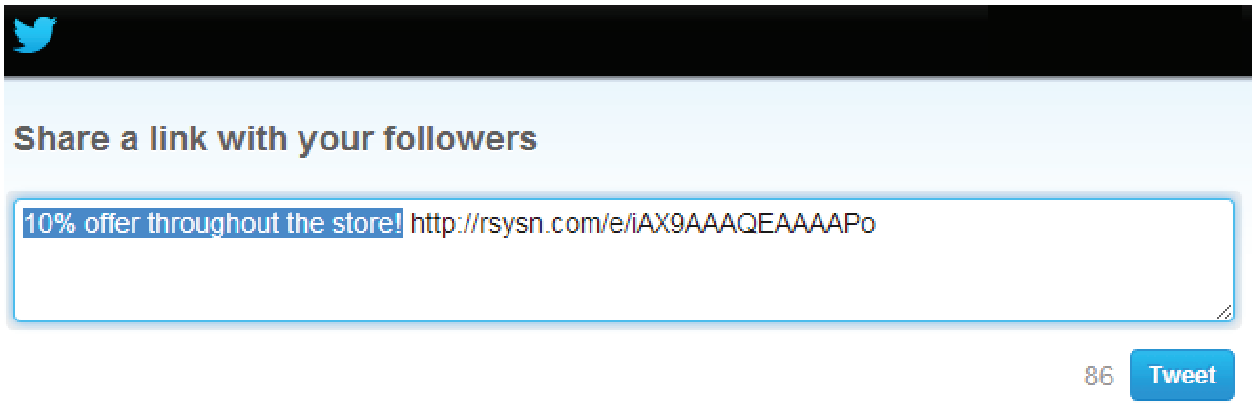
Creates a button that will enable the user to post a tweet in their Twitter feed about the message or part of a message they received. For example, you can tweet about your current campaign or a specific event.

In addition to posting the story, you can track two additional actions: when someone clicks on the post button, and when someone clicks the shared link in Twitter. To do this, you need to create two tracking entries in the link table. The link table entries do not need to be added manually. Upon launch, the system automatically adds and maintains the entries in the link table. The Link URLs are computed for both of the tracking actions and might not be URLs that you can easily recognize.

Example:

```
<a href="${twitterpost( '10% offer throughout the store!')}">Post to
Twitter</a>
```

produces this output:



10% offer throughout the store! http://rsysn.com/e/iAX9AAAQEAAAAPo

Standard image

The standard image for posting to Twitter is available in Interact via the image SRC path shown in the following table. When you use this path, Interact automatically updates the SRC path to the proper Akamai URL for the given Interact account.

Note that you must type this path exactly as it appears in the table below (uppercase file extension).

Image Path	Image
/interact/ui/styles/images/sharetotwitter.png Example: `<img src="/interact/ui/styles/images/sharetotwitter.PNG">`	

utctolocal

```
utctolocal(datetime)
or
utctolocal(date)
```

Parameter	Description
datetime	A value that contains both date and time. The date and time is considered as being in UTC.
date	A value that identifies a UTC date. The date is interpreted as midnight on the given date.

Converts the datetime or date parameter from UTC time to the local datetime configured in the template.

Passing time-only variables will cause an error.

TIP: See `localtoutc` for the inverse operation.

Example with a date:

```
<#setting time_zone="America/Los_Angeles">
<#assign utc="2016-09-24"?date("yyyy-MM-dd")>
<#assign local=utctolocal(utc)>
```

```
${local?string("yyyy-MM-dd HH:mm:ss")}/${utc?string("yyyy-MM-dd
HH:mm:ss")}
```

produces this output:

```
2016-09-23 17:00:00/2016-09-24 00:00:00
```

Example with date-time:

```
<#setting time_zone="America/Los_Angeles">
<#assign utc="2016-09-24 13:00:00"?datetime("yyyy-MM-dd HH:mm:ss")>
<#assign local=utctolocal(utc)>
${local?string("yyyy-MM-dd HH:mm:ss")}/${utc?string("yyyy-MM-dd
HH:mm:ss")}
```

produces this output:

```
2016-09-24 06:00:00/2016-09-24 13:00:00
```

weekadd

```
weekadd(date-expr, weeks-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
weeks-expr	The number of weeks to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of weeks specified by `weeks-expr` to a base date specified by `date-expr`. To subtract weeks, specify a negative number in `weeks-expr`.

You cannot use fractional numbers. Fractional numbers are rounded down. For example, 1.5 becomes one, and -1.5 becomes -1. This means that specifying 1.5 weeks adds one month.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`, `?date`, and `?time` to specify the date type.

Example 1: Advance one week forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${weekadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${weekadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2013-01-03 13:25:03
2012-12-20 13:25:03
```

Example 2: Advance one week on a date without time:

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${weekadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output (losing the time portion):

```
2013-01-03 00:00:00
```

xslt

```
xslt(source-node, transform-node)
```

Parameter	Description
source-node	A node that identifies the xml to be transformed. The node must be the root of the xml document, otherwise an error occurs.
transform-node	A node that identifies the xslt transformation to be applied to the source node.

Allows the advanced transformation of XML by using the standard XSL1.0 language.

XSL is an industry standard transformation language used to convert XML into one of the following:

- Plain text
- XML
- HTML

Use XSL only if you are familiar with its concepts.

The result of this method is a string scalar. If the transformation output specified `xml` as the output, and you need further access by using nodes, you might need to apply `parsexml` again on this result.

Example:

```
<#assign source=
  parsexml('<books><book name="Hamlet"/><book
name="Ulyses"/></books>')>
<#assign transform=parsexml(
'<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">' +
  '<xsl:output method="text"/>' +

  '<xsl:template match="/books/book">' +
  '[<xsl:value-of select="@name"/>]' +
  '</xsl:template>' +
'</xsl:stylesheet>')>
<#assign result=xslt(source, transform)>
${result}
```

produces this output:

```
[War and Peace][Ulyses]
```

The above example hardcodes the XML and the transformation directly as RPL strings, which is not best practice. As best practice, store these externally and load them from either the Content Library or remotely from an http/https source. The following example illustrates this practice. In addition, the example assumes that the transformation in sales-transform.xml produces XML and the *sales* information is processed in a loop:

```
<#assign source=parsexml(load("cms://contentlibrary/xmls/sales-info-
2013-06-26.xml"))>
```

```
<#assign
transform=parsexml(load("cms://contentlibrary/transforms/sales-
transform.xml"))>
<#assign sales=parsexml(xslt(source, transform))>
<#list sales as sale>
    ${sale.date} - ${sale.category} - ${sale.discount}
</#list>
```

yearadd

```
yearadd(date-expr, years-expr)
```

Parameter	Description
date-expr	The base date from which the offset will be calculated.
years-expr	The number of years to add or subtract. For backward offsets, use a negative number.

Adds or subtracts the number of years specified by `years-expr` to a base date specified by `date-expr`. To subtract years, specify a negative number in `years-expr`.

You cannot use fractional numbers. Fractional numbers are rounded down. For example, 1.5 becomes one, and -1.5 becomes -1. This means that specifying 1.5 years adds one year.

NOTE: Due technical limitations, RPL cannot always determine the type of date it receives (date only, time only, or both). For this reason, you should use the string built-ins `?datetime`, `?date`, and `?time` to specify the date type.

Example 1: Advance one year forward and back:

```
<#assign date="2012-12-27 13:25:03"?datetime("yyyy-MM-dd HH:mm:ss")>
${yearadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
${yearadd(date, -1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output:

```
2013-12-27 13:25:03
2011-12-27 13:25:03
```

Example 2: Advance one year on a date without time

```
<#assign date="2012-12-27 13:25:03"?date("yyyy-MM-dd HH:mm:ss")>
${yearadd(date, 1)?string("yyyy-MM-dd HH:mm:ss")}
```

produces this output (losing the time portion)

```
2013-12-27 00:00:00
```

Chapter 8. Namespace Reference

Campaign

The campaign variables are defined in the campaign definition.

The following table shows all campaign variables.

Variable	Description
campaign.id	The unique identifier of the campaign.
campaign.name	The name of the campaign.
campaign.marketingprogram	The specified marketing program in the campaign definition.
campaign.marketingstrategy	The specified marketing strategy in the campaign definition.
campaign.externalcode	The specified external campaign code as defined in the campaign.

Datasource Hashes

Datasource namespaces, or hashes, contain the fields that are defined in the datasources user interface. The datasource appears as a top level hash, and its fields appear under that hash. Both the hash and its fields use the respective aliases for registration into the namespace.

For example, if you declared a namespace to provide email (from EMAIL_ADDRESS_) with the alias *profile*, you can specify the following interpolation:

```
${profile.email}
```

The above example obtains the EMAIL_ADDRESS_ of the individual record.

Environment

Variable	Description
environment.permissionpolicylink	A link to the URL describing the permission policy link.
environment.unsubscribelink	The URL to the unsubscribe link for the current account.
environment.poweredbyurl	The URL to the image representing the Responsys “powered-by” logo.
environment.responsyslinkurl	The link to the URL for Responsys.
environment.unsubscribemailbox	The mailbox used to unsubscribe the current recipient.
environment.implicitresponselink	The image tracking pixel for autosense-enabled and tractable messages.
environment.accountdisplayname	The name of the account.
environment.debug	You can set the debugging flag for a campaign in the Email Message Designer. This variable reflects the flag setting. To prevent debug information from being sent in standard launches, this flag will be true only for launches of type “preview” and “proof”. Use this flag in `<#if>` directives to enter information you want to see in test launches and previews.
environment.confirmbuiltin	Generates a link which, upon clicking, will mark the recipient as opted in. Use this variable with the opt-in form flow where: • The user enters their email in a form. • An email is sent with the opt-in confirmation (where this namespace entry is coded.) • Upon clicking the link, the customer is confirmed as opted in, the email permission reason is set to `FR:Confirmed`, and promotional emails can now be sent.

Launch

Launch contains the following elements:

Variable	Description
launch.id	Defines the current unique identifier of the given launch.
launch.type	Describes the launch type. The value is one of the following: standard proof preview

Message

Message describes attributes of the message campaign.

Variable	Description
message.format	Defines the format being used to personalize the current message. It can be: “H”=HTML the format “T”=text format.

NOTE: In some situations, two personalization datasources provide a field with the same name but different values. RPL uses only the first value, as per order in which the datasources were specified. Trying to access the second value using a Special Variables Reference.

RPL defines a set of special variables RPL. To access these variables, use the `.variable_name` syntax, for example, `.version`.

The following table lists all supported special variables.

Variable	Description

Variable	Description
.data_model	A hash that you can use to access the data model directly. That is, variables created with the `global` directive are not visible here.
.error	This variable is accessible in the body of the `recover` directive, where it stores the error message from which you are recovering.
.fields	This variable is a hash whose keys are the field names and whose values are the field values. For more information, see “About Fields and Field Types”.
.field_types	This variable is a hash whose keys are the field names and whose values are the readable data types of the fields. For more information, see “About Fields and Field Types”.
.globals	A hash that you can use to access the globally accessible variables (the data model variables and the variables created with `global` directive). Note that variables created with `assign` or `macro` are not global, thus they never hide the variables when you use globals.
.lang	Returns the language part of the current value of the locale setting. For example if `.locale` is *`en_US`*, then `.lang` is *`en`*.
.locale	Returns the current value of the locale setting. This is a string, for example *`en_US`*. For more information about locale strings, see the `setting` directive.
.locals	A hash that you can use to access local variables (the variables created with the `local` directive, and the parameters of macro).
.main	A hash that you can use to access the main namespace. Note that global variables such as the variables of data model are not visible through this hash.
.namespace	A hash that you can use to access the current namespace. Note that global variables such as the variables of data model are not visible through this hash.
.node	The node you are currently processing with the visitor pattern (i.e. with the `visit`, `recurse`, ...etc. directives).

Variable	Description
.now	Returns the current date-time. Usage examples: `"Page generated: ${.now}"`, `"Today is ${.now?date}"`, `"The current time is ${.now?time}"`.
.output_encoding	Returns the name of the current output charset. This special variable does not exist if the framework that encapsulates RPL does not specify the output charset for RPL.
.template_name	The name of the current template.
.today	Returns the current date as of midnight of the current day. This is sensitive to the time zone of the processing environment. Unlike `.now?date`, `.today` returns a date-time, not a string representation of the date as per formatting settings.
.url_escaping_charset	If exists, stores the name of the charset used for URL escaping. If this variable does not exist, it means that the charset to use for URL encoding has not been specified. In this case, the `url` built-in uses the charset specified by the o`utput_encoding` special variable for URL encoding.
.vars	Expression .vars.foo returns the same variable as expression foo. This is useful when you have to use square bracket syntax, since that works only for hash sub-variables and requires an artificial parent hash. For example, to read a top-level variable that has a name that would confuse RPL, you can write `.vars["A confusing name!"]`. Or, to access a top-level variable with dynamic name given with variable `varName` you can write `.vars[varName]`. Note that the hash returned by `.vars` does not support `?keys` and `?values`.

About Fields and Field Types

The underlying data model is provided by multiple data sources. One of those data sources is the recipient record. The recipient record is the set of fields and values coming from the personalization datasources, including the profile list, PETs, etc.

RPL provides a way to reach the personalization record by utilizing the `.fields` and `.field_types` special variables. You can use these to see a summary of the fields when the debug flag is on, as shown in the following example.

```
<#if environment.debug>
  <table cellspacing="0" cellpadding="0" border="1">
    <tr><th>FIELD</th><th>VALUE</th><th>TYPE</th></tr>
    <#list .fields?keys?sort as fieldname>
      <tr>
        <td>${fieldname}</td>
        <td>${.fields[fieldname]}</td>
        <td>#{.field_types[fieldname]}</td></tr>
    </#list>
  </table>
</#if>
```

Notice that `.fields` and `.field_types` are preceded by a period.

Appendix A. Reserved Names in RPL

The following table lists RPL keywords. If you use these as top-level variables, you must use the square-bracket syntax (as .vars["in"]).

Keyword	Description
true	Boolean true
false	Boolean false'
gt	Comparison operator "greater than"
gte	Comparison operator "greater than or equal to"
lt	Comparison operator "less than"
lte	Comparison operator "less than or equal to"
as	Used by several directives
in	Used by several directives
using	Used by several directives

Appendix B. Operator Precedence

The following table shows the precedence assigned to operators. The operators in this table are listed in precedence order: the higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with lower precedence. Operators on the same line have equal precedence. When binary operators (operators with two “parameters”, as + and -) of equal precedence appear next to each other, they are evaluated in left-to-right order.

Operator group	Operators
Highest precedence operators	[subvarName] [subStringRange] . ? (methodParams) expr! expr??
Unary prefix operators	+expr -expr !expr
Multiplicative	* / %
Additive	+ -
Relational	< > <= >= (and quivalents: gt, lt, etc.)
Equality	== != (and equivalents: =)
logical AND	&&
logical OR	||
numerical range	..

Note that RPL precedence rules are the same as those in C, Java, or JavaScript, with support for additional RPL operators that do not exist in those languages.

Appendix C. Identifiers

Identifiers are the names of variables as used in the namespace. These identifiers must follow a predefined format as described in this section.

Variable names

A variable name is defined as letter (letter | digit)*

This means that a variable can begin with a letter, followed by multiple letters or digits.

Letters can be any of the following: A-Z, a-z, _, $, @. Numbers must be 0-9.

Example:

```
<#assign total_price_1 = unit_price_1 * quantity_1>
```

Variables with the same name but different case are considered different variables. For example, the following variables are different:

```
<#assign variable = Variable>
```

Appendix D. Glossary

Comment

Content that will not be included in the output. ITL comments are similar to HTML comments. ITL comments begin with <#-- and end with -->. Everything between these tags will be excluded from the output.

Directive

Instructions to ITL used in templates. Directives begin with <# or <@.

Hash

A variable that acts as a container for other variables (known as sub-variables). Sub-variables of a hash are accessed by name.

Interpolation

An instruction to convert an expression to text and to insert that text into the output. An interpolation begins with ${ and ends with }. Note that interpolations do not obtain a value, they only execute an expression.

Method

Calculates something based on given parameters and returns the result.

Namespace

A set of available directives, methods, ITL built-ins, and additional structures/properties that you can use.

Scalar

A variable that stores a single value. A scalar is of a specific type: string, number, date/time, or boolean.

Sequence

A structure that stores sub-variables sequentially. Sub-variables in a sequence are accessed using a numerical index.