ooRexx Documentation 1.0

RexxHTTP

Serving web content with Rexx

ooRexx Documentation 1.0 RexxHTTP
Serving web content with Rexx
Edition 2026.07.10

Author Josep Maria Blasco

Preface

This book describes RexxHTTP, a rexxlet processor for writing web applications in ooRexx running as CGI programs under the Apache HTTP server. It opens with the basic concepts behind the web and CGI, then a short, hands-on tutorial, covers installation and the writing of rexxlets, explains the request/response model at the heart of the design, gives a complete reference to the four classes that RexxHTTP places at a rexxlet's disposal, and closes with the rationale behind the design.

This book is intended for people who plan to develop web applications in ooRexx. It assumes familiarity with the ooRexx language, but no prior experience with the web: the opening chapter introduces the HTTP and CGI protocols, and the way the Apache HTTP server invokes a rexxlet, from the ground up.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

1.1. Typographic Conventions

Typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold` is used to highlight literal strings, class names, or inline code examples. For example:

> The **`Class`** class comparison methods return **`.true`** or **`.false`**, the result of performing the comparison operation.
>
> This method is exactly equivalent to **`subWord(`*`n`*`, 1)`**.

`Mono-spaced Normal` denotes method names or source code in program listings set off as separate examples.

> This method has no effect on the action of any `hasEntry`, `hasIndex`, `items`, `remove`, or `supplier` message sent to the collection.

```
-- reverse an array
a = .Array~of("one", "two", "three", "four", "five")

-- five, four, three, two, one
aReverse = .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")
```

Proportional Italic is used for method and function variables and arguments.

> A supplier loop specifies one or two control variables, *index*, and *item*, which receive a different value on each repetition of the loop.
>
> Returns a string of length *length* with *string* centered in it and with *pad* characters added as necessary to make up length.

1.2. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed, like mandatory initialization. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

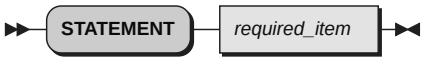
  The ▶▶— symbol indicates the beginning of a statement.

  The —··· symbol indicates that the statement syntax is continued on the next line.

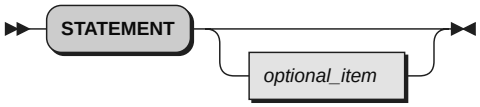
  The ···— symbol indicates that a statement is continued from the previous line.

  The —▶◀ symbol indicates the end of a statement.

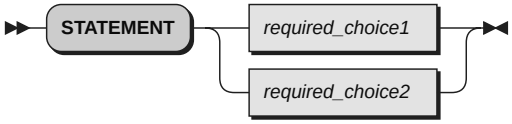
- Required items appear on the horizontal line (the main path).



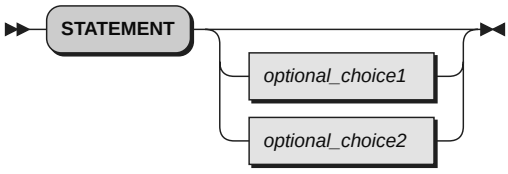
- Optional items appear below the main path.



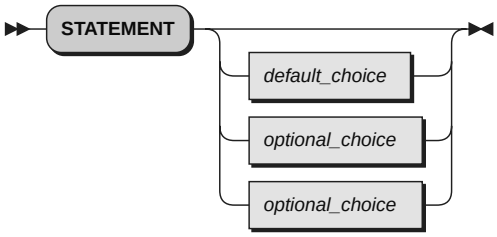
- If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.



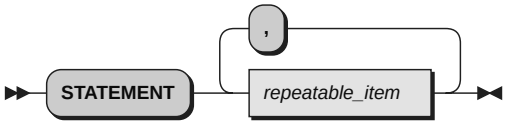
- If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it is usually the topmost item of the stack of items below the main path.



- A path returning to the left above the main line indicates an item that can be repeated.



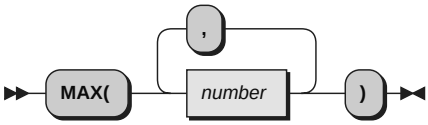
  A repeat path above a stack indicates that you can repeat the items in the stack.

- A pointed rectangle around an item indicates that the item is a fragment, a part of the syntax diagram that appears in greater detail below the main diagram.

- Keywords appear in uppercase (for example, **SIGNAL**). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, *index*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:



3. Getting Help and Submitting Feedback

The Open Object Rexx Project has a number of methods to obtain help and submit feedback for ooRexx and the extension packages that are part of ooRexx. These methods, in no particular order of preference, are listed below.

3.1. The Open Object Rexx SourceForge Site

Open Object Rexx utilizes SourceForge to house its source repositories, mailing lists and other project features at *https://sourceforge.net/projects/oorexx*. ooRexx uses the Developer and User mailing lists at *https://sourceforge.net/p/oorexx/mailman* for discussions concerning ooRexx. The ooRexx user is most likely to get timely replies from one of these mailing lists.

Here is a list of some of the most useful facilities provided by SourceForge.

The Developer Mailing List
: Subscribe to the oorexx-devel mailing list at *https://sourceforge.net/projects/oorexx/lists/oorexx-devel* to discuss ooRexx project development activities and future interpreter enhancements. You can find its archive of past messages at *https://sourceforge.net/p/oorexx/mailman/oorexx-devel*.

The Users Mailing List
: Subscribe to the oorexx-users mailing list at *https://sourceforge.net/projects/oorexx/lists/oorexx-users* to discuss how to use ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List
: Subscribe to the oorexx-announce mailing list at *https://sourceforge.net/projects/oorexx/lists/oorexx-announce* to receive announcements of significant ooRexx project events.

The Bug Mailing List
: Subscribe to the oorexx-bugs mailing list at *https://sourceforge.net/projects/oorexx/lists/oorexx-bugs* to monitor changes in the ooRexx bug tracking system.

Bug Reports
: You can view ooRexx bug reports at *https://sourceforge.net/p/oorexx/bugs*. To be able to create new bug reports, you will need to first register for a SourceForge userid at *https://sourceforge.net/user/registration*. When reporting a bug, please try to provide as much information as possible to help developers determine the cause of the issue. Sample program code that can reproduce your problem will make it easier to debug reported problems.

Documentation Feedback
: You can submit feedback for, or report errors in, the documentation at *https://sourceforge.net/p/oorexx/documentation*. Please try to provide as much information in a documentation report as possible. In addition to listing the document and section the report concerns, direct quotes of the text will help the developers locate the text in the source code for the document. (Section numbers are generated when the document is produced and are not available in the source code itself.) Suggestions as to how to reword or fix the existing text should also be included.

Request For Enhancement
: You can suggest new ooRexx features or enhancements at *https://sourceforge.net/p/oorexx/feature-requests*.

Patch Reports
: If you create an enhancement patch for ooRexx please post the patch at *https://sourceforge.net/p/oorexx/patches*. Please provide as much information in the patch report as possible so that the developers can evaluate the enhancement as quickly as possible.

  Please do not post bug fix patches here, instead you should open a bug report at *https://sourceforge.net/p/oorexx/bugs* and attach the patch to it.

The ooRexx Forums
: The ooRexx project maintains a set of forums that anyone may contribute to or monitor. They are located at *https://sourceforge.net/p/oorexx/discussion*. There are currently three forums available: Help, Developers and Open Discussion. In addition, you can monitor the forums via email.

3.2. The Rexx Language Association Mailing List

The Rexx Language Association maintains a forum at *https://groups.io/g/rexxla-members/topics*.

Basic Concepts

This chapter introduces, from the ground up, the handful of ideas the rest of the book takes for granted: what the web is, how a browser and a server talk to each other over HTTP, and how CGI lets a program build a page on demand. It assumes no prior experience with any of this. A reader already at home with the web, HTTP and CGI can skip straight to *Introduction*.

1.1. The Web Is a Web

The web is, before anything else, a *web*: an enormous collection of resources, tied to one another by links. A resource can be a page of text, but it can just as easily be a picture, a piece of music, a video, or a file of any other kind. What makes all of these into a *web*, rather than a mere pile, is that they point at each other.

A pointer from one resource to another is called a **hyperlink**, or just a link. When you are reading a page and a word or a picture takes you somewhere else when you act on it, you have followed a link. Doing this over and over — jumping from one resource to the next, and from that one to another — is what we call **browsing** the web, or navigating it. Browsing is nothing more than following links, one after another, to wherever you want to go.

1.2. Resources Need Identifiers

To be able to follow links, we need a way to identify what they point at. So every resource has an identifier, and since a link has to keep working over time and for everyone who follows it, that identifier must always designate the same resource.

Such an identifier is called a **URI**, short for *Uniform Resource Identifier*. The *Uniform* part means that every identifier follows one agreed-upon syntax — a kind of syntactic protocol — so that the same form serves to name a resource of any kind. Something like **`http://example.com/photos/sunset.jpg`** is a URI. It is the identifier of one particular resource, and any link that points at that resource carries this identifier inside it. A link, then, is really just a URI tucked into a resource, ready to be followed.

Two related initials turn up alongside URI and are worth placing here. A **URL**, a *Uniform Resource Locator*, is a URI that identifies a resource by saying where to find it — the sunset above is a URL, because it names a host and a path to reach. A **URN**, a *Uniform Resource Name*, identifies a resource by name alone, saying nothing about where it lives. Both are URIs; the everyday web address is the URL kind, and it is the only one this book has any use for. The distinction is worth recognising, not worth dwelling on.

A URI *names* a resource, but the resource itself has to reside somewhere. It resides on a **server**: a program, running on some computer on the network, that holds resources and hands them out when they are asked for. (We will say *resides on* and *is held* for now, as if each resource were simply a file sitting on that computer. This is the right picture to start with, and we will refine it before the end of this chapter.)

1.3. The Browser and the Protocol

The resources you want to look at are almost never on your own computer; they are out on servers, scattered across the network. Something has to fetch them for you and show them to you. That something is a **browser**. When you ask for a URI, the browser finds the server named in it, asks that server for the resource, receives it, and presents it to you — as a page, say, with its pictures in place.

For the browser and the server to understand each other, they have to speak a common language: a fixed set of rules for *I would like this resource, please* and *here it is*. Such a set of rules is called a

protocol, and the one the web is built on is **HTTP**, the *HyperText Transfer Protocol*. Every time you follow a link, your browser is using HTTP to ask some server for a resource, and the server is using HTTP to answer.

What comes back is often a page, and a page is usually written in **HTML**, which lays out its structure — this is a heading, this is a paragraph, this is a link. The way the page looks — its colours, its spacing, its fonts — is chosen separately, using **CSS**. And anything the page *does* on its own — reacting to you, changing as you go — is programmed in **JavaScript**. Alongside these pages travel all the other resources too: plain text files, images, sound, and the rest.

1.4. Where the Resource Comes From

The earliest servers were *pure* servers in a very literal sense. A resource was a file, sitting on the server's disk, and when a browser asked for it, the server found that file and sent it back exactly as it was stored. Ask for `/photos/sunset.jpg`, and you get the bytes of `sunset.jpg`. This is still the basic model of the web, and it is worth holding on to: at bottom, a server answers a request by handing back a resource.

But before long this ran into a wall. Some pages simply cannot be prepared in advance, because what they should say depends on something not known until the request arrives — on who is asking, or on the time of day, or on the contents of a database that changes every minute. There are far too many possible versions to store them all as files, and they will not sit still.

And even when a page *could* in principle be stored, storing it can be its own trap. Imagine a site of forty thousand pages that all share a common look: the same header, the same footer, the same colours and logo. The day the company changes its branding, every one of those forty thousand files would have to be edited to match. Done by hand, this is hopeless; done badly, the site ends up half in the old style and half in the new, which is worse than either. The header, the footer, the styling — anything shared across many pages — really wants to live in *one* place, and be applied to every page as it goes out.

Both of these push in the same direction. What is needed is a way to **produce the resource on demand**: not to fetch a file that was prepared in advance, but to *run a program* that builds the answer at the very moment it is asked for.

Several technologies grew up to do exactly this, and one of the oldest and most thoroughly proven is **CGI**.

1.5. CGI, Up Close

CGI — the *Common Gateway Interface* — is itself a **protocol**, in exactly the sense we met earlier: an agreed-upon set of rules, this time between a web server and a program. The agreement says: *when a request arrives for a certain URI, do not look for a file; instead, run this program, and let whatever the program produces be the answer.* The program is no longer a passive file waiting to be sent. It runs, fresh, on every request, and its job is to *build* the resource.

For this to work, two things have to happen. The server has to tell the program everything it needs to know about the request, and the program has to hand back a complete answer. CGI settles both, and it settles them using the plainest tools a program already has.

First, the request *in*. When the server runs the program, it places the details of the request into the program's **environment variables** — the same ordinary variables any program can read from the system it runs in. Which URI was asked for, what kind of request it was, who is asking, what browser they are using, and dozens of other facts: each one is set as an environment variable with an agreed-upon name, ready for the program to read. If the request carries a body of data with it — as it does

when someone submits a form — that body is fed to the program through its **standard input**, the very same input stream a program would read if it were reading from the keyboard.

Then, the answer *out*. Here CGI is wonderfully blunt: whatever the program writes to its **standard output** — the same place a program prints to when it prints to the screen — *becomes the response*. The server takes that output and sends it back to the browser as the answer to the request.

But the answer cannot be only the content. The browser, remember, needs to know what *kind* of thing it is receiving: is this an HTML page, a plain text file, a picture? So CGI asks the program to say so first. Before writing a single byte of the actual content, the program writes a short **header** announcing the content's type — for an HTML page, a line that amounts to *this is HTML* — then a blank line, and only then the content itself.

The smallest complete answer a CGI program can give looks like this:

```
Content-Type: text/html

Hello, world!
```

One header line, naming the type of what follows; a blank line, marking the end of the headers; and then the content. The header comes first because the browser needs to be told what is coming before it arrives.

So the whole shape of a CGI program is this. It starts up. It reads the request out of its environment variables, and out of its standard input if there is a body. It works out what the answer should be. It writes a header saying what kind of answer it is. It writes the answer. It stops. And on the next request, it all happens again, from the beginning.

CGI is **language-neutral**. Nothing in the protocol mentions any particular programming language. Environment variables, standard input, standard output, and the ability to write a few lines of text are things that *every* programming language has. So a CGI program can be written in anything at all — and that is a large part of why CGI has lasted: it asks almost nothing of the language it is written in.

It should be noted that CGI is not a very fast protocol. A brand-new program is started for every single request, and starting a program has a cost. For a site under tremendous load this cost adds up, and other technologies exist that avoid it. But for the great majority of sites — a small business, a personal site, an internal tool, anything that is not straining under millions of requests an hour — CGI is *more than enough*, and its simplicity is worth far more than the speed it gives up. It is easy to understand, easy to write, and it works the same way everywhere.

1.6. Where This Is Going

Simple as it is, CGI still leaves a fair amount on your plate. You have to reach into the environment variables and pull out the ones you care about, by their exact agreed-upon names. You have to read and make sense of the body on standard input. You have to remember to write the header before the content, in the right form, with the blank line in the right place. None of it is hard, but all of it is bookkeeping — the same bookkeeping, on every program you write — and none of it is the actual work you sat down to do.

This is exactly where RexxHTTP comes in. RexxHTTP takes all of this machinery — the environment variables, the standard input, the headers, the careful order of things — and handles it for you, so that what you write is the part that matters: the program that decides what the answer should be. The rest of this book is about writing that program.

Introduction

RexxHTTP is a rexxlet processor for ooRexx and Apache. It runs as a CGI program under the Apache HTTP server: Apache hands it an incoming request, and RexxHTTP turns the raw CGI environment into a small set of objects, runs the rexxlet against them, and writes the result back as an HTTP response. A rexxlet is an ordinary ooRexx program, so the whole of ooRexx is available to it, with the HTTP request and the response it is about to build handed in as objects rather than scraped out of environment variables by hand.

For each request RexxHTTP builds three objects. A **request** object exposes the request line, the headers and the GET/POST arguments through named methods, decoding percent- and plus-escapes automatically. A **response** object collects the status line and the response headers — cookies are among those headers — and allows them to be attached. And an **output stream** receives the body: it is a buffering subclass of the built-in `Stream` class, which allows the response headers to be written *lazily* — status, headers and cookies may still be changed after the body has already been started, right up until the response is committed. The request and the response reach the rexxlet as the `.request` and `.response` objects, placed in the environment by RexxHTTP. The body is produced with ordinary `Say` or `CharOut` instructions. The response is committed on the first flush, which normally happens when the rexxlet ends. This is the central idea of the design, and the chapter *The Request/Response Model* returns to it in detail.

The rest of this book is organized as follows:

- *A Short Tutorial* is a short, hands-on tour that builds a handful of small rexxlets, just enough to get a feel for how RexxHTTP works before going into detail.
- *Installation* explains how to install RexxHTTP and how to configure Apache to run rexxlets.
- *Writing Rexxlets* covers the same ground as the short tutorial, but more slowly and methodically, naming each piece and verifying every rexxlet against a running server.
- *The Request/Response Model* describes the model that lies at the heart of the design — the central idea introduced above.
- *Class Reference* is the complete reference to the four classes that RexxHTTP places at a rexxlet's disposal.
- *Rationale* explains why RexxHTTP is built the way it is.
- *License* reproduces the Apache License 2.0, under which RexxHTTP is released.

RexxHTTP requires ooRexx 5.3.0 or later and Apache 2.4 or later. It is released under the Apache License 2.0 and published at *https://rexx.epbcn.com/rexxhttp/* and on GitHub at *https://github.com/JosepMariaBlasco/RexxHTTP*. The first of these is the preferred one, as it takes advantage of the **Rexx Highlighter** to colour its program listings.

A Short Tutorial

This chapter conveys an idea of what RexxHTTP can do. It assumes the program is already installed, so that the reader can concentrate on understanding *what it does*; the details of installation come later, in *Installation*.

3.1. A One-Line Rexxlet

A rexxlet is nothing more than an ordinary ooRexx program. Anyone who knows ooRexx can already write one. Here is a complete, working rexxlet:

```
Say "Hello, world!"
```

One line. Request the rexxlet from a browser and **`Hello, world!`** will come back. That single line is the whole rexxlet: RexxHTTP has already given the response a sensible content type (**`text/plain`**, **`charset=utf-8`**) and everything else a well-formed reply needs, so the only thing left to supply is what to say.

3.2. Reading a Parameter

To accept parameters, the rexxlet reads them from the request object. The most direct way to send some is the *query string* — the part of a URL after the **`?`**. Add **`?Name=Josep`** to the URL and the rexxlet can read **`Name`** back:

```
If .request~arg("Name", "Omitted") Then Say "NAME parameter is missing"
Else                                    Say "Hello," .request~arg("Name")"!"
```

The `arg` method works similarly to the built-in ooRexx ARG function, but it takes a name instead of a number to identify its arguments.

3.3. Storing State

HTTP is a stateless protocol. An application that needs to keep state across requests can use cookies for that purpose: small named values the browser holds and sends back on every later request to the same place. The rexxlet reads an incoming cookie the same way it reads an argument — `cookie` on the request object, with the same options.

The following rexxlet greets the visitor by name and remembers the name for next time. It can be read as one decision: has the visitor been seen before, and is a name being supplied now?

```
nameGiven = .request~arg("Name", "Exists")      -- was Name sent at all?
newName   = .request~arg("Name")                -- its value ("" if empty)
nameKnown = .request~cookie("Name", "Exists")   -- do we have a stored name?
oldName   = .request~cookie("Name")

Select
  When nameGiven, newName \== "" Then Do        -- a name is being given
   .response~addcookie(.HTTP.Cookie~new("Name", newName))
    If nameKnown, oldName \== newName
      Then Say "Goodbye," oldName"; and hello," newName"!"
      Else Say "Hello," newName"!"
  End
  When nameGiven, nameKnown Then Do             -- Name= empty: forget me
    forget = .HTTP.Cookie~new("Name", "")
    forget~max_age = 0
```

```
    .response~addcookie(forget)
     Say "Goodbye," oldName"!"
   End
   When nameKnown Then Say "Welcome back," oldName"!"  -- no name, but we remember
   Otherwise           Say "NAME parameter is missing"
 End
```

Every branch is ordinary ooRexx; the only new things are the two cookie operations. Handing a fresh **`HTTP.Cookie`** to `addcookie` stores a value; setting that cookie's `max_age` to zero tells the browser to discard it, which is how the rexxlet forgets a name when the visitor sends an empty one.

3.4. Sending HTML

For richer, structured responses a rexxlet sends HTML. When this happens, the rexxlet sets the **`Content-Type`** header:

```
.response~content_type = "text/html"

 Say "<!DOCTYPE html>"
 Say "<html lang=""en"">"
 Say "  <head><meta charset=""utf-8""><title>Hello</title></head>"
 Say "  <body>"
 Say "    <h1>Hello, world from HTML!</h1>"
 Say "    <p>This is a paragraph.</p>"
 Say "  </body>"
 Say "</html>"
```

3.5. HTML in a Resource

To handle large blocks of HTML, it is usually practical to use a **`::RESOURCE`** directive:

```
.response~content_type = "text/html"
 Say .resources~page

::RESOURCE page
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Hello</title></head>
  <body>
    <h1>Hello, world from HTML!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
::END
```

3.6. Filling in the Template

HTML pages can display *forms* that collect data, and a rexxlet reads that data with the same `arg` method. A second resource holds a *placeholder* — a marker written here as **`%name%`** — that the rexxlet replaces with the submitted value on the way out.

```
.response~content_type = "text/html"

 -- A name came in: greet
 If .request~arg("Name", "Exists"), .request~arg("Name") \== "" Then Do
   greeting = .resources~greeting~makeString
   name     = .request~arg("Name")
   name     = .response~encodeHTML(name)
```

```
    Say greeting~caselessChangeStr("%name%", name)
    Exit
  End

  -- Nothing typed yet: ask
  Say .resources~form~makeString

::RESOURCE form
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Greeter</title></head>
  <body>
    <form method="get">
      <label>Your name: <input name="Name"></label>
      <button>Greet me</button>
    </form>
  </body>
</html>
::END

::RESOURCE greeting
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Greeter</title></head>
  <body>
    <h1>Hello, %name%!</h1>
  </body>
</html>
::END
```

Ask for the rexxlet with no parameter and the first resource goes out untouched: a form with a single field. Fill it in and submit, and the browser comes back to the same rexxlet with **`Name`** set — the form has built the query string on the visitor's behalf, the very thing *Reading a Parameter* typed out by hand.

This time **`Name`** is there, so the rexxlet reaches for the second resource and fills it in. The work is one line: `caselessChangeStr`, an ordinary ooRexx string method, replaces every **`%name%`** in the page with the value that came in. Nothing about it is particular to the web — it is the same method one would use to edit any string — and that is the point: the page is just text, the substitution is just ooRexx, and the template fills itself in with a tool that was already at hand. Reading the argument is the same `arg` from before, and it stays the same whether the form sends its data in the URL or in the body of the request; `arg` reads it the same way either way.

3.7. A Page That Remembers Its Visitor

The greeter of the previous section forgets the visitor the moment the response goes out: send it a name and it fills in the page, but come back without one and it has nothing to show but the empty form again. *Storing State* already solved remembering, with a cookie — only there the answer came back as plain text. Nothing prevents doing both: store the name in a cookie when it is given, and serve the result as a proper HTML page either way. That is the whole chapter in one rexxlet — a form to ask, a template to fill, and a cookie to remember.

```
.response~content_type = "text/html"

 Select
   When .request~arg("Name", "Exists") Then Do      -- a name is being given
     name = .request~arg("Name")
    .response~addcookie(.HTTP.Cookie~new("Name", name))
     page = .resources~greeting~makeString
     name = .response~encodeHTML(name)
     page = page~caselessChangeStr("%name%", name)
   End
```

```
    When .request~cookie("Name", "Exists") Then Do   -- we have seen you before
      page = .resources~welcome~makeString
      name = .response~encodeHTML(.request~cookie("Name"))
     page = page~caselessChangeStr("%name%", name)
    End
    Otherwise page = .resources~form~makeString      -- a stranger: ask
  End

  Say page

::RESOURCE form
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Greeter</title></head>
  <body>
    <form method="get">
      <label>Your name: <input name="Name"></label>
      <button>Remember me</button>
    </form>
  </body>
</html>
::END

::RESOURCE greeting
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Greeter</title></head>
  <body>
    <h1>Hello, %name%! I will remember you.</h1>
  </body>
</html>
::END

::RESOURCE welcome
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Greeter</title></head>
  <body>
    <h1>Welcome back, %name%!</h1>
  </body>
</html>
::END
```

Three visits tell the story. The first time, a stranger arrives with no name and no cookie, and the **`form`** resource goes out: the page asks who the visitor is. The form is filled in and submitted; now **`Name`** is set, so the rexxlet hands back a **`HTTP.Cookie`** — the same `addcookie` from *Storing State* — and fills the **`greeting`** template with the supplied name. On a later visit no name is sent, but the browser sends the cookie along with the request; the rexxlet finds it with `cookie` and fills the **`welcome`** template instead. The page knows the visitor, who typed nothing the second time.

These ideas are all here at once, and each is pulling its weight. The form collects a value without a hand-built URL; the cookie carries that value from one visit to the next; the resource holds the page as readable markup; `caselessChangeStr` drops the name into it; and **`content_type`** makes the browser render the result. None of it is more than a few lines, and together they are an ordinary web page that remembers its visitors — which is most of what a small web application ever has to do.

3.8. Reaching Out to the Network

A rexxlet is an ordinary ooRexx program which can reach out to the world — read a file, query a database, call another service over the network — and weave what it finds into its reply. The code in the following example fetches two posts from a public test API, parses the JSON each returns, and lays them out as a page.

```
 .response~content_type = "text/html"

  posts = ""
  base  = "https://jsonplaceholder.typicode.com/posts/"

  Loop id = 1 To 2
    post = FetchJSON(base || id)
    If post == .nil
      Then posts = posts || "<p>Could not load post" id".</p>"
      Else Do
        title = .response~encodeHTML(post["title"])
        body  = .response~encodeHTML(post["body"])
        posts = posts || "<article><h2>#" post["id"] "&mdash;" title "</h2>" -
                    || "<p>" body "</p></article>"
      End
  End

  page = .resources~page~makeString
  Say page~caselessChangeStr("%posts%", posts)

  Exit

FetchJSON: Procedure
  Use Strict Arg url
  -- wget on a short leash: cap each stage at 5s and do not let its
  -- default twenty silent retries hang the request if the host is down.
  Address Command "wget -qO- --timeout=5 --tries=1" url With Output Stem lines.
  If RC \= 0 Then Return .nil
  text = ""
  Loop i = 1 To lines.0
    If i > 1 Then text ||= "0a"x
    text ||= lines.i
  End
  Signal On Syntax Name BadJSON
  Return .json~fromJson(text)
BadJSON:
  Return .nil

::RESOURCE page
<!DOCTYPE html>
<html lang="en">
  <head><meta charset="utf-8"><title>Two posts</title></head>
  <body>
    <h1>Two posts, fetched and merged</h1>
      %posts%
  </body>
</html>
::END

::Requires "json.cls"
```

The loop is the whole of it. For each post, **`FetchJSON`** downloads a URL and hands back a parsed object — or **`.nil`** if anything went wrong. The pieces are joined into the **`posts`** string, dropped into the template, and sent.

The use of `encodeHTML` is mandatory to sanitize text received over the network, to avoid malformed output or XSS attacks. As to **`FetchJSON`**, it is called using a short timeout and a single try: a rexxlet that waits on the network is only as responsive as the slowest host it calls, and the leash keeps one slow answer from holding the request open.

Installation

RexxHTTP runs as a CGI program under Apache. There are two ways to install it. The package ships with an installer that does the whole job in one command — it fetches or installs Apache, deploys RexxHTTP, and leaves a complete installation you can open in a browser and explore, welcome page and worked examples included. That is the recommended route, and it is the one described first, in *The Automated Installer*. It should work on a machine where Apache is not yet set up; if it does not fit your situation — you already run Apache, or you want to understand and place each piece yourself — *Installing by Hand* is the manual recipe, from the prerequisites through the directives that turn a file into a rexxlet.

4.1. The Automated Installer

The package includes an installer that sets up a complete, working RexxHTTP installation. It installs Apache, deploys the RexxHTTP sources, plants a welcome page and a set of example rexxlets, and finishes by checking that the result actually serves a request — everything a first working install needs, ready to open in a browser and explore.

It is written for the clean-machine case. If it finds that Apache is already installed, it stops without changing anything and points to the manual recipe — placing RexxHTTP onto an Apache that already exists, or that you would rather configure yourself, is what *Installing by Hand* is for.

One thing per platform is worth having ready before you start: on **Windows**, a Command Prompt opened with *Run as administrator*; on **Linux** (Debian, Ubuntu, or WSL), the **sudo** password; on **macOS**, **Homebrew** already installed. To install RexxHTTP, go to the unpacked package and run:

```
cd install
rexx install
```

It first checks the prerequisites — that the ooRexx interpreter is version 5.2.0 or later, and that the tools it needs are present — then shows what it is about to do and asks for confirmation before changing anything. When the installation is finished, it prints an address for you to open and confirm the result works. On Linux and Windows that is **`http://localhost/`**; on macOS, where the Homebrew Apache serves on port 8080 so that it needs no administrator rights, it is **`http://localhost:8080/`**. Opening that address shows the welcome page, which is itself a rexxlet — if it appears, the processor is working — and links to the bundled examples.

A few options adjust its behaviour: **`--yes`** answers every prompt in advance, for an unattended run; on Windows, **`--root`** sets the server root (the default is **`C:\Apache24`**), **`--zip-url`** overrides the Apache download, and **`--user`** switches to a non-administrator install, described next. On any platform, **`--port`** changes the port Apache listens on, described further below. Running **`rexx install --help`** lists them.

`--user` answers a different need: installing with no administrator rights at all, the case of a student, or any machine where nobody is going to hand out that access. Under **`--user`** the default root becomes **`%USERPROFILE%\Apache24`** rather than **`C:\Apache24`** — still overridable with **`--root`** — and nothing is registered as a Windows service. Apache is started instead from a **`start.cmd`** left at the root of the server tree, which opens the server in its own window and hands the console back; closing that window stops the server, and unlike a service it does not start again on its own after a reboot — reopening **`start.cmd`** does. How to remove a **`--user`** install is covered below, alongside the rest of uninstallation.

`--port` changes the port Apache listens on. Unlike the options above it applies on every platform. Left alone, the installer uses the platform's own default — the same one behind the address it prints to

open at the end: 80 on Linux and Windows, 8080 on macOS. Passing **`--port`** with a number from 1 to 65535 puts Apache on that port instead, and the closing address is adjusted to match. This matters most when the default port is already taken: before it changes anything, the installer checks whether the port it means to use is free, and if it is not, it stops and suggests trying again with another **`--port`** rather than disturbing whatever already holds it. The uninstaller needs no such option — it restores the original port on its own, so **`--port`** is an install-time choice only.

To reverse the installation, a companion uninstaller removes what was put in place. The installer leaves it on the system rather than in the unpacked package, so it is still there to run once the download is gone. Its home differs by platform. On **Linux** and **macOS** it lives in **`/usr/local/share/rexxhttp`**; go there and run:

```
rexx uninstall
```

On **Windows** the uninstaller lives outside the Apache tree, in a per-machine location that stays put no matter where Apache itself was installed: by default **`C:\ProgramData\rexxhttp`**. There it takes the form of a launcher you run by its full path from an elevated console — by default **`C:\ProgramData\rexxhttp\rexxhttp-uninstall.cmd`**. You need not memorise it: the installer prints the exact path when it finishes. A **`--user`** install is the exception: having never had administrator rights to write there, it left no launcher in **`ProgramData`**. Reverse it instead from wherever the package was unpacked, with **`rexx uninstall --user`** (adding **`--root`** too, if that was overridden at install time).

By default the uninstaller is symmetric with the install: it removes RexxHTTP and the Apache the installer had added, returning the machine to its earlier state. To keep Apache and retire only RexxHTTP — the case where RexxHTTP was a guest on a server used for other things too — add **`--keep-apache`**. On Windows this leaves the Apache service registered and running, merely reloaded without the RexxHTTP handler; on Linux and macOS it likewise leaves the running Apache untouched beyond dropping the handler. Either way the uninstaller cleans up after itself completely, with no manual step left to the user.

4.2. Optional: the Rexx Highlighter

RexxHTTP optionally supports the **Rexx Highlighter**, which colours the Rexx program listings it serves. It is entirely optional: with no highlighter available, listings are served uncoloured and nothing else changes.

The highlighter is part of the **Rexx Parser**,[1] a separate ooRexx package. What matters is that the Parser be reachable *at install time*: the installer looks for it then — in the current directory, in `REXX_PATH`, and in `PATH` — and, if it finds it, records its location in the Apache configuration. It does not have to be on the system `PATH` permanently; it only has to be findable while RexxHTTP is being installed.

Because that location is written into the configuration once, the Parser must not be moved afterwards: RexxHTTP reads the recorded path at run time and does not search for the Parser again. Move the Parser and highlighting stops until RexxHTTP is reinstalled. If the Parser is not present at install time, no location is recorded and listings are simply served uncoloured; installing the Parser and reinstalling RexxHTTP enables highlighting later.

On some distributions the Parser is already present and reachable out of the box — the **net-oo-rexx** bundle, for instance, ships it ready to use, so highlighting works with no extra setup.

[1] *https://rexx.epbcn.com/rexx-parser/*

4.3. Installing by Hand

The rest of this chapter is the other route: setting up RexxHTTP by hand, one piece at a time. It is the route to take when the installer does not fit — when Apache is already running and the installer steps aside rather than touch it, or when the machine is not one of the three the installer knows, or simply when it is worth seeing what the installer does on your behalf. Nothing here depends on the installer having run; the two routes reach the same working server by different means. The sections that follow go in order: what must be in place first, how to install Apache on each platform, where the RexxHTTP sources go, the directives that mark which files are rexxlets, and a final check that the whole montage serves a request.

4.3.1. Prerequisites

RexxHTTP needs ooRexx 5.2.0 or later and an Apache 2.4 server with the CGI module (**`cgid`** or **`cgi`**) and the **`actions`** module enabled. Installing Apache and enabling those two modules is the subject of the next section; this one covers what has to be true before that. It also uses **`curl`** for the smoke test at the end.

Installing ooRexx itself is outside the scope of this manual; the interpreter is obtained and installed separately, from the Open Object Rexx project, at *https://sourceforge.net/projects/oorexx/*. RexxHTTP assumes a working **`rexx`** on the server and nothing more.

The two modules are needed for a specific reason: the CGI module is what runs the CGI program, and **`actions`** is what allows routing a request to a CGI program of choice rather than executing the requested file directly.

4.3.2. Installing Apache

RexxHTTP needs an Apache 2.4 server with two modules switched on: the *CGI module*, which runs CGI programs, and **`actions`**, which lets a request be routed to a CGI program of choice rather than to the file it names. The CGI module goes by one of two names depending on the build: **`cgid`** on the multi-threaded servers usual on Linux and macOS, and plain **`cgi`** on the default Windows build. The two are interchangeable as far as RexxHTTP is concerned; each platform below names the one that applies to it. This section installs Apache and enables both modules on Linux, macOS and Windows. Each platform reaches the same place — a running server ready for the RexxHTTP-specific configuration that follows — by the route that is idiomatic for it, so the commands differ even though the destination does not. A reader who already has Apache 2.4 with the CGI module and **`actions`** enabled can move on to the next section.

One difference is worth stating once, since it shapes all three. On Debian and its relatives Apache ships with the helper scripts **`a2enmod`** and **`a2ensite`**, which manage modules and sites by maintaining symlinks. The Homebrew and Apache Lounge builds have no such helpers: a module is enabled by editing a **`LoadModule`** line in the main configuration file, **`httpd.conf`**, by hand. The work is the same; only the mechanism differs.

4.3.2.1. Linux (Debian, Ubuntu, WSL)

On Debian or Ubuntu, the package manager installs Apache and the helper scripts enable the two modules:

```
sudo apt-get install -y apache2
sudo a2enmod cgid actions
sudo systemctl restart apache2
```

Apache listens on port 80 from the start and is already running as a system service. Visiting **`http://localhost/`** in a browser shows the default Apache page, confirming the server is up. On Red Hat

derivatives (Fedora, Rocky, AlmaLinux) the equivalent is **`dnf install httpd`**, with the two modules enabled by their **`LoadModule`** lines under **`/etc/httpd/conf.modules.d/`** rather than through **`a2enmod`**.

These same instructions apply unchanged inside the Windows Subsystem for Linux (WSL), running a Debian or Ubuntu distribution — a route many Windows users find easier than a native Windows install. The server is reached the same way as a native one: from a browser on the Windows side, it answers at **`http://localhost/`**, exactly as if Apache had been installed directly under Windows.

4.3.2.2. macOS (Homebrew)

The Apache that ships inside macOS is awkward to rely on — Apple has been retiring it — so the practical route is the **Homebrew** build, installed as the formula **`httpd`**:

```
brew install httpd
```

Homebrew reports where it put things, and the paths depend on the processor: on Apple Silicon the prefix is **`/opt/homebrew`**, on older Intel Macs it is **`/usr/local`**. The command **`brew --prefix`** prints the right one; the configuration file is then ***`PREFIX`*`/etc/httpd/httpd.conf`**. Open it in an editor. The two **`LoadModule`** lines for the modules RexxHTTP needs are present but commented out in the module section; remove the leading # from each:

```
LoadModule cgid_module lib/httpd/modules/mod_cgid.so
LoadModule actions_module lib/httpd/modules/mod_actions.so
```

Start the server — and restart it after any change to **`httpd.conf`** — with:

```
brew services start httpd
```

One surprise to expect: the Homebrew build listens on port **`8080`**, not **`80`**, so that it can run without administrator rights. The default page is therefore at **`http://localhost:8080/`**. To use the ordinary port 80, change the **`Listen 8080`** line in **`httpd.conf`** to **`Listen 80`** and start the service with **`sudo`**. The examples later in this chapter write **`http://localhost/`**; on an unmodified Homebrew server, add the **`:8080`** accordingly.

4.3.2.3. Windows (Apache Lounge)

On Windows, Apache is installed as a ready-to-run binary — there is nothing to compile. The build used here comes from **Apache Lounge**[2], which has supplied Windows binaries for many years and ships Apache on its own, without the extra stack RexxHTTP does not need. The download is a plain ZIP with no installer, so the steps below are done by hand, but none of them involves building anything.

The reason a third-party build is needed at all is that the Apache Software Foundation itself publishes only source code, not official Windows binaries, so a ready-to-run server comes from a distributor. Several are well established — **XAMPP** is the best known, bundling Apache with PHP and a database — and Apache Lounge is a trustworthy one; it is worth knowing only that it is a community distributor, not the Foundation itself.

Apache Lounge builds are compiled with Microsoft's Visual C++ tools and will not start without the matching *Visual C++ Redistributable* installed first; its absence shows up as a missing

[2] *https://www.apachelounge.com/download/*

VCRUNTIME140.dll when the server is launched. The redistributable is linked from the Apache Lounge download page, in a 64-bit and a 32-bit form; install the one that matches the build chosen below, before going further.

Then download the Apache ZIP — the Win64 build on a 64-bit Windows, the Win32 build on the rare 32-bit one. The archive already contains a folder named **Apache24**; place that folder at the root of a drive, so the result is **C:\Apache24** and not a doubled **C:\Apache24\Apache24**. That path is the default **ServerRoot** the shipped configuration expects; if the folder is placed elsewhere, edit the **Define SRVROOT** line near the top of **conf\httpd.conf** to match. Inside that file, paths are written with forward slashes even on Windows, because Apache treats the backslash as an escape character.

Open **conf\httpd.conf** in a text editor. The modules are enabled by finding their **LoadModule** lines, already present but commented out among the many others in the module section, and removing the leading # from each:

```
LoadModule cgi_module modules/mod_cgi.so
LoadModule actions_module modules/mod_actions.so
```

Then, lower down in the same file, find the **ServerName** line and uncomment it too, setting it to:

```
ServerName localhost
```

That last line is optional, but it spares the harmless **AH00558** warning Apache prints at startup when it cannot determine its own name.

Finally, start Apache. For trying things out, the simplest way is to run it by hand: open a Command Prompt, change to the **bin** directory, and launch the server in the foreground:

```
cd C:\Apache24\bin
httpd.exe
```

The window stays occupied while the server runs; **Ctrl**-C stops it and leaves nothing behind. The default page is meanwhile at **http://localhost/**, on the ordinary port 80.

For a permanent setup, Apache can instead be installed as a Windows service — **httpd.exe -k install** from an *Administrator* prompt, then started with **httpd.exe -k start** — but that is more than a test montage needs, and the service keeps running across reboots until removed with **httpd.exe -k uninstall**.

Many Windows users will find it easier still to skip the native Windows build altogether and install Apache inside the Windows Subsystem for Linux, following *Linux (Debian, Ubuntu, WSL)* above; the server is then reached from Windows at **http://localhost/** just the same.

4.3.3. Installing the Sources

A RexxHTTP installation is five files: the processor, **RexxHTTP.rex**, and the four classes it requires — **HTTP.Request.cls**, **HTTP.Response.cls**, **HTTP.OutputStream.cls** and **HTTP.Cookie.cls**. They are distributed together in the download, at *https://rexx.epbcn.com/rexxhttp/rexxhttp-latest.zip*, along with a sixth file used only on Unix, the wrapper **RexxHTTP_wrapper**.

Copy the six files — everything in the download's **src/** directory — to the server's **cgi-bin** directory. Where that is depends on the Apache installed in *Installing Apache*: on Debian or Ubuntu it is **/usr/lib/cgi-bin/**; on a Homebrew Apache, **cgi-bin** under the Homebrew prefix, such as **/opt/homebrew/var/www/cgi-bin/** on Apple Silicon or **/usr/local/var/www/cgi-bin/** on Intel; on

the Windows Apache Lounge build, **`C:\Apache24\cgi-bin\`**. Keeping the files together is what lets the processor find its classes and, on Unix, lets the wrapper find the processor.

On Unix — both Linux and macOS — there is one more step: make the wrapper executable.

```
chmod +x RexxHTTP_wrapper
```

The download is packaged on Windows, which does not carry a Unix execute permission, so the wrapper arrives without it whichever Unix unpacks it; Apache will not run a CGI program it cannot execute. The step is the same on Linux and macOS. It does not apply on Windows, where Apache runs **`RexxHTTP.rex`** directly and the wrapper is unused.

4.3.4. Telling Apache Which Files Are Rexxlets

Two things have to be true for RexxHTTP to serve a request, and they are the same on every platform. First, the handler must be declared: a named action, bound with **`Action`**, that routes a request to the processor instead of to the file the request names. Second, the files that are meant to be rexxlets must be marked as such, so that everything else is served as ordinary static content. Everything below is one tested way to satisfy those two requirements; it is not the only way. A reader who already runs Apache will see where to vary it, and the only point RexxHTTP genuinely imposes is that on Unix the handler must reach the processor through the wrapper, not run it directly. The rest — which directory, which URL prefix, where the directives live — is open to local taste.

The sample configuration puts the sources in the server's standard **`cgi-bin`** directory, which spares one directive: the prepackaged Apache builds already alias **`cgi-bin`** as executable CGI, so no **`ScriptAlias`** of one's own is needed. The handler, named **`RexxHTTP`**, is bound to the program Apache executes. This is the one line that differs between platforms, and it differs only in what it points at:

```
Action RexxHTTP /cgi-bin/RexxHTTP_wrapper
```

on Unix, where Apache runs the wrapper and the wrapper reaches **`RexxHTTP.rex`** beside it; and

```
Action RexxHTTP /cgi-bin/RexxHTTP.rex
```

on Windows, where Apache runs **`RexxHTTP.rex`** directly and no wrapper is involved. Either way, **`Action`** names the handler **`RexxHTTP`** and binds it to the processor. Nothing under **`cgi-bin`** is meant to be requested by the public — it is only the address the handler dispatches through.

With the handler declared, the configuration marks which files are rexxlets. The recommended model — the reasoning is in the *Rationale* chapter — is a **whitelist**: a file is a rexxlet only when it is marked so explicitly, and everything else is served straight off disk. A file is whitelisted by handing it to the handler with **`SetHandler`**, scoped to the directory the rexxlets live in.

That directory is the server's *document root* — the folder Apache serves web pages from, the one a bare **`http://localhost/`** maps to. It is *not* the **`cgi-bin`** directory where the sources were copied a moment ago: **`cgi-bin`** holds the processor, and the document root holds the rexxlets that the processor runs. The two are separate directories, and the path to the document root depends on the Apache installed in *Installing Apache*. On Linux and macOS it is typically **`/var/www/html`**; on the Windows Apache Lounge build it is **`htdocs`** under the server root. If a given Apache uses some other location, its **`DocumentRoot`** line in the configuration is what names it. The block below goes in that directory — on Linux and macOS:

```
<Directory "/var/www/html">
    Options +ExecCGI
```

```
    DirectoryIndex index
    <Files "index">
        SetHandler RexxHTTP
    </Files>
</Directory>
```

and on the Windows Apache Lounge build, where the document root is **`htdocs`** under the server root, the same block reads:

```
<Directory "${SRVROOT}/htdocs">
    Options +ExecCGI
    DirectoryIndex index
    <Files "index">
        SetHandler RexxHTTP
    </Files>
</Directory>
```

This whitelists a rexxlet named **`index`** and, with **`DirectoryIndex`**, makes it the file Apache serves when a directory is requested by its path alone. A request for that directory now runs the rexxlet; everything beside it — templates, images, stylesheets — is served as static content. The pattern rewards a tidy layout: give each area its own directory with an **`index`** rexxlet inside, and the URL structure simply mirrors the directory structure, with no filenames trailing in the path. Each additional rexxlet is one more **`<Files>`** block, and the same lines work verbatim in an **`.htaccess`** file inside the directory, where per-directory control is preferred to editing the server configuration, provided **`AllowOverride`** permits it.

Those few directives are all RexxHTTP adds. *Where* they go, and how they are activated, is the platform's business: on Debian or Ubuntu they can live in a fragment under **`conf-available`**, enabled with **`a2enconf`** and a reload; on a prepackaged Windows Apache they go in the bundled **`httpd.conf`** or an included fragment. RexxHTTP does not care which, as long as the handler and the **`<Files>`** whitelist end up in effect for the directory that holds the rexxlets.

If a rexxlet needs environment values of its own, set them the usual Apache way, with **`SetEnv`**, and read them back through the request object as *The Request/Response Model* describes.

4.3.5. Smoke Test

A minimal rexxlet confirms the montage end to end. Put this in a file named **`index`** in the whitelisted document root:

> Example 4.1. A Minimal Rexxlet
>
> ```
> .response~content_type = "text/plain"
> Say "Hola, mundo"
> ```

With **`index`** whitelisted (**`<Files "index">`**) and Apache reloaded, open a web browser and visit the server's root:

```
http://localhost/
```

On an unmodified Homebrew server on macOS, which listens on port **`8080`** rather than **`80`**, visit **`http://localhost:8080/`** instead.

The page should read **`Hola, mundo`**. That means all the moving parts line up: Apache routed the request through the handler, the processor — reached through the wrapper on Unix — built the

request and response objects, ran the rexxlet, and flushed the body back. If instead the browser shows an error page, on Unix the first place to look is the wrapper's line endings and the path it **`Call`**s; if the page shows the rexxlet's source text rather than its output, the file was served statically and is missing from the whitelist.

A second, more thorough check is to run the bundled integration suite, which stands up its own throwaway Apache montage, exercises the whole pipeline and tears it down again. It is described in the test suite's own readme and is the same suite the project verifies every release against.

Writing Rexxlets

A Short Tutorial showed what a rexxlet can build, from a one-line greeting up to a page that fetches data from the network. This chapter goes back over the same ground at a slower pace and names the parts: it builds a few small rexxlets, each introducing a little more of the API, and — unlike the short tutorial — it checks every one against a running server with **`curl`**, so the actual HTTP that goes back and forth is visible. It assumes RexxHTTP is installed and reachable, as the *Installation* chapter describes; the examples are written for a rexxlet directory mapped at **`/rexxlets`**, the layout the test suite uses, so a rexxlet in **`hello/index`** answers at **`/rexxlets/hello/`**. Adjust the path to wherever the local deployment maps its rexxlets.

A rexxlet is an ordinary ooRexx program. There is no framework class to subclass and no entry point to declare: the file *is* the rexxlet, and running it produces the response. What RexxHTTP adds is the two objects it hands the rexxlet — the request that came in, and the response being built — and the redirection that turns a plain `Say` into body output. Everything below is a consequence of those two facts.

5.1. A First Rexxlet

The smallest useful rexxlet greets the world. Put this in a file named **`index`** in a rexxlet directory — here **`hello/`**:

```
.response~content_type = "text/plain"
Say "Hello, world"
```

Request it:

```
$ curl http://127.0.0.1/rexxlets/hello/
Hello, world
```

The processor publishes the two objects a rexxlet works with in the global environment, as **`.request`** and **`.response`**. The assignment here sets the response's **`Content-Type`**; the `Say` writes the body.

That last point matters. `Say` does not print to a terminal here: while the rexxlet runs, the default output stream is the response's buffered body, so `Say` *appends a line to the response*. Nothing is sent yet — the body is collected in a buffer — and when the rexxlet ends, the processor flushes it, writing the headers and then the body to the client. A rexxlet that does nothing but `Say` still produces a complete, well-formed HTTP response; the processor handles the envelope.

The content type can go too, and so the smallest rexxlet that works is the single **`Say "Hello, world"`** of *A One-Line Rexxlet* — a fresh response already carries **`text/plain; charset=utf-8`**, and that default charset is why Unicode written straight into a rexxlet reaches the browser intact. What this chapter adds to that picture is what the processor does with the two objects: the request is built from the incoming call, the body is buffered, and `Say` appends to it.

5.2. Reading the Request

The short tutorial already read a parameter with `arg`; here is the whole of it. Request arguments — the query string of a **`GET`**, the form body of a **`POST`** — are reached through `arg`, which is modelled on the built-in ooRexx `Arg` function: called with no argument it returns the count, and otherwise the first argument selects one, by position when it is a whole number and by name otherwise. A second argument says what to return — **`Value`** (the default), **`Name`**, **`Exists`** or **`Omitted`**.

Here is a rexxlet that enumerates every argument it received, by position:

```
n = .request~arg()
Say "count="n
Do i = 1 To n
  Say i": ".request~arg(i, "Name")"="".request~arg(i, "Value")
End
```

Calling it shows the arguments back in the order they arrived, names in their original spelling:

```
$ curl "http://127.0.0.1/rexxlets/multi/?q=hello&Lang=ca"
count=2
1: q=hello
2: Lang=ca
```

Lookup by name is the more common case, and it is caseless — **`arg("lang")`** finds **`Lang`** — while **`Exists`** lets a rexxlet branch on whether an argument was sent at all:

```
Say "q=".request~arg("q", "Value")
Say "exists=".request~arg("q", "Exists")
Say "missing_exists=".request~arg("nope", "Exists")
```

```
$ curl "http://127.0.0.1/rexxlets/qs/?q=hello"
q=hello
exists=1
missing_exists=0
```

Two things are happening here that the rexxlet never has to code around. First, names and values arrive *decoded*: percent-escapes and the **+**-for-space convention of form encoding are already undone, on both sides of each pair, so an argument the form sent as **`user+name`** is reached as **`arg("user name")`**. Second, the arguments are guaranteed well-formed before the rexxlet runs. RexxHTTP applies a strict argument policy — every parameter must have the shape *name*=*value*, names must be non-empty, must not begin with a digit, and must be unique caselessly — and a request that breaks a rule is answered with **`400 Bad Request`** and never reaches the rexxlet. So `arg` never has to cope with a malformed or duplicated parameter; the rexxlet may assume every argument has a name and a value and that a name names exactly one argument. The *Rationale* chapter explains why strict-by-default is the right policy for this kind of deployment, rather than a hazard. The full repertoire of `arg` — positional versus named lookup, the four options, the treatment of duplicates — is in the **`HTTP.Request`** reference in the *Class Reference*.

The same rexxlet code serves a **`GET`** and a form **`POST`** without change: the processor parses the query string of the one and the **`application/x-www-form-urlencoded`** body of the other into the same arguments, so `arg` reads identically either way.

5.3. Setting a Cookie

A cookie the response sets is an **`HTTP.Cookie`** object: create it, set its attributes, and hand it to **`.response~addcookie`**. The cookie is scheduled then, and emitted as a **`Set-Cookie`** header when the response is committed.

```
c = .HTTP.Cookie~new("session")
c~value = "abc123"
c~path = "/"
.response~addcookie(c)
Say "cookie set"
```

```
$ curl -i "http://127.0.0.1/rexxlets/setck/"
```

```
HTTP/1.1 200 OK
...
Set-Cookie: session=abc123; Path=/
Content-Type: text/plain; charset=utf-8

cookie set
```

A freshly created cookie is the quietest thing possible — a session cookie with **`name=value`** and nothing else — and only the desired attributes are added: **`c~max_age = 86400`** for a lifetime, **`c~httponly = 1`** to keep it from JavaScript, **`c~samesite = "Lax"`** and so on. The cookie validates eagerly: an inconsistent combination, such as **`SameSite=None`** without **`Secure`**, is refused at `addcookie` rather than emitted as a header the browser would silently drop. The attribute set and the consistency rules are in the **`HTTP.Cookie`** reference.

Reading a cookie back is the request side, and it parallels `arg`:

```
Say "count=".request~cookie()
Say "session=".request~cookie("session", "Value")
```

```
$ curl --cookie "session=abc123" "http://127.0.0.1/rexxlets/readck/"
count=1
session=abc123
```

The two halves form a round trip: a value is taken verbatim on the way out (reserved characters must be encoded by the caller) and returned percent-decoded on the way in. Note one asymmetry from `arg`: cookie lookup by name is case-*sensitive*, because user agents treat **`session`** and **`Session`** as different cookies, and the option word is given as an abbreviation rather than by its first letter alone. The details are in the `cookie` entry of the **`HTTP.Request`** reference.

5.4. Headers After the Body: Lazy Headers

Because the body is buffered, the response headers stay changeable *after* output has begun — right up until the response is committed. This is unusual enough to be worth seeing directly. The following rexxlet writes a line of body, *then* sets a header, then writes more body:

```
Say "body first line"
.response~content_type = "text/plain"
.response~x_late_header = "yes"
Say "body second line"
```

The header set after the first `Say` still reaches the client:

```
$ curl -i "http://127.0.0.1/rexxlets/late/"
HTTP/1.1 200 OK
...
X-Late-Header: yes
Content-Type: text/plain; charset=utf-8

body first line
body second line
```

The body the rexxlet "wrote" went into the buffer, not onto the wire, so when **`x_late_header`** was assigned the head had not yet been sent and the new header simply joined the others. Nothing is committed until the first flush — here, the processor's automatic one after the rexxlet returns — at which point the whole head goes out in front of the whole body. (The message form **`.response~x_late_header = "yes"`** reaches the header **`X-Late-Header`**; the response object

turns underscores into hyphens. A header whose name truly contains an underscore needs the bracket form, **`response["X_Odd_Name"] =`** — see the **`HTTP.Response`** reference.)

This freedom has one edge. Once the response *is* committed the head is on the wire and can no longer change, so a header assignment after that point is a programming error and raises, rather than failing silently. A rexxlet that just writes body and lets the processor flush never meets this edge; a rexxlet that flushes early does, deliberately — which is the next topic.

5.5. Flushing Early

A rexxlet may flush the response itself, before it returns, by calling **`.response~flush`**:

```
.response~content_type = "text/plain"
Say "before flush"
.response~flush
Say "after flush"
```

```
$ curl "http://127.0.0.1/rexxlets/partial/"
before flush
after flush
```

The output is identical to a rexxlet that never flushed — and that is the honest result to show. The first flush commits the response: it writes the headers, then the body buffered so far, and from that point the head is frozen. What an explicit flush buys is therefore not what intuition (or the 2006 manual) suggests. It does *not* reliably make the client see “before flush” appear before “after flush”: in a production deployment **`mod_deflate`** sits downstream and buffers the CGI output to compress it, so progressive, client-visible streaming is not something a rexxlet can count on. What an early flush does buy is control over *when* the head is committed — fixing the headers and status at a chosen point — which is occasionally what is wanted and is the mechanism behind the lazy-header behaviour above. *The Request/Response Model* tells this story properly, **`mod_deflate`** and all; for everyday rexxlets, simply never calling `flush` and letting the processor do it is exactly right.

5.6. Errors and Redirects

Two common responses have their own one-line forms, written to be the last thing a rexxlet does. To send an error page:

```
If \recordExists(.request~arg("id")) Then
  Exit .response~error(404, "no such record")

.response~content_type = "text/html; charset=utf-8"
Say "<p>found it</p>"
```

`error(status [, detail])` sets the status, switches the content type to **`text/html`**, and replaces any buffered body with a short error page; the optional detail is HTML-escaped and shown as a diagnostic line. **`.response~404(detail)`** is shorthand for the commonest case. To redirect:

```
Exit .response~redirect("/rexxlets/hello/")
```

`redirect(location [, status])` defaults to **`302`**; pass **`301`** for a permanent move or **`303`** after a form **`POST`**. Both `error` and `redirect` follow the same commit discipline as a late header: they discard an uncommitted body and do their work, but raise if the response has already been committed, since by then the status can no longer change. Their full behaviour is in the **`HTTP.Response`** reference.

5.7. Where to Go Next

These rexxlets exercise most of the everyday API: reading arguments and cookies, writing the body, setting headers and cookies on the response, and the two shorthand exits. The next chapter, *Two Rexxlets in Production*, reads two rexxlets that put this same API to work in production — the page factory and the Markdown renderer that serve `rexx.epbcn.com`. The one idea underneath all of them — the buffered body, the lazy headers, the single commit — is the subject of *The Request/Response Model*, which is worth reading before building anything larger. The exhaustive, method-by-method account of the four classes is the *Class Reference*.

Two Rexxlets in Production

The rexxlets in the *Writing Rexxlets* chapter were written to teach one idea at a time. This chapter reads two that were written to do a job. They are not part of RexxHTTP — nothing in the processor depends on them — but they ship with it as **`apps/page.cls`** and **`md/index.rxl`**. The pair serves `rexx.epbcn.com/rexxhttp` — and, once RexxHTTP is installed, the same pair is very likely already running on the reader's own machine, since the installer plants a browsable sample site into the document root and these two rexxlets are what serve it. The first gives every page its shared frame, and the second turns the site's Markdown files into styled pages. They are worth reading precisely because they are ordinary. Each one is short, each uses only the API the tutorial already covered, and between them they show how little code a real page needs once the request and response objects are doing their share of the work.

The two are also a pair by design. The renderer produces a page's *content*; the factory wraps that content in the site's *frame*. Reading the factory first and the renderer second follows the same order the code runs in.

6.1. A Page Factory

Every page on a site tends to share a frame: the same **`<head>`**, the same header and footer, the same stylesheet links. Repeating that frame in every rexxlet is the kind of duplication that goes stale one copy at a time. **`apps/page.cls`** collects the frame in one place and exposes a small class, **`Page`**, that a rexxlet loads and uses to emit a complete document from just a title and a body.

A rexxlet loads the class by absolute path — anchored at the installation directory, which the request knows — and then, in the simplest case, builds a page in one line:

```
.context~package~loadPackage( .request~RexxHTTP_DIR"/apps/page.cls" )

.Page~new("My title")~body("<p>Hello</p>")~done
```

`new` takes the title, `body` sets the content, and `done` assembles the whole document and emits it. Because `body` returns **`self`**, the three calls chain. The class itself is barely more than those three methods and two resources.

6.1.1. Where the Body Comes From

Passing the body as a string is the explicit way. There is a second way that reads better in a rexxlet whose body is a lump of static HTML: declare the body as a **`::Resource`** and let the page find it. A rexxlet can write

```
.Page~new("My title")~done

::Resource BODY
<p>Hello</p>
::END
```

— no body argument at all — and the page picks up the **`BODY`** resource on its own. This is the part of the class worth slowing down for, because of *how* it finds that resource. The resource belongs to the rexxlet that called the page, not to **`page.cls`**, so the class cannot simply read its own resources: it has to look back at whoever invoked it. It does that by walking the call stack:

```
::Method page.Body Private
  Expose body
```

```
  -- 1) Body has been specified manually
  If body \== .Nil Then Return body

  -- 2) Look for a ::Resource BODY
  myself = .Context~package
  stackFrames = .context~stackFrames
  Do i = 1 To stackFrames~items
    exe = stackFrames[i]~executable
    If exe == .nil Then Iterate
    If exe~package \== myself Then Signal Found
  End

  -- 3) Not found: Return ""
  Return ""

Found:
  res = stackFrames[i]~executable~package~resource("BODY")
  If res \== .nil Then Return res~makeString
  Return ""
```

An explicit body, if there was one, wins immediately. Otherwise the method walks the stack frames from the top, skipping its own — the frames whose package is **`myself`** — until it reaches the first frame from a different package. That frame is the calling rexxlet, and its package is where a **`BODY`** resource, if the rexxlet declared one, lives. If nothing turns up, the page is simply empty. The whole search is a dozen lines and needs no cooperation from the caller: a rexxlet gets the convenience just by declaring a resource with the right name.

6.1.2. The Stylesheet Path

The frame links to a stylesheet, and that link has to be right whatever depth the current page sits at. A page served at the site root and a page served three directories down both need to reach the same **`css/rexxhttp.css`**, so the link cannot be absolute-from-root without knowing the mount point, and it cannot be a fixed number of **`../`** steps. The `done` method computes the right number of steps from two facts the request already carries:

```
depth   = .request~uri~countStr("/") - ,
          .request~RexxHTTP_URL~countStr("/")
cssPref = Copies("../", Max(depth, 0))
```

`uri` is the client's own URL, the one still showing in the address bar, and `RexxHTTP_URL` is the URL the installation is mounted at; counting the slashes in each and subtracting gives the page's depth below the mount point, and that many **`../`** steps climb back to it. The URL to count is `uri` and not, say, `document_uri`, because the browser resolves a relative link against the address it is sitting at — the client's URL, any trailing **`PATH_INFO`** and all. `document_uri` names the document Apache resolved, with that trailing tail already stripped and a DirectoryIndex folded into its served file; count over it and a request carrying a tail (**`/page.md/extra/stuff`**) yields too few **`../`**, and the stylesheet link points nowhere. The `ooRexx countStr` built-in does the counting; the `Max` keeps the count from going negative for a page sitting right at the mount point. From there, `done` substitutes **`cssPref`** and the title into the header and footer resources and emits the result:

```
Parse Value self~styleSheets With links "09"x options
out~append(                                  -
 .resources~page.header~makeString~          -
    changeStr("%title%",        title  )~ -
    changeStr("%stylelinks%",   links  )~ -
    changeStr("%styleoptions%", options)~ -
    changeStr("%css%",          cssPref)~ -
    changeStr("%home%",         brand  ), -
 .EndOfLine,                                 -
```

```
  self~page.Body,                    -
 .EndOfLine,                         -
 .resources~page.footer~makeString~  -
    changeStr("%css%",    cssPref)~  -
    changeStr("%home%",   brand)     -
)

Say out~string
```

The header and footer are held as **`::Resource`** blocks with **`%title%`**, **`%css%`** and **`%home%`** placeholders that `changeStr` fills in. Two more, **`%stylelinks%`** and **`%styleoptions%`**, take the tab-separated pair that `styleSheets` returns — one **`<link>`** per highlighting theme and one **`<option>`** to match — which is why the `Parse Value` sits just above the append. Emitting the page is one `Say`, and by now that is a familiar move: it appends to the buffered body, and the processor sends the bytes when the rexxlet ends. A shared frame, a body found by any of two routes, and a stylesheet link that is correct at any depth — the whole factory is a class small enough to read in one sitting.

Note

`styleSheets` builds those two lists by reading the **`rexx-STYLE.css`** sheets that actually sit in the **`css/`** directory: one **`<link>`** and one **`<option>`** per sheet it finds. Nothing is hard-coded, so adding a highlighting theme is just a file drop — the next page built picks it up.

6.2. A Markdown Renderer

The second rexxlet, **`md/index.rxl`**, does something a static site cannot: it turns every **`*.md`** file on the site into a styled HTML page at request time. Two lines of Apache configuration route every **`.md`** URL through it,

```
Action markdown /md
AddHandler markdown .md
```

so a request for **`/readme.md`** is handed to this rexxlet with the original file still identified by the request. The rexxlet reads that file, renders it, and wraps the result in the page factory from the previous section. It is short enough to read whole, and it is the clearest demonstration in the package of how much a rexxlet can do with the request object doing the routing work. Here it is, without its licence header:

```
  -- Load the page factory
 .context~package~loadPackage( .request~RexxHTTP_DIR"/apps/page.cls" )

 -- This servlet can be called in two different ways: when called directly,
 -- it prints help about itself; when called indirectly, using an Apache
 -- Action, it works as a preprocessor for Markdown.

 ------------------------------------------------------------------------------
 -- Direct request? Print some help and exit                                 --
 ------------------------------------------------------------------------------

 If .request~handler \== "markdown" Then Do
  .response~content_type = "text/html; charset=utf-8"
  .Page~new("RexxHTTP &mdash; the Markdown renderer")~body(.Resources~Help)~done
   Exit
 End
```

```
-------------------------------------------------------------------------------
-- Ok, we've been called as a handler:                                       --
--                                                                           --
--   Action markdown /md/index.rxl                                           --
--                                                                           --
-- In turn, .rxl files are also handled: by RexxHTTP.                        --
-------------------------------------------------------------------------------

-- Is pandoc actually usable?
Address COMMAND "pandoc -v" With Output Stem discard. Error Stem discard.

-- No? Serve the file verbatim as UTF-8 plain text.
If rc \== 0 Then Do
 .response~content_type = "text/plain; charset=utf-8"
  Say .File~readChars(.request~filename)~makeString
  Exit
End

-- We have Pandoc: we will emit HTML
.response~content_type = "text/html; charset=utf-8"

source = .File~readLines(.request~filename)

parserBin = .request~REXX_PARSER
If parserBin == "" Then
  parserAvailable = .False
Else
  parserAvailable = SysFileExists(parserBin"/Rexx.Parser.cls")

If parserAvailable Then Do
 .context~package~loadPackage( .request~REXX_PARSER"/FencedCode.cls" )
  -- "dark" is the default style for un-annotated fences; the client-side
  -- style chooser restyles those, and leaves author-styled ones alone.
  source = FencedCode( .request~filename, source, "dark" )
End

-- Run Pandoc and transform the output into a nice page
buffer = .Array~new
Address Command "pandoc -f markdown -t html" -
  With                                       -
    Input Using (source)                     -
    Output Using (buffer)

.Page~new("RexxHTTP")~body(buffer~makeString)~done

Exit
```

Three things in it are worth naming.

6.2.1. One File, Two Roles

The same file behaves differently depending on how it was reached. Asked for directly — someone visits **`/md/`** — it prints a short help page about itself. Reached through the Action, as the handler for some **`.md`** file, it does the rendering. It tells the two apart by reading one value from the request:

```
If .request~handler \== "markdown" Then ...
```

When Apache fires the Action, it records the name of the handler that routed the request. The request surfaces it as `handler`: the handler's name when the rexxlet was reached through an Action, or **`.Nil`** when it was asked for directly, with no Action in front of it. So the test reads as what it means — if the handler is not **`markdown`**, nothing routed here to render, and the rexxlet was reached directly and

serves its help; otherwise it renders the file. A rexxlet that documents itself when you visit it and works when you use it is a small courtesy, and it costs one comparison.

6.2.2. The File to Render

The rexxlet never parses a path or reconstructs a URL to find the Markdown file. It asks the request:

```
source = .File~readLines(.request~filename)
```

`filename` is the file on disk that the client asked for — the **`.md`** document, not this rexxlet — resolved by the request. The routing that got here through an Apache Action, with one URL standing in for another, is exactly the situation where working out the target file by hand is fiddly and easy to get wrong; the request has already done it. The renderer just reads what `filename` hands it. The same value is used for the plain-text fallback higher up, with `readChars` in place of `readLines`.

6.2.3. Degrading Gracefully

The renderer never assumes the tools it would like are present. It checks for Pandoc before using it, and if Pandoc is missing it serves the raw Markdown as plain text rather than failing — the document is always served, one way or another. It treats the Rexx Parser the same way: when the Parser is installed, the rexxlet loads its **`FencedCode.cls`** and pre-highlights the **```` ```rexx ````** fenced blocks into HTML before handing the document to Pandoc, which passes embedded HTML through untouched; when the Parser is absent, that step is skipped and the fences come out as plain code. The detail that makes this safe is that **`FencedCode.cls`** is loaded with `loadPackage` at run time, not with a **`::Requires`** directive — a **`::Requires`** for a file that is not there would stop the rexxlet from loading at all. Loading it dynamically, only after checking the Parser is present, is what lets one rexxlet run correctly on an install that has the Parser and on one that does not.

None of this is much code. The whole rexxlet is under a hundred lines, most of them either comment or the routine business of running a command and wrapping its output. What carries the weight is the request object — naming the file to render, naming the handler that routed here — and the page factory from the previous section, which turns rendered HTML into a finished page. That is the point of reading these two together: shown the request and response doing their part, a genuinely useful rexxlet is a short and readable thing.

6.3. Where to Go Next

These two rexxlets lean on the request and response objects without dwelling on how those objects come to hold what they hold. That machinery — the buffered body, the environment pool a name like `filename` or `document_uri` resolves against, the single commit — is the subject of the next chapter, *The Request/Response Model*, which is worth reading before building anything larger. The exhaustive, method-by-method account of the four classes is the *Class Reference*.

The Request/Response Model

The *Writing Rexxlets* chapter showed rexxlets that work without explaining why they work. This chapter is the why. It describes the three objects RexxHTTP builds for each request, the buffering that sits under the response, and the single moment — the *commit* — that turns a buffer full of headers and body into bytes on the wire. Almost everything that can surprise a rexxlet author, from a header that takes effect after it was "too late" to set, to one that mysteriously raises, follows from this one model.

7.1. The Three Objects

For each request, before the rexxlet runs, RexxHTTP builds three objects and hands the rexxlet the first two.

A **request** (**`HTTP.Request`**) is the incoming side, read-only. It holds the request line and headers — reached as named accessors or, for anything CGI passed as an environment variable, through the environment pool — and it holds the arguments and cookies the client sent, parsed and decoded. By the time the rexxlet runs the parsing is already done and the argument policy already applied, so the request a rexxlet sees is always well-formed.

A **response** (**`HTTP.Response`**) is the outgoing side, and it is the object this chapter is really about. It is made of three parts: a set of headers (seeded with one, **`Content-Type: text/plain; charset=utf-8`**), an optional set of cookies, and a buffered output stream that collects the body. The headers and cookies stay freely modifiable for as long as the body is being written — that is the whole point of the design — and are serialised only once, at commit.

An **output stream** (**`HTTP.OutputStream`**) is that body buffer: a write-only subclass of **`Stream`** that the response creates for itself. While the rexxlet runs, the default output stream is redirected to it, which is why a plain `Say` writes to the response body rather than to a console that is not there. The rexxlet almost never names this object; it just uses `Say`.

The request and response reach the rexxlet as the environment symbols **`.request`** and **`.response`**, placed there by the processor. Being environment symbols, they reach the two objects from any scope — a routine, a method — without the object having to be passed along, so the body of a rexxlet never has to thread **`response`** through every call that might need to set a header.

7.2. Buffering, and Why

A CGI program speaks to the web server through its standard output: the bytes it writes become the HTTP response, headers first, then a blank line, then the body. The catch is order. Once a single header byte has been written, the headers are decided; anything the program "sets" after that is too late, because it has already gone out — or worse, the body has, and the header lands in the middle of it.

RexxHTTP removes that catch by not writing the body straight to the server. The output stream collects it in a buffer instead. While the buffer fills, the headers and cookies are still just entries in two directories on the response object: setting one, changing one, adding a cookie are all ordinary updates to in-memory state, and the order in which the rexxlet does them relative to its `Say`s does not matter, because none of it has been serialised yet.

This is what the introduction and the *Writing Rexxlets* chapter called *lazy headers*. A rexxlet can write half its page, decide from what it computed there that the content type should be **`text/html`** rather than the default, set it, and finish the page — and the header is correct on the wire, because at the moment it was set nothing had yet been committed. The buffer buys the rexxlet the freedom to decide late.

7.3. Commit

Commit is the moment the buffered head becomes bytes. It happens once per response, and it does a fixed sequence: it writes the **`Content-Type`** header, then one **`Set-Cookie`** line per scheduled cookie, then every other header, then the blank line that separates head from body — all straight to the server's output stream, ahead of the buffered body. After commit the head is on the wire and cannot be changed.

Commit is **idempotent**: the first call emits the head and marks the response committed; any later call returns at once, having done nothing. A rexxlet therefore never has to track whether the response is "already" committed before asking for a flush — asking twice is harmless.

Rexxlets rarely call `commit` directly. It is driven by *flush*. The response's `flush` (and the output stream's, which it delegates to) commits the response if it has not been committed yet, and only then writes the buffered body out and empties the buffer. So the **first flush is what commits**, and that first flush is one of two things:

- one the rexxlet performs itself, by calling **`.response~flush`** — for instance to fix the headers at a deliberate point; or
- the one RexxHTTP performs automatically after the rexxlet returns.

A rexxlet that never flushes still gets exactly one flush, the processor's final one, and so still produces a complete response with its head committed once. The common case needs no `flush` at all.

7.4. Commit Discipline

Everything a rexxlet can do to the head — set a header (**`.response[...]=`** or the message form), schedule a cookie (`addcookie`), or replace the response wholesale (`error`, `404`, `redirect`) — is legal *before* commit and an error *after* it. The reason is physical: past commit the head is already on the wire, so a late change could not possibly take effect. RexxHTTP treats the attempt as a programming error and raises (**`Syntax`** 93.900, *the response is already committed*) rather than accept a call it would have to silently ignore. The body may still be written and flushed after commit; only the head is frozen.

This is why the order of guards matters in one place worth noting. When a cookie is added to an already-committed response, the failure reported is *already committed*, not *not a cookie* even if the argument is also wrong: the commit check comes first, because the commit state outranks argument validation. The same fail-fast governs `error` and `redirect`, which discard an uncommitted buffered body and do their work, but raise on a committed response instead of pretending to change a status that has already shipped.

There is also a brief internal window the rexxlet never sees but which explains the care in the design: while `commit` is partway through writing the head, the response is *committing* but not yet *committed*. RexxHTTP's own error handling distinguishes the two, so that a fault raised in the middle of emitting headers does not try to re-assert headers over a response that is already half-written. A rexxlet only ever observes the two stable states, before and after; `committed` reports which. The per-method detail — what `error`, `addcookie`, `commit` and `committed` each do, argument by argument — is in the *Class Reference*.

7.5. A Note on Streaming and mod_deflate

It is tempting to read "the rexxlet can flush early" as "the rexxlet can stream a page to the browser progressively, a piece at a time." The 2006 manual made roughly that promise. In a modern production deployment it no longer holds, and this manual will not repeat it.

The reason is **`mod_deflate`**. To compress the response, Apache's **`mod_deflate`** buffers the CGI program's output on its own account before sending it on, which means the boundaries at which a rexxlet flushes are not the boundaries at which the client receives anything: the compressor decides those. An early **`.response~flush`** does *not* reliably make the first half of a page appear in the browser before the rexxlet has finished computing the second half.

What an early flush still does — and the reason the flush mechanism is kept — is *internal* and entirely real. It commits the response at a chosen point, fixing the headers and status exactly there rather than at the end. That control over commit ordering is occasionally what a rexxlet wants, and it is the mechanism behind the lazy-header behaviour this chapter described. Think of `flush` as “decide the head now,” not as “push bytes to the browser now.” For the great majority of rexxlets neither concern arises: they write their body, return, and let the processor's final flush commit and send the whole thing at once.

Class Reference

8.1. The `HTTP.Cookie` Class

An **`HTTP.Cookie`** instance represents one cookie the response will set: one instance, one **`Set-Cookie`** header. The rexxlet creates the cookie, sets its attributes, and hands it to **`.response~addcookie`**:

```
c = .HTTP.Cookie~new("session", "abc123")
c~max_age  = 86400
c~samesite = "Lax"
c~httponly = 1
.response~addcookie(c)
```

This is a response-side class only. Cookies arriving *with* the request are not **`HTTP.Cookie`** instances; they reach the rexxlet as plain name/value pairs through **`.request~cookie`** (see the **`HTTP.Request`** reference).

`HTTP.Cookie` is a modern, RFC 6265 cookie: lifetime is expressed with **`Max-Age`** (the class does not emit **`Expires`**, which survives only for user agents long gone — when both are present, **`Max-Age`** wins anyway), and the **`SameSite`**, **`Secure`** and **`HttpOnly`** attributes and the **`__Host-`** and **`__Secure-`** name prefixes are supported.

The class validates eagerly, so that an invalid cookie fails in the rexxlet, never as a **`Set-Cookie`** line the browser silently discards. An invalid *name* is rejected at creation. Rules that involve *several* attributes are checked when the cookie is serialised — in practice, at `addcookie` — so that attributes may be set in any order.

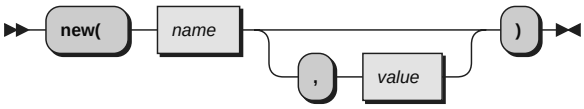
8.1.1. Consistency Rules

A cookie must satisfy all of these to serialise:

- **`SameSite=None`** requires **`Secure`**.
- A name with the **`__Secure-`** prefix requires **`Secure`**.
- A name with the **`__Host-`** prefix requires **`Secure`**, **`Path=/`**, and no **`Domain`**.

Prefix matching is case-insensitive, mirroring what user agents do: **`__host-`** is **`__Host-`**.

8.1.2. `new`

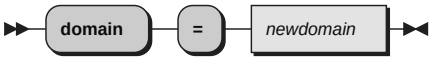


Creates a cookie named *name* with the given *value* (default: the null string — a cookie with a name and an empty value). The name is validated at once: it must be a token — no control characters, spaces, or separators — or creation raises. Everything else defaults to the quietest setting; set the attributes needed afterwards.

8.1.3. `domain`

The **Domain** attribute, emitted as given. The default, the null string, omits the attribute. A **__Host-** cookie must not set one.

8.1.4. domain=

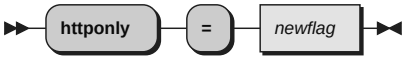


Sets the **Domain** attribute; see *domain*.

8.1.5. httponly

The **HttpOnly** attribute: **1** emits it, **0** — the default — does not.

8.1.6. httponly=



Sets the **HttpOnly** attribute; see *httponly*.

8.1.7. makestring

Returns the value of the **Set-Cookie** header this cookie will produce — exactly what `addcookie` will schedule and `commit` will emit. The consistency rules are enforced here, so a cookie that violates them raises at this point rather than going out malformed.

```
id=v; Max-Age=3600; Domain=example.org; Path=/; SameSite=Lax; Secure; HttpOnly
```

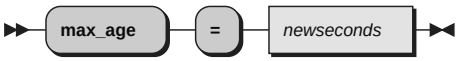
Rexxlets rarely need `makestring` — `addcookie` drives it — but it is convenient for logging or testing what a cookie will look like on the wire.

8.1.8. max_age

The cookie's lifetime, a whole number of seconds. A negative value — the default is −1 — makes it a session cookie: the attribute is omitted. Zero instructs the user agent to delete the cookie, which is also how a cookie is *deliberately* deleted — set **Max-Age** to zero on a cookie with the same name (and the same **Path** and **Domain**, if the original had them):

```
gone = .HTTP.Cookie~new("session", "")
gone~max_age = 0
.response~addcookie(gone)
```

8.1.9. max_age=



Sets the cookie's lifetime; see *max_age*.

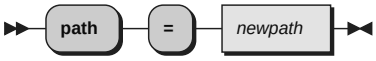
8.1.10. name

Returns the cookie's name. There is no `name=`: a cookie's name is fixed at creation.

8.1.11. path

The **`Path`** attribute, emitted as given. The default, the null string, omits the attribute. A **`__Host-`** cookie must have **`Path=/`**.

8.1.12. path=

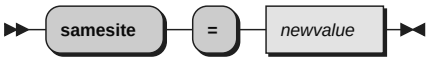


Sets the **`Path`** attribute; see *`path`*.

8.1.13. samesite

The **`SameSite`** attribute: **`Strict`**, **`Lax`** or **`None`**, accepted case-insensitively and stored in canonical spelling (the getter returns **`Strict`**, **`Lax`** or **`None`** regardless of how it was assigned). The null string — the default — omits the attribute; assigning it clears a previously set value. **`SameSite=None`** requires **`Secure`**; the rule is checked at serialisation, not here, so the two may be assigned in either order.

8.1.14. samesite=

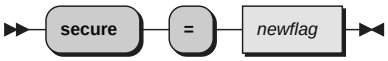


Sets the **`SameSite`** attribute; see *`samesite`*.

8.1.15. secure

The **`Secure`** attribute: **`1`** emits it, **`0`** — the default — does not. Required by **`SameSite=None`** and by the **`__Secure-`** and **`__Host-`** name prefixes.

8.1.16. secure=

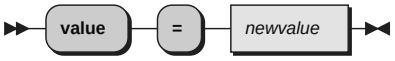


Sets the **`Secure`** attribute; see *`secure`*.

8.1.17. value

The cookie's value. As at creation, the new value is taken as given — encoding reserved characters is the caller's business.

8.1.18. value=



Sets the cookie's value; see *value*.

8.2. The HTTP.OutputStream Class

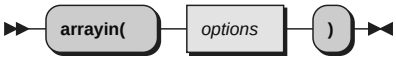
An **HTTP.OutputStream** collects the response body. The rexxlet does not create it: the response object does, and **.response~output** returns it. During rexxlet execution the default output stream (**.output**) is redirected to it, so the three statements below write to the same place:

```
Say "one"                          -- the usual way
.output~say("two")                 -- explicitly, via the default stream
.response~output~say("three")      -- explicitly, via the response
```

Nothing reaches the client until the stream is flushed; the first flush commits the response — headers out first, then the buffered body. This buffering is what makes lazy headers possible, and it is the reason the class exists; *The Request/Response Model* tells that story. The rexxlet rarely needs to do anything: the processor flushes once after the rexxlet returns.

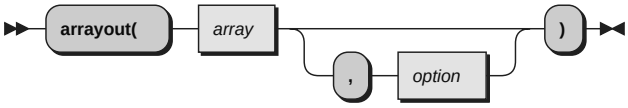
The class is a subclass of **Stream** and honours the stream protocol as far as it makes sense for what it is: a *write-only*, *transient* stream. It cannot be read (the input methods raise NOTREADY) and it cannot be positioned. Lines written to it end in LF, the web convention for text bodies.

8.2.1. arrayin



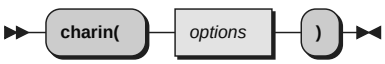
Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **0**, or **.nil**, as appropriate to the method).

8.2.2. arrayout



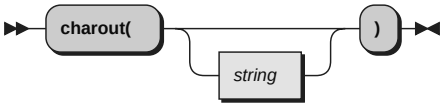
Appends the elements of *array* — a single-dimensional array, or an object that produces one when sent a **request("ARRAY")** message — to the body. With *option* **"LINES"** (the default; **"L"** will do) the elements are joined by LF, with no LF after the last one; with **"CHARS"** (**"C"**) they are concatenated with no separator.

8.2.3. charin



Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **0**, or **.nil**, as appropriate to the method).

8.2.4. charout



Appends *string* to the body exactly as given — no line end. The argument must have a string value. Returns **0** on success. Called with no argument, charout closes the stream (here: flushes it — see close). The stream protocol's positioning argument is not accepted: this is a transient stream.

```
.response~output~charout("no trailing newline here")
```

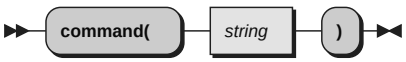
8.2.5. chars

Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **0**, or **.nil**, as appropriate to the method).

8.2.6. close

Flushes any pending output. An HTTP output stream has no real notion of being closed — it remains writable afterwards; *close* is how the stream protocol spells “I am done for now”, and here that means a flush.

8.2.7. command



The stream-protocol command interface: the verbs **OPEN**, **CLOSE**, **FLUSH**, **QUERY**, **SEEK** and **POSITION** are accepted and dispatched to the corresponding methods.

8.2.8. description

Returns the stream's state, followed by a description when there is something to describe — typically only when the underlying stream was not ready when the output stream was created.

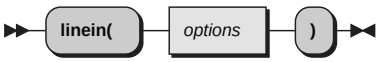
8.2.9. flush

The workhorse. Commits the response if it is not yet committed — this is where the headers get written, in front of everything — then writes the buffered body to the underlying CGI stream, flushes it, and empties the buffer. **.response~flush** does the same thing through the response object. Flushing explicitly mid-rexxlet is allowed, and fixes the point past which headers can no longer change; what it does *not* promise is client-visible streaming — see *The Request/Response Model*.

8.2.10. `init`

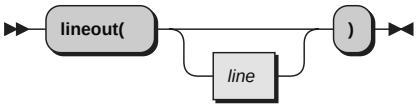
Rexxlets do not construct output streams: the response object creates one, wrapped around the real CGI output stream, when *it* is created. The wrapped stream must be a **`Stream`** instance proper — in particular, output streams do not nest: an **`HTTP.OutputStream`** will not accept another one as its underlying stream.

8.2.11. `linein`



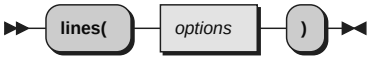
Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **`0`**, or **`.nil`**, as appropriate to the method).

8.2.12. `lineout`



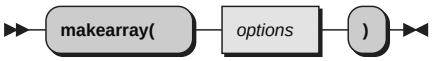
Appends *`line`*, which must have a string value, followed by LF. Returns **`0`** on success. Called with no argument, it closes (flushes) the stream, like `charout`. The positioning argument is not accepted.

8.2.13. `lines`



Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **`0`**, or **`.nil`**, as appropriate to the method).

8.2.14. `makearray`

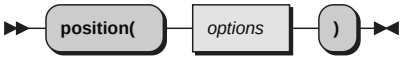


Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply — the call is ignored and returns a neutral value (the null string, **`0`**, or **`.nil`**, as appropriate to the method).

8.2.15. `open`

An HTTP output stream is always open, unless its underlying stream reported a problem when the output stream was created — in which case it cannot be opened at all. The method returns the stream's state and changes nothing.

8.2.16. `position`

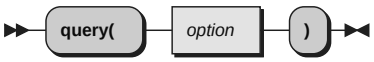


Not supported: a transient stream cannot be positioned, so the call raises. `position` is a synonym for *`seek`*.

8.2.17. `qualify`

Returns the stream's name, the fixed label **`HTTP.OutputStream:1`**.

8.2.18. `query`

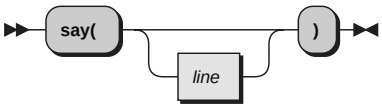


The stream-protocol query interface, answered as befits a transient stream that exists only for the duration of the request: **`EXISTS`** returns the stream's label, **`SIZE`** the null string, **`STREAMTYPE`** returns **`UNKNOWN`**, **`DATETIME`** and **`TIMESTAMP`** return the stream's creation time, and the **`POSITION`**/**`SEEK`** queries return **`1`**. **`HANDLE`** is not supported.

8.2.19. `reset`

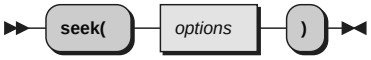
Empties the unflushed buffer, discarding whatever body the rexxlet has written so far. Because the response is not committed until the first flush, a rexxlet that hits an error after writing some output can call `reset` to throw that partial body away and emit a clean error page from scratch — see *`flush`* for when the commit actually happens.

8.2.20. `say`



`lineout` with a default: called without an argument it appends an empty line rather than closing the stream.

8.2.21. `seek`



Not supported: a transient stream cannot be positioned, so the call raises. `seek` is a synonym for *`position`*.

8.2.22. `state`

The stream's state — **`READY`** in any response ever to be sent.

8.2.23. `supplier`

Not supported. The stream is write-only, so calling this method raises the NOTREADY condition, with a description naming the stream. If the condition is not trapped, the usual ooRexx semantics apply

— the call is ignored and returns a neutral value (the null string, **0**, or **.nil**, as appropriate to the method).

8.2.24. underlyingstream

Returns the wrapped stream: the real CGI output, connected to the web server. This is the response's plumbing — `commit` writes the headers through it — and a rexxlet has no business writing to it directly: anything sent there bypasses the buffer and lands in front of content the response has not emitted yet, headers included.

8.3. The HTTP.Request Class

An **HTTP.Request** instance represents the request being served. The rexxlet does not create it: the processor instantiates one per incoming request, before the rexxlet runs, and hands it to the rexxlet as its first argument; like the response, it is also published in the global environment, as **.request** (the two forms are equivalent — see the introduction to **HTTP.Response**).

The request object is read-only, and answers three kinds of questions: what came with the request line and the headers (the environment pool, and a set of accessors), what arguments the query string or the request body carried (`arg`), and what cookies the user agent sent (`cookie`).

By the time the rexxlet runs, the groundwork is done: the processor has parsed the arguments — reading the request body, for a POST — and applied the argument policy, so the arguments a rexxlet sees are always well-formed; and creating the request has set the current directory to the directory of the rexxlet file, so relative paths in the rexxlet resolve next to it.

8.3.1. The Environment Pool

CGI delivers a request as environment variables, and the request object embraces that: any message that does not match one of the methods below answers with the value of the environment variable of the same name.

```
agent  = .request~http_user_agent     -- the User-Agent request header
server = .request~server_name         -- a standard CGI variable
raw    = .request~http_cookie         -- the raw Cookie header
```

A variable that is not set answers the null string. The pool is read-only: these messages accept no arguments, and there is no assignment form. The message name arrives uppercased — the usual ooRexx behaviour — which is just what the upper-case names of the CGI world expect.

The lookup understands the deployment it lives in. Under an **Action**-based deployment, Apache reaches the processor through an internal subrequest, and hands some variables over under a **REDIRECT_** prefix. The request object absorbs that detail:

- A name beginning with **HTTP_** — a request header, in the form Apache passes them — is looked up as given.
- Any other name is looked up as given first, and, only if that comes back empty, under a **REDIRECT_** prefix. So the standard CGI variables, which arrive unprefixed, are never shadowed, while a value that exists only behind the **Action** subrequest's **REDIRECT_** is still found.

So **.request~remote_addr** simply works, and a **SetEnv MYAPP_MODE production** in an **.htaccess** is **.request~myapp_mode** wherever Apache actually put the value — prefixed or not. There is no special namespace: an operator-defined variable is an ordinary pool name. If the prefixed

form is ever wanted explicitly, it can always be named: **`.request~redirect_remote_user`**. **`REXXHTTP_ARGPOLICY`**, the argument-policy switch, is read through this same pool — which is why a per-directory **`SetEnv`** is all it takes to configure it (next section).

8.3.2. Arguments, and the Strict Policy

Before the rexxlet runs, the processor parses the request arguments — the query string of a GET, the **`application/x-www-form-urlencoded`** body of a POST — and applies the *argument policy*. The policy of the 1.0, **`strict`**, the default and the only one, demands of every parameter that:

- it has the form *`name`*`=`*`value`* — an empty value (**`k=`**) is legitimate, a token without **`=`** is not;
- the name is not empty and does not begin with a digit (the ooRexx symbol convention — and a numeric name would be unreachable through `arg` anyway, since positions take precedence there);
- the name is unique, compared caselessly: **`X`** and **`x`** are the same name.

These rules are applied to the *decoded* name — the one the rexxlet will see and key on. Two spellings that decode to one name (say **`user+name`** and **`user%20name`**) are therefore a single name, and a request carrying both is rejected as a duplicate.

A request that breaks one of these rules is answered with **`400 Bad Request`**, stating the reason, and the rexxlet never runs. Stray **`&`**s — doubled, or trailing — are ignored, not rejected. The policy is configured per directory or per rexxlet with **`SetEnv REXXHTTP_ARGPOLICY`** in the Apache configuration; the value is caseless, **`strict`** is the only value the 1.0 recognises, and an unknown value is answered with a **`500`** configuration error — an operator mistake, not a client one.

The payoff is what the rexxlet gets to assume: every argument it sees has a name and a value, and a name names exactly one argument.

Argument names and values alike are delivered decoded — percent-escapes, and the **`+`**-for-space convention of form encoding, applied to both sides of each pair as **`application/x-www-form-urlencoded`** prescribes. So a field named **`user name`**, which a form sends as **`user+name`** (or **`user%20name`**), is reached as **`arg("user name")`**.

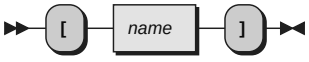
8.3.3. `decodeForm` (Class Method)

The class-method face of *`decodeForm`*: decodes one **`application/x-www-form-urlencoded`** field — **`%xx`** triples and **`+`**-for-space — with no request instance in hand. The body lives here, on the class; the instance method delegates to it. The processor uses it internally to split a form body into its fields.

8.3.4. `decodeURIComponent` (Class Method)

The class-method face of *`decodeURIComponent`*: percent-decodes *`string`* with no request instance in hand. The body lives here, on the class; the instance method delegates to it. The processor uses it internally to decode path and query components.

8.3.5. `[]`



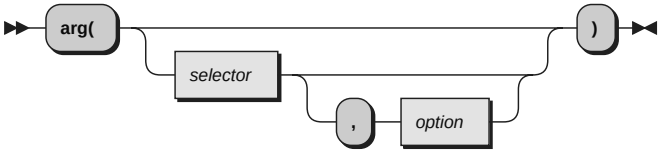
Bracket lookup: *name*, which must have a string value, is uppercased, its hyphens become underscores, and the result is looked up in the environment pool — always, never against a like-named method. The bracket form is the raw-variable door: it reaches an environment variable by its HTTP spelling even when a method shares the name.

```
length = .request["Content-Length"]    -- the CONTENT_LENGTH variable, raw
agent  = .request["HTTP-User-Agent"]   -- the HTTP_USER_AGENT variable
```

This is the complement of the message form: **`.request~content_length`** returns the method (the parsed length), while **`.request["Content-Length"]`** returns the raw environment variable of that name. Use **`~name`** for the library's view, **`[name]`** for the variable underneath.

There is no bracket assignment: the request is read-only.

8.3.6. `arg`



Access to the request arguments, modelled on the built-in `Arg` function. Called with no arguments, it returns how many there are. Otherwise the first argument, which must be given and must have a string value, selects one — by position, when it is a positive whole number; by name, compared caselessly, in any other case — and *option* says what to return about it. The option is identified by its first nonblank character, caselessly, again in the manner of the built-in:

- **`Value`** — the default — returns the argument's value, decoded; the null string when there is no such argument.
- **`Name`** returns the argument's name, in its original spelling.
- **`Exists`** returns whether such an argument exists.
- **`Omitted`** returns the opposite.

```
-- /rexxlet?q=hello&Lang=ca
.request~arg                    -- 2
.request~arg(2, "Name")         -- "Lang" (the spelling is preserved...)
.request~arg("lang")            -- "ca"   (...but lookup is caseless)
.request~arg("page", "Exists") -- 0
```

Positional access reaches the arguments in the order the request sent them; a position beyond the count yields the null string (and **`Exists`** and **`Omitted`** answer accordingly). Under the strict policy these are the only shapes a request can have: see *Arguments, and the Strict Policy* for what `arg` never has to cope with.

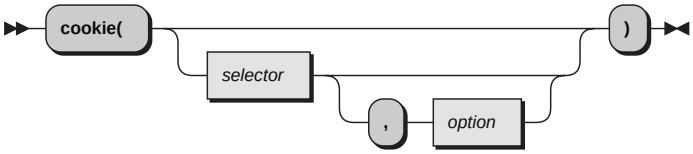
8.3.7. `content_length`

The size of the request body, in bytes, as the request declared it; the null string when the request carried none (a GET, typically).

8.3.8. content_type

The media type of the request body, as the request declared it — for a form POST, **`application/x-www-form-urlencoded`**; the null string when the request carried none.

8.3.9. cookie



Access to the cookies the user agent sent, with the same shape as `arg`: no arguments for the count, a first argument — required, with a string value — that selects by position when it is a positive whole number and by name in any other case, and an *`option`* choosing among **`Name`**, **`Value`** (the default), **`Exists`** and **`Omitted`**, with the same meanings as in `arg`. Two differences, both deliberate. Lookup by name is case-*sensitive*: **`session`** and **`Session`** are different cookies, to the request object as to the user agent. And the option is given as an abbreviation of the option word — any leading part will do, caselessly — rather than by its first character alone.

Cookie values are delivered percent-decoded, and only percent-decoded: the **+**-for-space convention belongs to form encoding, not to cookies, so a literal **+** in a value survives the trip. This is the reading half of the round trip stated in the **`HTTP.Cookie`** reference — encode on output, read plain on input — and it is safe for the whole repertoire: an encoded **`;`** (**`%3B`**) comes back as a **`;`** *inside* the value, never as a cookie separator.

When the request carries several cookies with the same name — the same name under different **`Path`**s, say — positional access reaches every one; lookup by name resolves to the last one sent.

```
If .request~cookie("session", "Exists") Then
  session = .request~cookie("session")
```

8.3.10. decodeForm

Decodes one field of an **`application/x-www-form-urlencoded`** body: **`%xx`** triples become their bytes, *and* a **+** becomes a space. Since a conforming form serialiser emits a space as **+** but a hand-built query string may use **`%20`**, it accepts both. It is the inverse of the response object's `encodeForm`, and the routine the processor uses internally to split a form body into its fields. This is the instance face; the body lives on the class method *`decodeForm` (Class Method)*, to which it delegates.

8.3.11. decodeURIComponent

Reverses percent-encoding: each **`%xx`** triple, where **`xx`** is two hex digits, becomes the byte it names; everything else passes through untouched. It is the inverse of `encodeURIComponent` on the response object, and the routine the processor uses internally to decode path and query components. A lone **%** not followed by two hex digits is left as a literal **%**, so malformed input degrades gracefully rather than raising. This is the instance face; the body lives on the class method *`decodeURIComponent` (Class Method)*, to which it delegates.

8.3.12. document_uri

The URL-space path of the document Apache actually resolved and served, carrying its URL prefix (for example **`/mtx/target.mtx`** or **`/ali/sub/index.mtx`**). It is the disk-to-URL twin of *`filename`*, exactly as *`script_name`* is the twin of *`script_filename`*: `filename` names the served document on disk, `document_uri` names that same document in URL space.

It therefore resolves what *`uri`* leaves raw. Because `uri` is the client's typed URL with the query string removed, it keeps any extra `path_info` tail and, for a **`DirectoryIndex`** request, names the directory. `document_uri` instead follows `filename`: the extra tail is gone and a **`DirectoryIndex`** is resolved to the index actually served. Under an Action it names the original document, not the processor.

Where the **`REXXHTTP_BASE`** / **`REXXHTTP_URLBASE`** pair is not configured, it falls back to the stored *`uri`*, so pre-anchor deployments are unaffected.

See also method *`filename`*.

See also method *`script_name`*.

8.3.13. filename

The full filesystem path of the rexxlet file. `path_translated` answers the same.

8.3.14. handler

The name of the handler that routed the request, or **`.Nil`** when there was none. When Apache reaches a rexxlet through an **`Action`** directive, it records the handler's name — the name given to the **`Action`** — and the request surfaces it here. A rexxlet asked for directly, with no Action in front of it, was routed by no handler, and `handler` answers **`.Nil`**.

This lets one file serve two roles and tell them apart: a rexxlet that is both reachable on its own and wired as a handler can branch on whether `handler` names the handler it expects. See *One File, Two Roles* for a worked example, and the boolean twin *`is_action`*.

8.3.15. is_action

`.True` when the request reached the rexxlet through an Apache **`Action`** — that is, when a handler routed it — and **`.False`** otherwise. It is the boolean reading of *`handler`*: true exactly when `handler` is not **`.Nil`**. Reach for `is_action` when only the fact of the routing matters, and for `handler` when the handler's name does.

8.3.16. method

The request method, as sent: **`GET`**, **`POST`**, and so on. `method` and *`request_method`* are one method under two names.

See also method *request_method*.

8.3.17. path_info

The extra path: what the request URI carries beyond the part that locates the rexxlet, decoded, **/**-separated. A request for **/test/show/reports/2026**, served by the rexxlet file **show**, has the `script_name` **/test/show** and the `path_info` **/reports/2026**. The null string when there is none.

8.3.18. path_translated

The full filesystem path of the rexxlet file — where the request URI landed once Apache translated it. `filename` answers the same.

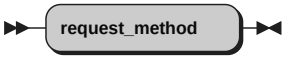
8.3.19. post_string

The body of a POST request, exactly as received — still in its transfer encoding; the processor has already read it by the time the rexxlet runs. The null string for a GET. The parsed, decoded view of the same data is `arg`.

8.3.20. query_string

The query string, exactly as received — still encoded, without the **?**. The null string when there is none. The parsed, decoded view of the same data is `arg`.

8.3.21. request_method



The request method, as sent. `request_method` and *method* are one method under two names.

See also method *method*.

8.3.22. request_uri

The request URI exactly as the client sent it: path and query string, still encoded. `request_uri` and *unparseduri* are one method under two names.

See also method *unparseduri*.

8.3.23. request_url

The self-referential URL of the request: the client's own URL with the query string peeled off — a URL to hang return links off, not the CGI processor path. It differs from *`script_name`* for a **`DirectoryIndex`** request: when the client asks for **`/test/`**, `request_url` answers **`/test/`** (the URL as typed), whereas `script_name` answers the RFC 3875 resolved resource, **`/test/index.rxl`**. For a rexlet the client names directly, the two coincide.

This value was named `script_name` until the CGI **`SCRIPT_NAME`** of RFC 3875 took over that name; it is the value the production clients actually consumed.

See also method *`script_name`*.

8.3.24. RexxHTTP_DIR

The filesystem directory of the RexxHTTP package, with no trailing **`/`**.

See also method *`RexxHTTP_URL`*.

8.3.25. RexxHTTP_URL

The URL of the RexxHTTP package, or the empty string if RexxHTTP is not reachable. When it is not empty it always ends with **`/`**, following the **`mod_dir`** convention.

See also method *`RexxHTTP_DIR`*.

8.3.26. script_filename

The filesystem path of the processor — the rexxlet Apache resolved and ran, the disk-space twin of *`script_name`*. When the client names a rexlet directly, this equals *`filename`*. Under an Action the two differ: `script_filename` names the rexlet, while `filename` and *`path_translated`* name the document it serves.

See also method *`script_name`*.

See also method *`filename`*.

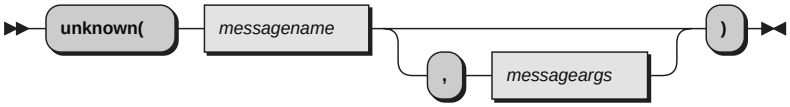
8.3.27. script_name

The URI path that locates the rexxlet, decoded: `uri` without the `path_info`.

8.3.28. system_version

The processor identification string — for this release, **`REXXHTTP/1.0 20260611`**.

8.3.29. unknown



The environment pool described at the top of this reference is implemented by `unknown`: a message that matches no method answers the corresponding environment variable. The get form only — a pool message carries no arguments, and one that does is answered with a syntax error rather than a value. See *The Environment Pool* for the lookup rules.

8.3.30. unparseduri

The request URI exactly as the client sent it. `unparseduri` and *`request_uri`* are one method under two names.

See also method *`request_uri`*.

8.3.31. uri

The path of the request, decoded, query string excluded: `script_name` plus `path_info`.

8.3.32. validate

The processor's hook. Parses the arguments eagerly — reading the body of a POST — and applies the argument policy, returning the null string when the request is acceptable, and a short **`code:detail`** description of the first violation otherwise. The processor calls it before the rexxlet runs — this is where the **`400`** of the strict policy comes from — so a rexxlet never needs to: by the time a rexxlet runs, `validate` has already returned the null string. It is public so the processor, which is a separate program, can reach it; it is not part of the face a rexxlet uses.

8.4. The HTTP.Response Class

An **`HTTP.Response`** instance represents the response being built for the current request. The rexxlet does not create it: the processor instantiates one response per request, before the rexxlet runs, and hands it to the rexxlet as its second argument. It is also published in the global environment, so both of these reach the same object:

```
Use Arg request, response      -- the second argument the processor passes...
...
.response~flush                -- ...or the environment symbol
```

Since the processor passes the response to the rexxlet, it can be picked up as the second argument; but the environment symbol works in any scope — an internal routine, a **`::Routine`**, a method — without the variable having to travel there, and that is the form, **`.request`** and **`.response`**, this manual uses throughout.

A response is made of three parts: a set of *headers*, an optional set of *cookies*, and a buffered *output stream* that collects the body. The headers and cookies remain freely modifiable while the rexxlet

writes the body; they are serialised only when the response is *committed*, which happens at the first flush (see *The Request/Response Model*). After the rexxlet returns, the processor issues a final flush, so a rexxlet that never touches the response object beyond `Say` still produces a complete, well-formed response.

8.4.1. Headers, and the Two Ways to Name Them

Headers live in a directory keyed by the upper-cased header name. Two different syntaxes reach that directory, and they normalise the name differently — deliberately so, because each syntax has its own grammar:

- **Message form. `.response~cache_control = "no-store"`** and its read counterpart **`.response~cache_control`**. A hyphen is not legal inside a ooRexx message name (it would parse as a subtraction), so the message form uses an underscore wherever the HTTP name has a hyphen; the response object translates underscores to hyphens to reach the real header name.
- **Bracket form. `.response["Cache-Control"] = "no-store"`** and **`.response["Cache-Control"]`**. Here the name is an ordinary string, the hyphen poses no problem, and the name is used as written; only its case is normalised.

Both routes converge on the same internal key (**`CACHE-CONTROL`**), each starting from what its own grammar allows. One practical consequence: a header whose real name contains an underscore can only be reached through the bracket form, since the message form would turn that underscore into a hyphen.

Header names are case-insensitive in HTTP. Internally they are stored upper-cased — that is what makes the lookup case-insensitive, so that the message form, the bracket form and any mix of case all reach the same entry. On the way out, though, the name is title-cased for readability: the response to the example above carries the header line **`Cache-Control: no-store`**, not **`CACHE-CONTROL: no-store`**. The rule is mechanical — each hyphen-separated word gets an initial capital — so a few acronym headers come out unconventionally (**`WWW-Authenticate`** would emit as **`Www-Authenticate`**). RexxHTTP never sends those itself, and since HTTP treats the names case-insensitively no client minds; only a test that compares a raw header block case-sensitively need take note.

Every response carries a **`Content-Type`**: the header directory is seeded at creation time with the default **`text/plain; charset=utf-8`**, which the rexxlet may overwrite but not remove.

8.4.2. Encoding Text for Output

Four methods turn raw text into a form that is safe to drop into HTML or into a URL. They are the counterpart to the two decoders on the request object (`decodeURIComponent` and `decodeForm`), which undo the URL forms on the way in; there is deliberately no HTML *decoder*, since a rexxlet emits HTML but is never handed HTML to parse.

Each method exists twice: as a class method, callable with no response in hand (**`.HTTP.Response~encodeHTML(s)`**), and as an instance method on a live response (**`.response~encodeHTML(s)`**). The two are identical — the instance form delegates to the class form — so a rexxlet uses whichever is closer to hand. None of them touches the response; they are pure string functions that happen to live here because emitting safe output is the response's job. The class methods are grouped first below, then each name reappears as an instance method in its alphabetical place.

8.4.3. The Status Line

There is no first-class status method, and none is needed. Under CGI the response status is set through the **`Status`** pseudo-header (RFC 3875), which to the response object is a header like any other:

```
.response~status = "404 Not Found"
```

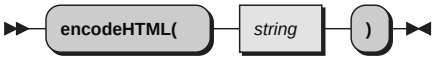
`commit` emits it with the rest of the headers; Apache consumes it and builds the corresponding HTTP status line (the **`Status`** line itself does not reach the client). A response that sets no status is a **`200 OK`**.

Setting **`status`** by hand is the low-level route. For the common cases — sending an error page or redirecting — `error`, `404` and `redirect` set the status, the matching headers and (for errors) a body in one call, and are written to be the last statement a rexxlet runs: **`Exit .response~error(404)`**. The reason phrase that goes with a code is supplied by `statusReason`.

8.4.4. `encodeForm` (Class Method)

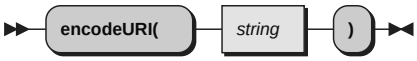
The class form of `encodeForm`, callable without a response in hand (**`.HTTP.Response~encodeForm(text)`**). The body lives here; the instance method *`encodeForm`* delegates to it. See that entry for what the encoding does.

8.4.5. `encodeHTML` (Class Method)



The class form of `encodeHTML`, callable without a response in hand (**`.HTTP.Response~encodeHTML(text)`**). The body lives here; the instance method *`encodeHTML`* delegates to it. See that entry for what the encoding does.

8.4.6. `encodeURI` (Class Method)



The class form of `encodeURI`, callable without a response in hand (**`.HTTP.Response~encodeURI(text)`**). The body lives here; the instance method *`encodeURI`* delegates to it. See that entry for what the encoding does.

8.4.7. `encodeURIComponent` (Class Method)

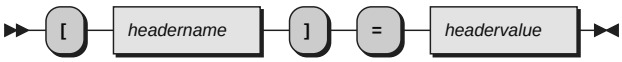
The class form of `encodeURIComponent`, callable without a response in hand (**`.HTTP.Response~encodeURIComponent(text)`**). The body lives here; the instance method *`encodeURIComponent`* delegates to it. See that entry for what the encoding does.

8.4.8. `[]`

```
value = .response[headername]
```

Returns the current value of header *headername*, or **.nil** if no such header has been set. The name is case-insensitive and is given literally, hyphens included. *headername* must have a string value.

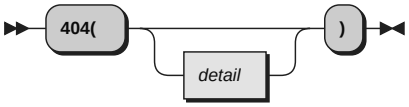
8.4.9. []=




```
.response[headername] = headervalue
```

Sets header *headername* to *headervalue*, creating or overwriting it. Headers are single-valued: assigning to an existing name replaces its value. Both arguments must have string values. Headers may be set after body output has begun — that is the point of the design; see *The Request/Response Model* — but must be set before the first flush: once the response is committed the head is already on the wire, so a late assignment cannot take effect and is treated as a programming error rather than silently ignored — it raises (error 93.900, *the response is already committed*). This matches `error`, `redirect` and `addcookie`, which obey the same discipline; see *Commit discipline* in *The Request/Response Model*.

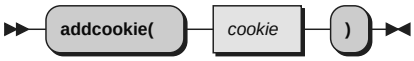
8.4.10. 404




```
.response~404
.response~404( detail )                -- detail is optional
```

Shorthand for **error(404 [, detail])** — the single most common error from a rexxlet, given its own name. Same return value and same buffering rules; see *error*.

8.4.11. addcookie




```
.response~addcookie( cookie )
```

Schedules *cookie*, which must be an **HTTP.Cookie** instance, for emission as a **Set-Cookie** header at commit time. The cookie must be internally consistent: an inconsistent one — say, **SameSite=None** without **Secure** — is not accepted, and `addcookie` raises at once. The consistency rules are given in the **HTTP.Cookie** reference.

A cookie scheduled after the response is committed could never be emitted — the head is already on the wire — so, like `[]=`, `addcookie` treats this as a programming error and raises (error 93.900) rather than dropping the cookie silently. The commit check comes first, before the argument is even type-checked: past commit the failure is *already committed*, not *not a cookie*.

The cookie is stored *by copy*: changes made to the cookie object after `addcookie` returns do not affect what will be emitted. Cookies are keyed by the triple (name, **Path**, **Domain**), matching the identity rules user agents apply: adding a cookie with the same name but a different **Path** or **Domain**

yields an additional **`Set-Cookie`** line, while re-adding one with the same triple replaces the previous one. The relative order of the **`Set-Cookie`** lines in the response is unspecified.

```
c = .HTTP.Cookie~new("session", "abc123")
c~httponly = 1
.response~addcookie(c)
```

8.4.12. commit


```
.response~commit
```

Serialises the response head: writes the **`Content-Type`** header, then one **`Set-Cookie`** line per scheduled cookie, then the remaining headers, then the blank line that separates head from body — all directly to the underlying CGI output stream, ahead of the buffered body. Commit is idempotent: the first call emits the head and marks the response committed; subsequent calls return immediately.

Rexxlets rarely call `commit` directly: the first `flush` — explicit, or the processor's final one — commits the response if needed. Once the head is committed it cannot be changed: any later attempt to set a header (`[]=` or **`response~name = value`**), schedule a cookie (`addcookie`) or replace the response wholesale (`error`, `404`, `redirect`) raises error 93.900 rather than failing silently. The body may still be written and flushed; only the head is frozen.

8.4.13. committed


```
state = .response~committed
```

Returns **`.true`** if the response has been committed (its headers have been emitted), **`.false`** otherwise. Read-only and takes no arguments. Its companion `committing` covers the brief window *during* commit, before this flag flips.

8.4.14. committing


```
state = .response~committing
```

Returns **`.true`** from the moment `commit` begins emitting the head until it finishes — the window in which some header bytes have already gone to the stream but `committed` is still **`.false`**. Read-only and takes no arguments. It lets the error handler tell a half-written response apart from an untouched one, so it never tries to re-assert headers over a head that is already partly on the wire. A rexxlet rarely needs it; `committed` is the flag of everyday use.

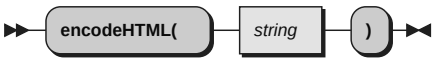
8.4.15. encodeForm


```
enc = .response~encodeForm( text )
```

```
enc = .HTTP.Response~encodeForm( text )
```

Percent-encodes *text* for an **application/x-www-form-urlencoded** body, the conforming serialisation defined by the WHATWG URL Standard. It is the twin of encodeURIComponent with the two differences the form media type mandates: the safe set is narrower — only **A**–**Z**, **a**–**z**, **0**–**9**, *****, **-**, **.**, _ stay raw, so **~** *is* escaped here though it is not by encodeURIComponent — and a space becomes **+** rather than **%20**. It is the exact inverse of the request object's decodeForm; the two are designed as a round-trip pair. This is the instance form; the body lives on the class method *encodeForm (Class Method)*.

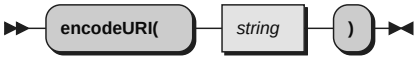
8.4.16. encodeHTML




```
safe = .response~encodeHTML( text )
safe = .HTTP.Response~encodeHTML( text )
```

Escapes the five characters that are unsafe in HTML, returning a string fit to interpolate into either element content or a quoted attribute value. The five are **&** → **&**, **<** → **<**, **>** → **>**, **"** → **"** and **'** → **'**; **&** is converted first, so an escape is never double-escaped. Escaping the two quote characters as well as the three markup characters is what makes the result safe inside an attribute, not only between tags — a value escaped for element content but interpolated into **title="..."** would otherwise let a **"** break out of the attribute. This is the method error uses for its diagnostic line. This is the instance form; the body lives on the class method *encodeHTML (Class Method)*.

8.4.17. encodeURI




```
enc = .response~encodeURI( text )
enc = .HTTP.Response~encodeURI( text )
```

Percent-encodes an *already-assembled* URL, escaping only what is illegal or unwise in a URL — control characters, space, the high-ASCII bytes, and the literal characters **< > % " { } [ ] \ ^ `** — while leaving intact the reserved characters that give a URL its structure (**/ ? # & = @ : + ; , $! ' ( ) ***). A space becomes **%20**. The contrast with encodeURIComponent is the whole point of having both: that method escapes **& / = ? #**, this one preserves them. The two carry the same names as their JavaScript namesakes and divide the work the same way — encodeURI for a complete URL, encodeURIComponent for a value going inside one. Reach for the one that matches the job; they are not interchangeable. This is the instance form; the body lives on the class method *encodeURI (Class Method)*.

8.4.18. encodeURIComponent

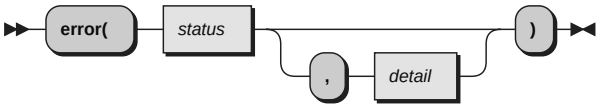

```
enc = .response~encodeURIComponent( text )
enc = .HTTP.Response~encodeURIComponent( text )
```

Percent-encodes *text* for use as a single component inside a URL — a query field value, one path segment — following RFC 3986. Every character outside the unreserved set (**A**–**Z**, **a**–**z**, **0**–**9**, **-**, **.**, _,

~) is replaced by **%** followed by its hexadecimal byte value; a space becomes **%20**. Crucially it escapes the reserved characters **& / = ? #** too, so a component carrying any of them cannot break out and alter the structure of the URL it sits in. Use it on the *pieces*, never on a whole URL — for that, see `encodeURI`. This is the instance form; the body lives on the class method *`encodeURIComponent` (Class Method)*.

```
url = "/search?q=".response~encodeURIComponent(term)
```

8.4.19. `error`




```
.response~error( status, detail )
.response~error( status )                -- detail is optional
```

Emits an error response with HTTP status *`status`* and a short HTML page, and returns **0**. *`status`* is a numeric code; *`detail`*, if given, is shown on the page as a single diagnostic line and is HTML-escaped (see the encoding methods above), so a detail containing any of **& < > " '** is displayed literally rather than interpreted.

The reason phrase is supplied automatically for the codes a rexxlet is likely to use (the **3xx** redirects, **400**, **401**, **403**, **404**, **405**, **500**, **502**, **503**). A code outside that set still works: it is given a generic phrase for its family — *Redirect*, *Client Error* or *Server Error* — so an unusual status is never refused. The lookup itself is *`statusReason`*.

`error` follows the buffering model. If the response has **not** been committed, any body the rexxlet had written so far is discarded, the status and a **`text/html`** content type are set, and the error page replaces the buffered body; the normal flush then sends it. If the response **has** been committed — the head has already reached the client, so the status can no longer be changed — `error` raises a **Syntax** condition rather than pretend to do the impossible. This mirrors the rexxlet rule for `sendError` on a committed response.

Because it returns **0**, `error` reads naturally as a rexxlet's final act:

```
If \record~exists Then Exit .response~error(404, "no such record")
```

8.4.20. `flush`


```
.response~flush
```

Commits the response if it is not yet committed, then flushes the buffered output stream, writing the body collected so far to the underlying CGI stream. Returns nothing. The processor calls it once after the rexxlet returns; a rexxlet may also call it explicitly at any point (see *The Request/Response Model* for what an early flush does — and does not — buy once **`mod_deflate`** sits downstream).

8.4.21. `init`

Rexxlets do not create responses; the processor does, once per request. A fresh response starts with exactly one header, **`Content-Type: text/plain; charset=utf-8`**; no cookies; not committed;

and a new **`HTTP.OutputStream`** wrapped around the current default output stream as its body buffer.

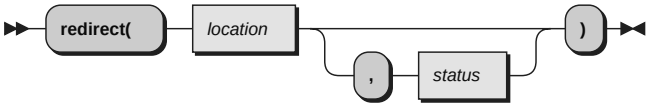
8.4.22. `output`


```
stream = .response~output
```

Returns the response's buffered output stream, an **`HTTP.OutputStream`** instance. During rexxlet execution the default output stream (**`.output`**) is redirected to this same object, which is why a plain `Say` writes to the response body; **`.response~output`** is the explicit handle, for stream methods like `~charout` or an explicit `~flush`:

```
.response~output~charout("no trailing newline here")
```

8.4.23. `redirect`




```
.response~redirect( location, status )
.response~redirect( location )          -- status defaults to 302 (Found)
```

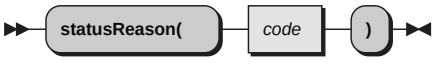
Emits a redirect to *`location`* and returns **`0`**. It sets the **`Location`** header and a **`3xx`** status; *`status`* defaults to **`302`** (*Found*), the conventional temporary redirect. No body is sent.

A different code is requested explicitly. The two that come up in practice: **`301`** (*Moved Permanently*) when a resource has acquired its canonical URL and the new address should replace the old — for example when an optional slug is added to a path — and **`303`** (*See Other*) after a form **`POST`**, so that a refresh re-fetches the result page with **`GET`** instead of resubmitting the form. Note that **`301`** is cached aggressively by user agents; use it only for a move that is genuinely permanent, never for one that might be undone.

```
Exit .response~redirect("/publishers/acme-press/", 301)
```

`redirect` obeys the same commit discipline as `error`: it discards an uncommitted buffered body and raises if the response is already committed.

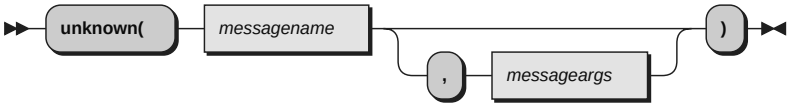
8.4.24. `statusReason`




```
phrase = .response~statusReason( code )
```

Returns the reason phrase for an HTTP status *`code`* — **`.response~statusReason(404)`** yields **`Not Found`**. A short table covers the codes the apps actually use (the **`3xx`** redirects, **`400`**, **`401`**, **`403`**, **`404`**, **`405`**, **`500`**, **`502`**, **`503`**); a code not in the table falls back to a generic phrase for its family (*Redirect*, *Client Error*, *Server Error*), so an unusual code is never rejected. This is the lookup `error` and `redirect` use to label the status they set; a rexxlet seldom calls it directly.

8.4.25. unknown



The message forms used to name a header —

```
.response~content_type = "text/html"   -- set
type = .response~content_type          -- get
```

— are not individual methods: they are resolved by `unknown`, which maps any unknown message to the header of the same name, translating underscores to hyphens (so **`content_type`** reaches **`CONTENT-TYPE`**). Unlike the request object, where only the get branch operates, *both* branches are live on the response: a message ending in **`=`** is the set branch and writes the header (subject to the commit guard — a set after commit raises error 93.900, exactly as `[]=` does); a message without **`=`** is the get branch and reads it. The set form requires exactly one value, which must have a string value; the get form accepts no arguments. Reading a header that has not been set returns **`.nil`**, exactly as with the bracket form.

Because *every* unknown message becomes a header access, the response object cannot tell a header from a typo. A misspelled getter that carries arguments fails fast (**`.response~addcokie(c)`** raises, since the get form accepts no arguments), but a misspelled *assignment* fails silently: **`.response~comitted = 1`** quietly creates — and emits — a header named **`COMITTED`**. Spelling must be watched; that is the price of the convenience.

Rationale

This chapter explains the design — not what RexxHTTP does, which the rest of the manual covers, but why it does it that way. The shape of the 1.0 is the result of a few decisions, most of them inherited from the 2006 original and kept because they are still right, a couple of them new and made by subtraction.

The largest decision is one of those subtractions, and the rest of this chapter reads more clearly once it is on the table. The 2006 RexxHTTP tried to be portable along three axes at once: it ran under both CGI and mod_rexx, on OS/2 and Windows as well as Unix, and aimed at Microsoft IIS beside Apache. Much of the original rationale was the bookkeeping of holding those axes together — reconciling the variables CGI provided with the ones mod_rexx provided, working around an OS/2 Object Rexx that lacked a **`MutableBuffer`**, and so on. The 1.0 keeps one point in that space: CGI, on Apache, on a current ooRexx. mod_rexx is no longer maintained upstream, IIS and OS/2 are not targets anyone is asking for, and committing to a single gateway is what lets the rest of the design be simple. Several of the decisions below were originally compromises between two gateways; with only one gateway left, they reduce to a single clear answer, and this chapter states the answer rather than the old negotiation.

That wider original is not lost: it is described, in full, by the manual that shipped with it — Josep Maria Blasco, *RexxHttp: Servlet programming in Rexx*, version 0.1, 2 November 2006, available at *https://www.epbcn.com/pdf/josep-maria-blasco/2006-11-02-RexxHttp-Servlet-programming-in-Rexx.pdf*. It remains the record of the portable, multi-gateway design; this manual documents the narrower 1.0 that grew out of it, and the decisions below are the ones that narrowing settled.

9.1. Associating Rexx Files with a Handler

A rexxlet is an ordinary ooRexx file, and the web server has to be told to run it through RexxHTTP rather than serve it or execute it directly. Apache's **`Action`** directive is the mechanism: it binds a handler name to the RexxHTTP processor, and a small wrapper invokes that handler for the rexxlet files it is given. This is the same approach the 2006 manual described, and it has aged well precisely because it asks nothing unusual of Apache — **`Action`** is a stock directive, and the routing is data in the configuration rather than logic in the processor.

What has changed is the disposition. In 2006 the natural move was to map broad extensions — **`.htm`**, **`.html`**, **`.rsp`** — to the handler, because the point was to run whole trees of pages, some of them compiled, through one gateway. The 1.0 has no page compilers and no such ambition, and the *Installation* chapter recommends the opposite default: a whitelist, so that only the files explicitly marked are run as rexxlets and everything else is served as a static file. On infrastructure that has served tens of thousands of URLs for years, “run as code only what is named” is the conservative and correct default; opening an extension to the handler is the exception, made deliberately.

9.2. The Environment-Variable Request Model

CGI delivers a request as a set of environment variables, and RexxHTTP takes that literally: the request object answers any message it does not recognise with the value of the like-named environment variable. This is the single most Rexx-like decision in the design. ooRexx programs reach their world through **`Value`** and through variables; making **`request~remote_addr`** resolve to the **`REMOTE_ADDR`** the server set means a rexxlet author touches the CGI environment the same way they touch everything else, without a lookup table or an import step. The mechanics are in the Class Reference under *The Environment Pool*; the reason is that the request object should feel like Rexx, not like a wrapper bolted over Rexx.

This convenience raises one question. If **`request~anything`** can name an environment variable, then a variable could share a name with a method — present or future — and the author needs a predictable answer to which one wins. The 2006 manual settled it by carving out a namespace: operator-set variables had to begin with an underscore, where no method would ever live. It worked, but it asked the author to remember a naming rule, and the variables that motivated it — timezone hints for the old Netscape-style cookies, settings for the page compilers — are all gone from the 1.0. So the rule went with them, in favour of something plainer: two doors instead of one reserved name. **`request~name`** is the library's view, where a method wins when one exists, exactly as ooRexx method lookup already works; **`request["name"]`** is the raw variable underneath, which always reaches the environment pool and never a method. An operator-set variable is then just an ordinary variable — **`SetEnv MYAPP_MODE production`** is **`request~myapp_mode`**, no underscore required — and if a future method ever shadowed it, the bracket form reaches it regardless. The collision question is answered not by reserving names but by giving the author a way to ask for either side of one.

9.3. The Problem of Request Values

The trickiest part of the 2006 rationale was titled “the problem of request values,” and the problem was real: what Apache actually invokes is the RexxHTTP processor, not the rexxlet, so the raw CGI variables describe the processor's invocation, not the request the rexxlet means to read. On top of that, the CGI specification is in places ambiguous or self-contradictory, and the 2006 system had to paper over differences between what CGI reported and what mod_rexx reported for the same notion.

With mod_rexx gone, the second half of that problem evaporates: there is one gateway, so there is nothing to reconcile. What remains is the first half — reconstructing the values a rexxlet *should* see from the values the processor is *handed* — and a commitment the 2006 design made and the 1.0 keeps: where CGI is inconsistent, RexxHTTP is consistent, and it documents what its methods mean rather than passing the server's quirks through. Three cases from the original are worth keeping, because the methods still behave as described and the reasoning is still the cleanest account of why:

`path_translated` always answers the full filesystem path of the rexxlet file, whether or not there is extra path information after it. The CGI specification's own definition of this variable is incoherent — it promises a “translated version” of the extra path info, which by construction has no physical mapping — and Apache, left to itself, returns something useless. RexxHTTP picks the meaning that is actually useful and holds to it.

`script_name` always answers the virtual path that locates the rexxlet, with any extra path information stripped — what is needed for building self-referential URLs — and `path_info` answers that extra path, decoded, or the null string when there is none. The split between the two is clean by design, so a rexxlet can always ask “where am I?” and “what came after me?” as separate questions.

`request_uri` (with its alias `unparseduri`) answers the request URI exactly as the client sent it, still encoded; `uri` answers the decoded path with the query string removed. Naming both, with stable meanings, spares the rexxlet author from guessing which encoding and which trimming a given server happened to apply.

The thread through all three is the same: the request object is a *specification* the rexxlet can rely on, not a thin pass-through of whatever the gateway felt like reporting.

9.4. Strict by Default

The 1.0 parses GET and POST arguments under a single, default, and currently only argument policy, **`strict`**: every parameter must be a genuine *`name=value`* pair, the name must be a usable ooRexx symbol and must be unique caselessly, and a request that breaks any of these is answered **`400 Bad`**

`Request` before the rexxlet runs at all. The mechanics are in *Arguments, and the Strict Policy*; the decision is to make this the default rather than an opt-in.

The reasoning is specific to the kind of service RexxHTTP exists to run. For a public site with a large, stable set of URLs — the case RexxHTTP was extracted from — malformed argument input is not a case to be handled gracefully; it is a request that cannot have come from the site's own pages, and the safe, predictable response is to reject it at the door. Strict-by-default turns a class of ambiguous input into a clear **`400`**, and in exchange every rexxlet downstream gets to assume that each argument it sees has a name and a value, and that a name names exactly one argument. The strictness is a feature paid for once, at the boundary, so that no rexxlet has to pay for it again.

Appendix A. License

RexxHTTP is free software, released under the Apache License, Version 2.0.

> Copyright 2006–2026 Josep Maria Blasco
>
> Licensed under the Apache License, Version 2.0 (the “License”); you may not use this software except in compliance with the License. You may obtain a copy of the License at
>
> > *http://www.apache.org/licenses/LICENSE-2.0*
>
> Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The complete text of the License follows, and also accompanies the distribution in the file **`LICENSE`** at the root of the repository.

A.1. The Apache License, Version 2.0

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other

modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W