Zinc: an industrial quality level HTTP/S framework

Sven van Caekenberghe

March 12, 2024

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

You have in your hand a great documentation of an important and super nice library of Pharo. Zinc has been developed by Sven van Caekenberghe over the years with the constant attention to excellent design, specifications and speed. Zinc is a cornerstone of Pharo and as a community, we all thank Sven for his dedication.

The contents of this booklet have been extracted from the Pharo book A web Perspective which you can find at http://books.pharo.org. You may wonder why. The reason is simple, maintaining up to date a book of multiple chapters and authors is a daunting task. Maintaining some focused and beautiful documentation is a lot easier. So as an editor of the Pharo Technology Collection, I decided that it was important to get agile and make sure that such lovely content can evolve gracefully.

20 March 2020. S. Ducasse

CHAPTER 1

Character Encoding and Resource Meta Description

The rise of the Internet and of Open Standards resulted in the adoption of a number of fundamental mechanisms to enable communication and collaboration between different systems.

One such mechanism is the ability to encode strings or characters to bytes or to decode strings or characters from bytes. Different encoding standards have been developed over the years and Pharo supports many current and legacy encodings.

Another important aspect of collaboration is the ability to describe resources such as files. Both Mime-Type and URLs or URIs are basic building blocks for creating meta descriptions of resources and Pharo also has objects that implement these fundamental aspects.

In this chapter we discuss Character encoding, MIME types and URL/URIs. They are essential for the correct implementation of HTTP, but they are independent from it, as they are used for many other purposes.

1.1 Character Encoding

We will first show how to get Unicode from characters and strings within Pharo. We will then show how to decode and encode characters and strings from and to bytes.

1.2 Characters and Strings use Unicode Internally

Proper character encoding and decoding is crucial in today's international world. Internally, Pharo stores characters and strings using Unicode.Unicodehttp://en.wikipedia.org/wiki/Unicode is a very large internationally standardized collection of code points (integer numbers) representing all of the world languages' characters.

We can obtain the code point (Unicode value) of a character by sending it the `codePoint` message, for example:

```
$H codePoint
>>> 72
```

Here are some example strings in multiple languages with their Unicode code points:

```
'Hello' collect: #codePoint as: Array.
>>> #(72 101 108 108 111)
```

```
'Les élèves français' collect: #codePoint as: Array.
>>> #(76 101 115 32 233 108 232 118 101 115
        32 102 114 97 110 231 97 105 115)
```

```
'Ελλάδα' collect: #codePoint as: Array.
>>> #(917 955 955 940 948 945)
```

For a simple language like English, all characters have code points below 128 (which fits in 7 bits, for historical reasons). These characters are part of ASCIIhttp://en.wikipedia.org/wiki/ASCII. The very first part of the so called Basic Multilingual Plane of Unicode (the first 128 code points of it) are identical to ASCII.

```
$a codePoint
>>> 97
```

Next come a number of European languages, like French, which have code points below 256 (fitting in 8 bits or one byte). These characters are part of Latin-1 (ISO-8859-1)http://en.wikipedia.org/wiki/ISO/IEC_8859-1, whose first 256 code points are identical in Unicode.

```
$é codePoint
>>> 233
```

And finally, there are hundreds of other languages, like Chinese, Japanese, Cyrillic, Arabic or Greek. You can see from the example above: Greece written in Greek, that those code points are higher than 256 (and thus no longer fit in one byte).

```
λ
$ codePoint
>>> 955
```

Unicode code points are often written using a specific hexadecimal notation. For example, the previous character, the Greek lowercase lambda, is written as`U+03BB`. The Pharo inspector also shows this value next to the codepoint.

The good thing is, we can work with text in any language in Pharo. However, to display everything correctly a font must be used that is capable of showing all the characters (or glyphs) needed, for example Arial Unicode MS.

1.3 Encoding and Decoding

For communication with the world outside Pharo, the operating system, files, the internet, et cetera, we have to represent our strings as a collection of bytes. Yet code points are different to bytes, as will be shown below. Therefore we need a way to transform our internal strings into external collection of bytes and vice versa.

Character encoding is the standard way of converting a native Pharo string, i.e. a collection of Unicode code points, to a series of bytes. Character decoding is the reverse process: interpreting a series of bytes as a collection of Unicode code points, to create a Pharo string.

To implement character encoding or decoding, a concrete subclass of the class `ZnCharacterEncoder` is used, e.g. `ZnUTF8Encoder`. Character encoders do the following:

- encode a character (message `nextPut:toStream:`) or string (message `next:putAll:startingAt:toStream:`) onto a binary stream
- convert a string (`encodeString:`) to a byte array
- decode a binary stream to a character (`nextFromStream:`) or string (`readInto:startingAt:count:from`
- convert a byte array to string (`decodeBytes:`)
- compute the number of bytes that are needed to encode a character (`encodedByteCountFor:`) or string (`encodedByteCountForString:`)
- move a binary stream backwards one character (`backOnStream:`)

Character encoders do proper error handling, throwing an error of the class `ZnCharacterEncodingError` when something goes wrong. The strict/lenient setting controls some behavior in this respect, and this will be discussed later in this chapter.

The recommended encoding is the primary internet encoding: UTF-8http://en.wikipedia.org/wiki/UTF-8. It is a variable length encoding that is optimized somewhat for ASCII and to a lesser degree for Latin1 and some other common European encodings.

1.4 Converting Strings and ByteArrays

The first use of encoders is to convert Strings to ByteArrays and vice-versa. We however deal only indirectly with character encoders. The `ByteArray` and `String` classes have some convenience methods to do encoding and decoding:

```
'Hello' utf8Encoded.
>>> #[72 101 108 108 111]

'Hello' encodeWith: #latin1.
>>> #[72 101 108 108 111]
```

Our ASCII string, `'Hello'` encodes identically using either UTF-8 or Latin-1.

```
'Les élèves français' utf8Encoded.
>>> #[76 101 115 32 195 169 108 195 168 118 101 115
        32 102 114 97 110 195 167 97 105 115]

'Les élèves français' encodeWith: #latin1.
>>> #[76 101 115 32 233 108 232 118 101 115
       32 102 114 97 110 231 97 105 115]
```

Our French string, `'Les élèves français'`, encodes differently though. The reason is that UTF-8 uses two bytes for the accented letters like é, è and ç. Note how for Latin-1, and **only** for Latin-1 and ASCII, the Unicode code points are equal to the encoded byte values.

```
'éèç' utf8Encoded.
>>> #[195 169 195 168 195 167]

'éèç' encodeWith: #latin1.
>>> #[233 232 231]

'éèç' collect: #codePoint as: ByteArray
>>> #[233 232 231]
```

```
'Ελλάδα' utf8Encoded.
>>> #[206 149 206 187 206 187 206 172 206 180 206 177]

'Ελλάδα' encodeWith: #latin1.
>>> ZnCharacterEncodingError: 'Character Unicode code point outside
    encoder range'
```

Our greek string, `'Ελλάδα'`, gives an error when we try to encode it using Latin-1. The reason is that the Greek letters are outside of the alphabet of Latin-1. Still, UTF-8 manages to encode them using just two bytes.

The reverse process, decoding, is equally simple:

```
#[72 101 108 108 111] utf8Decoded.
>>> 'Hello'

#[72 101 108 108 111] decodeWith: #latin1.
>>> 'Hello'
```

```
#[76 101 115 32 195 169 108 195 168 118 101 115
  32 102 114 97 110 195 167 97 105 115] utf8Decoded.
>>> 'Les élèves français'

#[76 101 115 32 195 169 108 195 168 118 101 115
  32 102 114 97 110 195 167 97 105 115] decodeWith: #latin1.
>>> 'Les Ã©lÃ¨ves franÃ§ais'

#[76 101 115 32 233 108 232 118 101 115
  32 102 114 97 110 231 97 105 115] utf8Decoded.
>>> ZnInvalidUTF8: 'Illegal continuation byte for utf-8 encoding'
```

```
#[76 101 115 32 233 108 232 118 101 115
  32 102 114 97 110 231 97 105 115] decodeWith: #latin1.
>>> 'Les élèves français'

#[206 149 206 187 206 187 206 172 206 180 206 177] utf8Decoded.
>>> 'Ελλάδα'

#[206 149 206 187 206 187 206 172 206 180 206 177] decodeWith:
    #latin1.
>>> ZnCharacterEncodingError: 'Character Unicode code point outside
```

```
    encoder range'
```

Our English `'Hello'`, being pure ASCII, can be decoded using either UTF-8 or Latin-1. Our French `'Les élèves français'` is another story: using the wrong encoding gives either gibberish or `ZnInvalidUTF8` error. The same is true for our Greek `'Ελλάδα'`.

You might wonder why in the first case the `latin1` encoder produced gibberish, while in the second case it gave an error. This is because in the second case, there was a byte with value 149, which is outside its alphabet. So called byte encoders, like Latin-1, take a subset of Unicode characters and compress them in 256 possible byte values. This can be seen by inspecting the character or byte domains of a `ZnByteEncoder`, as follows:

```
(ZnByteEncoder newForEncoding: 'iso-8859-1') byteDomain.
(ZnByteEncoder newForEncoding: 'ISO_8859_7') characterDomain.
```

Note that identifiers for encodings are interpreted flexibly (case and punctuation do not matter).

There exists a special `ZnNullEncoder` that basically does nothing: it treats bytes are characters and vice versa. This is actually mostly equivalent to Latin-1 or ISO-8859-1. (And yes, that is a bit confusing.)

1.5 Converting Streams

The second primary use of encoders is when dealing with streams. More specifically, when interpreting a binary read or write stream as a character stream. Note that at their lowest level, all streams to and from the operating system or network are binary and thus need the use of an encoder when treating them as character streams.

To treat a binary write stream as a character write stream, wrap it with a `ZnCharacterWriteStream`. Similary, `ZnCharacterReadStream` should be used to treat a binary read stream as a character stream. Here is an example:

```
'encoding-test.txt' asFileReference writeStreamDo: [ :out |
   (ZnCharacterWriteStream on: out binary encoding: #utf8)
      nextPutAll: 'Hello'; space; nextPutAll: 'Ελλάδα'; crlf;
      nextPutAll: 'Les élèves français'; crlf ].

'encoding-test.txt' asFileReference readStreamDo: [ :in |
   (ZnCharacterReadStream on: in binary encoding: #utf8)
      upToEnd ]

>>> 'Hello Ελλάδα
Les élèves français
'
```

We used the message `on:encoding:` here, but there is also a plain message `on:` instance creation message that defaults to the UTF-8 encoding. Internally, the character streams will use an encoder instance to do the actual work.

1.6 ByteStrings and WideStrings are Concrete Subclasses of String

Up until now we spoke about Strings as being a collection of Characters, each of which is represented as a Unicode code point. And this is conceptually totally how they should be thought about. However, in reality, the class `String` is an abstract class with two concrete subclasses. This will show up when inspecting `String` instances, so it is important to understand what is going on. Consider the following example strings:

```
'Hello' class
>>> ByteString
```

```
'Les élèves français' class
>>> ByteString
```

```
'Ελλάδα' class
>>> WideString
```

Simple ASCII strings are ByteStrings. Strings using special characters may be WideStrings or may still be ByteStrings. The explanation of the use of the `WideString` or `ByteString` class is very simple when considering the Unicode code points used for each character.

In the first case, for ASCII, the code points are always less than 128. Hence they fit in one byte. The second string is using Latin-1 characters, whose code points are less than 256. These still fit in a byte. A `ByteString` is a `String` that only stores Unicode code points that fit in a byte, in an implementation that is very efficient. Note that `ByteString` is a variable byte subclass of `String`.

Our last example has code points that no longer fit in a byte. To be able to store these, `WideString` allocates 32-bit (4 byte) slots for each character. This implementation is necessarily less efficient. Note that `WideString` is a variable word subclass of `String`.

In practice, the difference between `ByteString` and `WideString` should not matter. Conversions are done automatically when needed.

```
'abc' copy at: 1 put: α$; class.
>>> WideString
```

As the above example shows, in a `ByteString` `'abc'` putting the Unicode character $α, converts it to a `WideString`. (This is actually done using a `becomeForward:` message.) When benchmarking, this conversion might show up as taking significant time. If you know upfront that you will need WideStrings, it can be better to start with the right type.

1.7 ByteString and ByteArray Equivalence is an Implementation Detail

There is another implementation detail worth mentioning: for the Pharo virtual machine, more specifically, for a number of primitives, `ByteString` and `ByteArray` instances are equivalent. Given what we now know, that makes sense. Consider the following code:

```
'abcdef' asByteArray.
>>> #[97 98 99 100 101 102]

'ABC' asByteArray.
>>> #[65 66 67]
```

```
'abcdef' copy replaceFrom: 1 to: 3 with: #[65 66 67].
>>> 'ABCdef'

#[97 98 99 100 101 102] copy replaceFrom: 1 to: 3 with: 'ABC'.
>>> #[65 66 67 100 101 102]
```

In the third expression, we send the message `replaceFrom:to:with:` on a `ByteString`, but give a `ByteArray` as third argument. So we are replacing part of a `ByteString` with a `ByteArray`. And it works!

The last example goes the other way around: we replace part of a `ByteArray` with a `ByteString`, which works as well.

What about doing the same mix up with elements ?

```
'abc' copy at: 1 put: 65; yourself.
>>> Error: improper store into indexable object

#[97 98 99] copy at: 1 put: $A; yourself.
>>> Error: improper store into indexable object
```

This is more what we expect: we're not allowed to do this. We are mixing two types that are not equivalent, like `Character` and `Integer`.

So although it is true that there is some equivalence between `ByteString` and `ByteArray`, you should not mix up the two. It is an implementation detail that you should not rely upon.

1.8 Beware of Bogus Conversions

Given a string, it is tempting to send it the message `asByteArray` to convert it to bytes. Similary, it is tempting to convert a byte array by sending it the message `asString`. These are however bogus conversions that should not be used as for some strings they will work, but for others not. Success depends on the code points of the characters in the string. Basically the conversion is possible for strings for which the following property holds:

```
'Hello' allSatisfy: [ :each | each codePoint < 256 ].
>>> true

'Les élèves français' allSatisfy: [ :each | each codePoint < 256 ].
>>> true

'Ελλάδα' allSatisfy: [ :each | each codePoint < 256 ].
>>> false
```

Now, even though the first two can be converted, they will not be using the same encoding. Here is a way to explicitly express this idea:

```
#(null ascii latin1 utf8) allSatisfy: [ :each |
  ('Hello' encodeWith: each) = 'Hello' asByteArray ].
>>> true.

('Les élèves français' encodeWith: #latin1) = 'Les élèves français'
    asByteArray.
>>> true.
```

```
('Les élèves français' encodeWith: #null) = 'Les élèves français'
    asByteArray.
>>> true.

'Les élèves français' utf8Encoded = 'Les élèves français'
    asByteArray.
>>> false.
```

For pure ASCII strings, with all code points below 128, no encoding (null encoding), ASCII, Latin-1 and UTF-8 are all the same. For other `ByteString` instances, like `'Les élèves français'`, only Latin-1 works. In that case it is also equivalent of doing no encoding.

The lazy conversion for proper Unicode WideStrings will give unexpected results:

```
'Ελλάδα' asByteArray.
>>> #[0 0 3 149 0 0 3 187 0 0 3 187 0 0 3 172 0 0 3 180 0 0 3 177]
```

This 'conversion' does not correspond to any known encoding. It is the result of writing 4-byte Unicode code points as Integers. Using this is a bug no matter how you look at it. In this century you will look silly for not implementing proper support for all languages. When converting between strings and bytes, use a proper, explicit encoding.

1.9 Strict and Lenient Encoding

No encoding (or the null encoder) and Latin-1 encoding are in fact not completely the same. This is because there are 'holes' in the table: some byte values are undefined, which a strict encoder won't allow, and the default encoder is strict.

For example, the Unicode code point 150 is strictly speaking not in Latin-1:

```
ZnByteEncoder latin1 encodeString: 150 asCharacter asString.
>>> ZnCharacterEncodingError: 'Character Unicode code point outside
    encoder range'

ZnByteEncoder latin1 decodeBytes: #[ 150 ].
>>> ZnCharacterEncodingError: 'Character Unicode code point outside
    encoder range'
```

The encoder can however be instructed to `beLenient`, which will produce a silent conversion (if that is possible). In this case, Unicode character 150 (`U+0096`) is an unprintable control character meaning 'Start of Protected Area' (SPA) and is strictly speaking not part of Latin-1.

```
ZnByteEncoder latin1 beLenient encodeString: 150 asCharacter
    asString.
>>> #[ 150 ]

ZnByteEncoder latin1 beLenient decodeBytes: #[ 150 ].
>>> ''
```

You can explicity access both the allowed byte or character values, i.e. the domain of encoder or decoder:

```
ZnByteEncoder latin1 characterDomain includes: 150 asCharacter.
>>> false

ZnByteEncoder latin1 byteDomain includes: 150.
>>> false
```

Note that the lower half of a byte encoding, the ASCII part between 0 and 127, is always treated as a one to one mapping.

1.10 Available Encoders

Pharo comes with support for the most important encodings currently used, as well as with support for some important legacy encodings. Seen as the classes implementing them, the following encoders are available:

- `ZnUTF8Encoder`
- `ZnUTF16Encoder`
- `ZnByteEncoder`
- `ZnNullEncoder`

Where `ZnByteEncoder` groups a large number of encodings. This list is available as `ZnByteEncoder knownEncodingIdentifiers`. Here is a list of all recognized, canonical names: arabic, cp1250, cp1251, cp1252, cp1253, cp1254, cp1255, cp1256, cp1257, cp1258, cp850, cp866, cp874, cyrillic, dos874, doslatin1, greek, hebrew, ibm819, ibm850, ibm866, iso885910, iso885911, iso885913, iso885914, iso885915, iso885916, iso88592, iso88593, iso88594, iso88595, iso88596, iso88597, iso88598, iso88599, koi8, koi8r, koi8u, latin2, latin3, latin4, latin5, latin6, mac, maccyrillic, macintosh, macroman, oem850, windows1250, windows1251, windows1252, windows1253, windows1254, windows1255, windows1256, windows1257, windows1258, windows874, xcp1250, xcp1251, xcp1252, xcp1253, xcp1254, xcp1255, xcp1256, xcp1257, xcp1258, xmaccyrillic and xmacroman.

1.11 Mime-Types

A mime-type is a standard, cross-platform definition of a file or document type or format. The official term is an Internet media typehttp://en.wikipedia.org/wiki/Internet_media_type*media*type).

Mime-types are modeled using `ZnMimeType` objects, which have 3 components:

1. a main type, for example `text` or `image`,
2. a sub type, for example `plain` or `html`, or `jpeg`, `png` or `gif`, and
3. a number of attributes, for example `charset=utf-8`.

The mime-type syntax is as follows:

`<main>/<sub> [;<param1>=<value1>[,<param2>=<value2>]*]`.

Creating Mime-Types

Instances of `ZnMimeType` are created by explicitly specifying its components, through parsing a string or by accessing predefined values. In any case, a new instance is always created.

The class side of `ZnMimeType` has some convenience methods (in the protocol `convenience`) for accessing well known mime-types, which is the recommended way for obtaining these mime-types:

```
ZnMimeType textHtml
>>> text/plain;charset=utf-8

ZnMimeType imagePng
>>> image/png
```

Here is an example of how to create a mime-type by explicitly specifying its components:

```
ZnMimeType main: 'image' sub: 'png'
>>> image/png
```

The main parsing interface of `ZnMimeType` is the class side `fromString:` message.

```
ZnMimeType fromString: 'image/png'
>>> image/png
```

To make it easier to write code that accepts both instances and strings, the `asZnMimeType` message can be used:

```
'image/png' asZnMimeType
>>> image/png

ZnMimeType imagePng asZnMimeType = 'image/png' asZnMimeType
>>> true
```

Finally, `ZnMimeType` also knows how to convert file name extensions to mime-types using the `forFilenameExtension:` message. This mapping is based on the Debian/Ubuntu `/etc/mime.types` file, which is encoded into the method `mimeTypeFilenameExtensionsSpec`.

```
ZnMimeType forFilenameExtension: 'html'.
>>> text/html;charset=utf-8
```

In most applications, the concept of a default mime-type exists. It basically means: we don't know what these bytes represent.

```
ZnMimeType default
>>> application/octet-stream
```

1.12 Working with Mime-Types

Once you have a ZnMimeType instance, you can access its components using the `main`, `sub` and `parameters` messages.

An important aspect of mime-types is whether the type is textual or binary, which is testable with the `isBinary` message. Typically, text, XML or JSON are considered textual, while images are binary.

For textual (non-binary) types, the encoding (or charset parameter) defaults to UTF-8, the prevalent internet standard. With the convencience messages`charSet:`, `setCharSetUTF8` and `clearCharSet` you can manipulate the charset parameter.

Comparing mime-types using the standard `=` message takes all components into account, including the parameters. Different parameters lead to different mime-types. As a result, when charsets are involved it is often better to compare using the `matches:` message, as follows:

```
'text/plain' asZnMimeType = ZnMimeType textPlain.
>>> false

ZnMimeType textPlain = 'text/plain' asZnMimeType.
>>> false

'text/plain' asZnMimeType matches: ZnMimeType textPlain.
>>> true

ZnMimeType textPlain matches: 'text/plain' asZnMimeType.
>>> true
```

The charset=UTF-8 that is part of what `ZnMimeType textPlain` returns is not taken into account in the second set of comparisons.

The main or sub types can be a wildcard, indicated by a `*`. This allows for matching. Obviously, everything matches `*/*` (`ZnMimeType any`). Otherwise, when the sub type is `*`, the main types must be equal. Here is an example.

```
ZnMimeType text.
>>> text/*

ZnMimeType textHtml matches: ZnMimeType text.
>>> true

ZnMimeType textPlain matches: ZnMimeType text.
>>> true

ZnMimeType applicationXml matches: ZnMimeType text.
>>> false
```

1.13 URLs

URLs (or URIs) are a way to name or identify an entity. Often, they also contain information of where the entity they name or identify can be accessed.

We will be using the terms URL (Uniform Resource Locatorhttp://en.wikipedia.org/wiki/Uniform_resource_locator*resource*locator)) and URI (Uniform Resource Identifierhttp://en.wikipedia.org/wiki/Uniform_resource_identifier*resource*identifier)) interchangeably, as is most commonly done in practice. A URI is just a name or identification, while a URL also contains information on how to find or access a resource. Consider the following example: the URI `/documents/cv.html` identifies and names a document, while the URL `http://john-doe.com/documents/cv.html` also specifies that we can use HTTP to access this resource on a specific server.

By considering most parts of an URL as optional, we can use one abstraction to implement both URI and URL using one class. The class `ZnUrl` models URLs (or URIs) and has the following components:

1. scheme - like `#http`, `#https` , `#ws`, `#wws`, `#file` or `nil`
2. host - hostname string or `nil`
3. port - port integer or `nil`
4. segments - collection of path segments, ends with `#/` for directories
5. query - query dictionary or `nil`

6. fragment - fragment string or `nil`
7. username - username string or `nil`
8. password - password string or `nil`

The syntax of the external representation of a ZnUrl informally looks like this:`scheme://username:password@host:`

1.14 Creating URLs

ZnUrls are most often created by parsing an external representation using either the `fromString:` class message or by sending the `asUrl` or `asZnUrl` convenience message to a string.

```
ZnUrl fromString: 'http://www.google.com/search?q=Smalltalk'.
'http://www.google.com/search?q=Smalltalk' asUrl.
```

The same instance can also be constructed programmatically:

```
ZnUrl new
   scheme: #http;
   host: 'www.google.com';
   addPathSegment: 'search';
   queryAt: 'q' put: 'Smalltalk';
   yourself.
```

`ZnUrl` components can be manipulated destructively. Here is an example:

```
'http://www.google.com/?one=1&two=2' asZnUrl
   queryAt: 'three' put: '3';
   queryRemoveKey: 'one';
   yourself.

>>> http://www.google.com/?two=2&three=3
```

1.15 External and Internal Representation of URLs

Some characters of parts of a URL are considered as illegal because including them would interfere with the syntax and further processing. They thus have to be encoded. The methods of `ZnUrl` in the`accessing` protocols do not do any encoding, while those in `parsing` and `printing` do. Here is an example:

```
'http://www.google.com'
   addPathSegment: 'an encoding';
   queryAt: 'and more' put: 'here, too';
   yourself
>>> http://www.google.com/an%20encoding?and%20more=here,%20too
```

The ZnUrl parser is somewhat forgiving and accepts some unencoded URLs as well, like most browsers would.

```
'http://www.example.com:8888/a path?q=a, b, c' asZnUrl.
>>> http://www.example.com:8888/a%20path?q=a,%20b,%20c
```

1.16 Relative URLs

ZnUrl can parse in the context of a default scheme, like a browser would do.

```
ZnUrl fromString: 'www.example.com' defaultScheme: #http
>>> http://www.example.com/
```

Given a known scheme, ZnUrl knows its default port, and this is accessed by `portOrDefault`.

A path defaults to what is commonly referred to as slash, which is testable with `isSlash`. Paths are most often (but don't have to be) interpreted as filesystem paths. To support this, the `isFilePath` and `isDirectoryPath` tests and `file` and `directory` accessors are provided.

ZnUrl has some support to handle one URL in the context of another one, this is also known as a relative URL in the context of an absolute URL. This is implemented using the `isAbsolute`, `isRelative` and `inContextOf:` methods. For example:

```
'/folder/file.txt' asZnUrl inContextOf:
    'http://fileserver.example.net:4400' asZnUrl.
>>> http://fileserver.example.net:4400/folder/file.txt
```

1.17 Operations on URLs

To add operations to URLs you could add an extension method to the ZnUrl class. In many cases though, it will not work on all kinds of URLs but only on a subset. In other words, you need to dispatch, not just on the scheme but maybe even on other URL elements. That is where `ZnUrlOperation` comes in.

The first step for its use is defining a name for the operation. For example, the symbol `#retrieveContents`. Second, one or more subclasses of`ZnUrlOperation` need to be defined, each defining the class side message `operation` to return the name, `#retrieveContents` in the example. Then all subclasses with the same operation form the group of applicable implementations. Third, these handler subclasses overwrite `performOperation` to do the actual work.

Given a ZnUrl instance, sending the message `performOperation:` or`performOperation:with:` will send the message `performOperation:with:on:` to `ZnUrlOperation`. In turn, it will look for an applicable handler subclass, instanciate and invoke it.

Each subclass will be sent `handlesOperation:with:on:` to test if it can handle the named operation with an optional argument on a specific URL. The default implementation already covers the most common case: the operation name has to match and the scheme of the URL has to be part of the collection returned by `schemes`.

For our example, the message `retrieveContents` on ZnUrl is implemented as an operation named `#retrieveContents`. The handler class is either the class `ZnHttpRetrieveContents` for the schemes `http` and `https` or the class `ZnFileRetrieveContents` for the scheme `file`.

This dispatching mechanism is more powerful than scheme specific `ZnUrl` subclasses because other elements can be taken into account. It also addresses another issue with scheme specific `ZnUrl` subclasses, which is that there are an infinite number of schemes which no hierarchy could cover.

1.18 Odds and Ends

Sometimes, the combination of a host and port are referred to as authority, and this is accessable with the `authority` message.

There are convenience methods to download the resource a ZnUrl points to: `retrieveContents` and `saveContentsToFile`. The first retrieves the contents and returns it directly, while the expression saves the contents directly to a file.

```
'http://zn.stfx.eu/zn/numbers.txt' asZnUrl retrieveContents.
'http://zn.stfx.eu/zn/numbers.txt' asZnUrl saveContentsToFile:
    'numbers.txt'.
```

ZnUrl can be used to handle file URLs. Use `isFile` to test for this scheme.

Given a file URL, it can be converted to a regular `FileReference` using the `asFileReference` message. In the other direction, you can get a file URL from a`FileReference` using the `asUrl` or `asZnUrl` messages. Do keep in mind that there is no such thing as a relative file URL, only absolute file URLs exist.

CHAPTER 2

Zinc HTTP: The Client Side

HTTP is arguably the most important application level network protocol for what we consider to be the Internet. It is the protocol that allows web browsers and web servers to communicate. It is also becoming the most popular protocol for implementing web services.

With Zinc, Pharo has out of the box support for HTTP. Zinc is a robust, fast and elegant HTTP client and server library written and maintained by Sven van Caekenberghe.

2.1 HTTP and Zinc

HTTP, short for Hypertext Transfer Protocol, functions as a request-response protocol in the client-server computing model. As an application level protocol it is layered on top of a reliable transport such as a TCP socket stream. The most important standard specification document describing HTTP version 1.1 is RFC 2616http://tools.ietf.org/html/rfc2616. As usual, a good starting point for learning about HTTP is its Wikipedia articlehttp://en.wikipedia.org/wiki/Http.

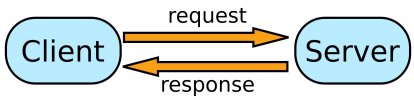

Figure 2-1 Client/Server interacting via request/response

A client, often called user-agent, submits an HTTP request to a server which will respond with an HTTP response (see Fig. 2-1). The initiative of the communication lies with the client. In HTTP parlance, the client requests a resource. A resource, sometimes also called an entity, is the combination of a collection of bytes and a mime-type. A simple text resource will consist of bytes encoding the string in some encoding, for example UTF-8, and the mime-type`text/plain;charset=utf-8`, in contrast, an HTML resource will have a mime-type like `text/html;charset=utf-8`.

To specify which resource you want, a URL (Uniform Resource Locator) is used. Web addresses are the most common form of URL. Consider for examplehttp://pharo.org/files/pharo-logo-small.png : it is a URL that refers to a PNG image resource on a specific server.

The reliable transport connection between an HTTP client and server is used bidirectionally: both to send the request as well as to receive the response. It can be used for just one request/response cycle, as was the case for HTTP version 1.0, or it can be reused for multiple request/response cycles, as is the default for HTTP version 1.1.

Zinc, the short form for Zinc HTTPhttp://zn.stfx.eu/ Components, is an open-source framework to deal with HTTP. It models most concepts of HTTP and its related standards and offers both client and server functionality. One of its key goals is to offer understandability (Pharo's design principle number one). Anyone with a basic understanding of Pharo and the HTTP principles should be able to understand what is going on and learn, by looking at the implementation. Zinc, or Zn, after its namespace prefix, is an integral part of Pharo since version 1.3. It has been ported to other systems such as Gemstone.

The reference Zn implementation lives in several places:

- http://www.squeaksource.com/ZincHTTPComponents
- http://mc.stfx.eu/ZincHTTPComponents
- https://www.github.com/svenvc/zinc

Installation or updating instructions can be found on its web sitehttp://zn.stfx.eu/.

2.2 Doing a Simple Request

The key object to programmatically execute HTTP requests is called `ZnClient`. You instantiate it, use its rich API to configure and execute an HTTP request and access the response. `ZnClient` is a stateful object that acts as a builder.

Basic Usage

Let's get started with the simplest possible usage.

```
ZnClient new get: 'http://zn.stfx.eu/zn/small.html'.
```

Select the expression and print its result. You should get a `String` back containing a very small HTML document. The `get:` method belongs to the convenience API. Let's use a more general API to be a bit more explicit about what happened.

```
ZnClient new
   url: 'http://zn.stfx.eu/zn/small.html';
   get;
   response.
```

Here we explicitly set the url of the resource to access using `url:`, then we execute an HTTP GET using `get` and we finally ask for the response object using `response`. The above returns a `ZnResponse` object. Of course you can inspect it. It consists of 3 elements:

1. a `ZnStatusLine` object,
2. a `ZnHeaders` object and
3. an optional `ZnEntity` object.

The status line says HTTP/1.1 200 OK, which means the request was successful. This can be tested by sending `isSuccess` to either the response object or the client itself. The headers contain meta data related to the response, including:

- the content-type (a mime-type), accessible with the `contentType` message

- the content-length (a byte count), accessible with the `contentLength` message
- the date the response was generated
- the server that generated the response

The entity is the actual resource: the bytes that should be interpreted in the context of the content-type mime-type. Zn automatically converts non-binary mime-types into `Strings` using the correct encoding. In our example, the entity is an instance of `ZnStringEntity`, a concrete subclass of `ZnEntity`.

Like any object, you can inspect or explore the `ZnResponse` object. You might be wondering how this response was actually transferred over the network. That is easy with Zinc, as the key HTTP objects all implement `writeOn:` that displays the raw format of the response i.e., what has been transmitted through the network.

```
| response |
response := ZnClient new
   url: 'http://zn.stfx.eu/zn/small.html';
   get;
   response.
response writeOn: Transcript.
Transcript flush.
```

If you have the Transcript open, you should see something like the following:

```
HTTP/1.1 200 OK
Date: Thu, 26 Mar 2015 23:26:49 GMT
Modification-Date: Thu, 10 Feb 2011 08:32:30 GMT
Content-Length: 113
Server: Zinc HTTP Components 1.0
Vary: Accept-Encoding
Content-Type: text/html;charset=utf-8

<html>
<head><title>Small</title></head>
<body><h1>Small</h1><p>This is a small HTML document</p></body>
</html>
```

The first CRLF terminated line is the status line. Next are the headers, each on a line with a key and a value. An empty line ends the headers. Finally, the entity bytes follows, either up to the content length or up to the end of the stream.

You might wonder what the request looked like when it went over the network? You can find it out using the same technique.

```
| request |
request := (ZnClient new)
   url: 'http://zn.stfx.eu/zn/small.html';
   get;
   request.
request writeOn: Transcript.
Transcript flush.
```

In an opened Transcript you will now see:

```
GET /zn/small.html HTTP/1.1
Accept: */*
User-Agent: Zinc HTTP Components 1.0
Host: zn.stfx.eu
```

A ZnRequest object consists of 3 elements:

1. a ZnRequestLine object,
2. a ZnHeaders object and
3. an optional ZnEntity object.

The request line contains the HTTP method (sometimes called verb), URL and the HTTP protocol version. Next come the request headers, similar to the response headers, meta data including:

- the host we want to talk to,
- the kind of mime-types that we accept or prefer, and
- the user-agent that we are.

If you look carefully at the Transcript you will see the empty line terminating the headers. For most kinds of requests, like for a GET, there is no entity.

For debugging and for learning, it can be helpful to enable logging on the client. Try the following.

```
ZnClient new
   logToTranscript;
   get: 'http://zn.stfx.eu/zn/small.html'.
```

This will print out some information on the Transcript, as shown below.

```
2015-03-26 20:32:30 001 Connection Established zn.stfx.eu:80
    46.137.113.215 223ms
2015-03-26 20:32:30 002 Request Written a ZnRequest(GET
    /zn/small.html) 0ms
2015-03-26 20:32:30 003 Response Read a ZnResponse(200 OK
    text/html;charset=utf-8 113B) 223ms
2015-03-26 20:32:30 004 GET /zn/small.html 200 113B 223ms
```

In a later subsection about server logging, which uses the same mechanism, you will learn how to interpret and customize logging.

2.3 Simplified HTTP Requests

Although ZnClient is absolutely the preferred object to deal with all the intricacies of HTTP, you sometimes wish you could to a quick HTTP request with an absolute minimum amount of typing, especially during debugging. For these occasions there is ZnEasy, a class side only API for quick HTTP requests.

```
ZnEasy get: 'http://zn.stfx.eu/zn/numbers.txt'.
```

The result is always a ZnResponse object. Apart from basic authentication, there are no other options. A nice feature here, more as an example, is some direct ways to ask for image resources as ready to use Forms.

```
ZnEasy getGif:
   'http://esug.org/data/Logos+Graphics/ESUG-Logo/2006/gif/',
   'esug-Logo-Version3.3.-13092006.gif'.
ZnEasy getJpeg: 'http://caretaker.wolf359.be/sun-fire-x2100.jpg'.
ZnEasy getPng: 'http://pharo.org/files/pharo.png'.

(ZnEasy getPng: 'http://chart.googleapis.com/chart?cht=tx&chl=',
   'a^2+b^2=c^2') asMorph openInHand.
```

When you explore the implementation, you will notice that `ZnEasy` uses a `ZnClient` object internally.

2.4 HTTP Success?

A simple view of HTTP is: you request a resource and get a response back containing the resource. But even if the mechanics of HTTP did work, and even that is not guaranteed (see the next section), the response could not be what you expected.

HTTP defines a whole set of so called status codes to define various situations. These codes turn up as part of the status line of a response. The dictionary mapping numeric codes to their textual reason string is predefined.

```
ZnConstants httpStatusCodes
```

A good overview can be found in the Wikipedia article List of HTTP status codeshttp://en.wikipedia.org/wiki/List_of_HTTP_status_codes*of* HTTP*status*codes). The most common code, the one that indicates success is numeric code 200 with reason 'OK'. Have a look at the `testing` protocol of `ZnResponse` for how to interpret some of them.

So if you do an HTTP request and get something back, you cannot just assume that all is well. You first have to make sure that the call itself (more specifically the response) was successful. As mentioned before, this is done by sending `isSuccess` to the response or the client.

```
| client |
client := ZnClient new.
client get: 'http://zn.stfx.eu/zn/numbers.txt'.
client isSuccess
   ifTrue: [ client contents lines collect: [ :each | each asNumber
    ] ]
   ifFalse: [ self inform: 'Something went wrong' ]
```

To make it easier to write better HTTP client code, `ZnClient` offers some useful status handling methods in its API. You can ask the client to consider non-successful HTTP responses as errors with the `enforceHTTPSuccess` option. The client will then automatically throw a `ZnHTTPUnsuccesful` exception. This is generally useful when the application code that uses Zinc handles errors.

Additionally, to install a local failure handler, there is the `ifFail:` option. This will invoke a block, optionally passing an exception, whenever something goes wrong. Together, this allows the above code to be rewritten as follows.

```
ZnClient new
   enforceHttpSuccess: true;
   ifFail: [ :ex | self inform: 'Cannot get numbers: ', ex
    printString ];
   get: 'http://zn.stfx.eu/zn/numbers.txt'.
```

Maybe it doesn't look like a big difference, but combined with some other options and features of `ZnClient` that we'll see later on, the code does become more elegant and more reliable at the same time.

2.5 Dealing with Networking Reality

As a network protocol, HTTP is much more complicated than an ordinary message send. The famousFallacies of Distributed Computinghttp://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing*of* Distributed_Computing) paper by Deutsch et. al. eloquently lists the issues involved:

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Zn will signal various exceptions when things go wrong, at different levels. `ZnClient` and the underlying framework have constants, settings and options to deal with various aspects related to these issues.

Doing an HTTP request-response cycle can take an unpredictable amount of time. Client code has to specify a timeout: the maximum amount of time to wait for a response, and be prepared for when that timeout is exceeded. When there is no answer within a specified timeout can mean that some networking component is extremely slow, but it could also mean that the server simply refuses to answer.

Setting the timeout directly on a `ZnClient` is the easiest.

```
ZnClient new
   timeout: 1;
   get: 'http://zn.stfx.eu/zn/small.html'.
```

The timeout counts for each socket level connect, read and write operation, separately. You can dynamically redefine the timeout using the `ZnConnectionTimeout` class, which is a `DynamicVariable` subclass.

```
ZnConnectionTimeout
   value: 5
   during: [ ^ ZnClient new get: 'http://zn.stfx.eu/zn/small.html' ].
```

Zn defines its global default timeout in seconds as a setting.

```
ZnNetworkingUtils defaultSocketStreamTimeout.
ZnNetworkingUtils defaultSocketStreamTimeout: 60.
```

This setting affects most framework level operations, if nothing else is specified.

During the execution of HTTP, various network exceptions, as subclasses of NetworkError, might be thrown. These will all be caught by the `ifFail:` block when installed.

To deal with temporary or intermittent network or server problems, `ZnClient` offers a retry protocol. You can set how many times a request should be retried and how many seconds to wait between retries.

```
ZnClient new
   numberOfRetries: 3;
   retryDelay: 2;
   get: 'http://zn.stfx.eu/zn/small.html'.
```

In the above example, the request will be tried up to 3 times, with a 2 second delay between attempts. Note that the definition of failure/success is broad: it includes for example the option to enforce HTTP success.

2.6 Building URL's

Zn uses `ZnUrl` objects to deal with URLs. `ZnClient` also contains an API to build URLs. Let us revisit our initial example, using explicit URL construction with the `ZnClient` API.

```
ZnClient new
   http;
   host: 'zn.stfx.eu';
   addPath: 'zn';
   addPath: 'small.html';
   get.
```

Instead of giving a string argument to be parsed into a `ZnUrl`, we now provide the necessary elements to construct the URL manually, by sending messages to our `ZnClient` object. With `http` we set what is called the scheme. Then we set the hostname. Since we don't specify a port, the default port for HTTP will be used, port 80. Next we add path elements, extending the path one by one.

A URL can also contain query parameters. Let's do a Google search as an example:

```
ZnClient new
   http;
   host: 'www.google.com';
   addPath: 'search';
   queryAt: 'q' put: 'Pharo Smalltalk';
   get.
```

Query parameters have a name and a value. Certain special characters have to be encoded. You can build the same URL with the `ZnUrl` object, in several ways.

```
ZnUrl new
   scheme: #http;
   host: 'www.google.com';
   port: 80;
   addPathSegment: 'search';
   queryAt: 'q' put: 'Pharo Smalltalk';
   yourself.
```

If you print the above expression, it gives you the printable representation of the URL.

```
http://www.google.com/search?q=Pharo%20Smalltalk
```

This string version can easily be parsed again into a `ZnUrl` object

```
'http://www.google.com/search?q=Pharo%20Smalltalk' asZnUrl.
'http://www.google.com:80/search?q=Pharo Smalltalk' asZnUrl.
```

Note how the `ZnUrl` parser is forgiving with respect to the space, like most browsers would do. When producing an external representation, proper encoding will take place. Please consult the class comment of `ZnUrl` for a more detailed look at the capabilities of `ZnUrl` as a standalone object.

2.7 Submitting HTML Forms

In many web applications HTML forms are used. Examples are forms to enter a search string, a form with a username and password to log in or complex registration forms. In the classic and most common way, this is implemented by sending the data entered in the fields of a form to the server when a submit button is clicked. It is possible to implement the same behavior programmatically using `ZnClient`.

First you have to find out how the form is implemented by looking at the HTML code. Here is an example.

```
<form action="search-handler" method="POST"
    enctype="application/x-www-form-urlencoded">
   Search for: <input type="text" name="search-field"/>
   <input type="submit" value="Go!"/>
</form>
```

This form shows one text input field, preceded by a 'Search for:' label and followed by a submit button with 'Go!' as label. Assuming this appears on a page with URL `http://www.search-engine.com/`, we can implement the behavior of the browser when the user clicks the button, submitting or sending the form data to the server.

```
ZnClient new
   url: 'http://www.search-engine.com/search-handler';
   formAt: 'search-field' put: 'Pharo Smalltalk';
   post.
```

The URL is composed by combining the URL of the page that contains the form with the action specified. There is no need to set the encoding of the request here because the form uses the default encoding `application/x-www-form-urlencoded`. By using the `formAt:put:` method to set the value of a field, an entity of type `ZnApplicationFormUrlEncodedEntity` will be created if needed, and the field name/value association will be stored in it. When finally `post` is invoked, the HTTP request sent to the server will include a properly encoded entity. As far as the server is concerned, it will seem as if a real user submitted the form. Consequently, the response should be the same as when you submit the form manually using a browser. Be careful to include all relevant fields, even the hidden ones.

There is a second type of form encoding called `multipart/form-data`. Here, instead of adding fields, you add `ZnMimePart` instances.

```
<form action="search-handler" method="POST"
    enctype="multipart/form-data">
   Search for: <input type="text" name="search-field"/>
   <input type="submit" value="Go!"/>
</form>
```

The code to submit this form would then be as follows.

```
ZnClient new
   url: 'http://www.search-engine.com/search-handler';
   addPart: (ZnMimePart
```

```
                fieldName: 'search-field'
                value: 'Pharo Smalltalk');
    post.
```

In this case, an entity of type `ZnMultiPartFormDataEntity` is created and used. This type is often used in forms that upload files. Here is an example.

```
<form action="upload-handler" method="POST"
     enctype="multipart/form-data">
    Photo file: <input type="file" name="photo-file"/>
    <input type="submit" value="Upload!"/>
</form>
```

This would be the way to do the upload programmatically.

```
ZnClient new
    url: 'http://www.search-engine.com/upload-handler';
    addPart: (ZnMimePart
                fieldName: 'photo-file'
                fileNamed: '/Pictures/cat.jpg');
    post.
```

Sometimes, the form's submit method is GET instead of POST, just send `get` instead of `post` to the client. Note that this technique of sending form data to a server is different than what happens with raw POST or PUT requests using a REST API. In a later subsection we will come back to this.

2.8 Basic Authentication, Cookies and Sessions

There are various techniques to add authentication, a mechanism to control who accesses which resources, to HTTP. This is orthogonal to HTTP itself. The simplest and most common form of authentication is called 'Basic Authentication'.

```
ZnClient new
    username: 'john@hacker.com' password: 'trustno1';
    get: 'http://www.example.com/secret.txt'.
```

That is all there is to it. If you want to understand how this works, look at how `ZnRequest>>#set-BasicAuthenticationUsername:password:` is implemented.

Basic authentication over plain HTTP is insecure because it transfers the username/password combination obfuscated by encoding it using the trivial Base64 encoding. When used over HTTPS, basic authentication is secure though. Note that when sending multiple requests while reusing the same client, authentication is reset for each request, to prevent the accidental transfer of sensitive data.

Basic authentication is not the same as a web application where you have to log in using a form. In such web applications, e.g an online store that has a login part and a shopping cart per user, state is needed. During the interaction with the web application, the server needs to know that your requests/responses are part of your session: you log in, you add items to your shopping cart and you finally check out and pay. It would be problematic if the server mixed the requests/responses of different users. However, HTTP is by design a stateless protocol: each request/response cycle is independent. This principle is crucial to the scalability of the internet.

The most commonly used technique to overcome this issue, enabling the tracking of state across different request/response cycles is the use of so called cookies. Cookies are basically key/value

pairs connected to a specific server domain. Using a special header, the server asks the client to remember or update the value of a cookie for a domain. On subsequent requests to the same domain, the client will use a special header to present the cookie and its value back to the server. Semantically, the server manages a key/value pair on the client.

As we saw before, a `ZnClient` instance is essentially stateful. It not only tries to reuse a network connection but it also maintains a`ZnUserAgentSession` object, which represents the session. One of the main functions of this session object is to manage cookies, just like your browser does. `ZnCookie` objects are held in a `ZnCookieJar` object inside the session object.

Cookie handling will happen automatically. This is a hypothetical example of how this might work, assuming a site where you have to log in before you are able to access a specific file.

```
ZnClient new
   url: 'http://cloud-storage.com/login';
   formAt: 'username' put: 'john.doe@acme.com';
   formAt: 'password' put: 'trustno1';
   post;
   get: 'http://cloud-storage.com/my-file'.
```

After the `post`, the server will presumably set a cookie to acknowledge a successful login. When a specific file is next requested from the same domain, the client presents the cookie to prove the login. The server knows it can send back the file because it recognizes the cookie as valid. By sending `session` to the client object, you can access the session object and then the remembered cookies.

2.9 PUT, POST, DELETE and other HTTP Methods

A regular request for a resource is done using a GET request. A GET request does not send an entity to the server. The only way for a GET request to transfer information to the server is by encoding it in the URL, either in the path or in query variables. (To be 100% correct we should add that data can be sent as custom headers as well.)

PUT and POST Methods

HTTP provides for two methods (or verbs) to send information to a server. These are called PUT and POST. They both send an entity to the server to transfer data.

In the subsection about submitting HTML forms we already saw how POST is used to send either a `ZnApplicationFormUrlEncodedEntity` or to send a `ZnMultiPartFormDataEntity` containing structured data to a server.

Apart from that, it is also possible to send a raw entity to a server. Of course, the server needs to be prepared to handle this kind of entity coming in. Here are a couple of examples of doing a raw PUT and POST request.

```
ZnClient new
   put: 'http://zn.stfx.eu/echo' contents:'Hello there!'.

ZnClient new
   post: 'http://zn.stfx.eu/echo' contents: #[0 1 2 3 4 5 6 7 8 9].

ZnClient new
   url: 'http://zn.stfx.eu/echo';
   entity: (ZnEntity
```

```
                with: '<xml><object><id>42</id></object></xml>'
                type: ZnMimeType applicationXml);
    post.
```

In the last example we explicitly set the entity to be XML and do a POST. In the first two examples, the convenience contents system is used to automatically create a `ZnStringEntity` of the type `ZnMimeType textPlain`, respectively a `ZnByteArrayEntity` of the type `ZnMimeType applicationOctectStream`.

The difference between PUT and POST is semantic. POST is generally used to create a new resource inside an existing collection or container, or to initiate some action or process. For this reason, the normal response to a POST request is to return the URL (or URI) of the newly created resource. Conventionally, the reponse contains this URL both in the `Location` header accessible via the message `location` and in the entity part.

When a POST successfully created the resource, its HTTP response will be 201 Created. PUT is generally used to update an existing resource of which you know the exact URL (or URI). When a PUT is successful, its HTTP response will be just 200 OK and nothing else will be returned. When we will discuss REST Web Service APIs, we will come back to this.

DELETE and other Methods

The fourth member of the common set of HTTP methods is DELETE. It is very similar to both GET and PUT: you just specify an URL of the resource that you want to delete or remove. When successful, the server will just reply with a 200 OK. That is all there is to it.

Certain HTTP based protocols, like WebDAV, use even more HTTP methods. These can be queried explicitly using the `method:` setter and the `execute` operation.

```
ZnClient new
    url: 'http://www.apache.org';
    method: #OPTIONS;
    execute;
    response.
```

An OPTIONS request does not return an entity, but only meta data that are included in the header of the response. In this example, the response header contains an extra meta data named `Allow` which specifies the list of HTTP methods that may be used on the resource.

2.10 Reusing Network Connections, Redirect Following and Checking for Newer Data

ZnClient Lifecycle

HTTP 1.1 defaults to keeping the client connection to a server open, and the server will do the same. This is useful and faster if you need to issue more than one request. `ZnClient` implements this behavior by default.

```
Array streamContents: [ :stream | | client |
    client := ZnClient new url: 'http://zn.stfx.eu'.
    (1 to: 10) collect: [ :each | | url |
        url := '/random/', each asString.
        stream nextPut: (client path: url; get) ].
    client close ].
```

The above example sets up a client to connect to a specific host. Then it collects the results of 10 different requests, asking for random strings of a specific size. All requests will go over the same network connection.

Neither party is required to keep the connection open for a long time, as this consumes resources. Both parties should be prepared to deal with connections closing, this is not an error. `ZnClient` will try to reuse an existing connection and reconnect once if this reuse fails. The option `connectionReuseTimeout` limits the maximum age for a connection to be reused.

Note how we also close the client using the message `close`. A network connection is an external resource, like a file, that should be properly closed after use. If you don't do that, they will get cleaned up eventually by the system, but it is more efficient to do it yourself.

In many situations, you only want to do one single request. HTTP 1.1 has provisions for this situation. The beOneShot option of `ZnClient` will do just that.

```
ZnClient new
   beOneShot;
   get: 'http://zn.stfx.eu/numbers.txt'.
```

With the beOneShot option, the client notifies the server that it will do just one request and both parties will consequently close the connection after use, automatically. In this case, an explicit close of the `ZnClient` object is no longer needed.

2.11 Redirects

Sometimes when requesting a URL, an HTTP server will not answer immediately but redirect you to another location. For example, Seaside actually does this on each request. This is done with a 301 or 302 response code. You can ask a `ZnResponse` whether it's a redirect with `isRedirect`. In case of a redirect response, the `Location` header will contain the location the server redirects you to. You can access that URL using `location`.

By default, `ZnClient` will follow redirects automatically for up to 3 redirects. You won't even notice unless you activate logging. If for some reason you want to disable this feature, send a `followRedirects: false` to your client. To modify the maximum number of redirects that could be followed, use `maxNumberOfRedirects:`.

Following redirects can be tricky when PUT or POST are involved. Zn implements the common behavior of changing a redirected PUT or POST into a GET while dropping the body entity. Cookies will be resubmitted. Zn also handles relative redirect URLs, although these are not strictly part of the standard.

2.12 If-Modified-Since

A client that already requested a resource in the past can also ask a server if that resource has been modified, i.e., is newer, since he last requested it. If so, the server will give a quick 304 Not Modified response without sending the resource over again. This is done by setting the If-Modified-Since header using `ifModifiedSince:`. This works both for regular requests as well as for downloads.

```
ZnClient new
   url: 'http://zn.stfx.eu/zn/numbers.txt';
   setIfModifiedSince: (Date year: 2011 month: 1 day: 1);
   downloadTo: FileLocator imageDirectory.

ZnClient new
```

```
url: 'http://zn.stfx.eu/zn/numbers.txt';
setIfModifiedSince: (Date year: 2012 month: 1 day: 1);
get;
response.
```

For this to work, the server has to honor this particular protocol interaction, of course.

2.13 Content-Types, Mime-Types and the Accept Header

Asking for a resource with a certain mime-type does not mean that the server will return something of this type. The extension at the end of a URL has no real significance, and the server might have been reconfigured since last you asked for this resource. For example, asking for `http://example.com/foo`,`http://example.com/foo.txt` or `http://example.com/foo.text` could all be the same or all be different, and this may change over time. This is why HTTP resources (entities) are accompanied by a content-type: a mime-type that is an official, cross-platform definition of a file or document type or format. Again, see the Wikipedia article Internet media typehttp://en.wikipedia.org/wiki/Mime-type for more details.

Zn models mime-types using its `ZnMimeType` object which has 3 components:

- a main type, for example text or image,
- a sub type, for example plain or html, or jpeg, png or gif, and
- a number of attributes, for example `charset=utf-8`.

The class side of `ZnMimeType` has some convenience methods for accessing well known mime-types, for example:

```
ZnMimeType textHtml.
```

Note that for textual (non-binary) types, the encoding defaults to UTF-8, the prevalent internet standard. Creating a `ZnMimeType` object is also as easy as sending `asZnMimeType` to a `String`.

```
'text/html;charset=utf-8' asZnMimeType.
```

The subtype can be a wildcard, indicated by a `*`. This allows for matching.

```
ZnMimeType textHtml matches: ZnMimeType text.
```

With `ZnClient` you can set the accept request header to indicate what you as a client expect, and optionally enforce that the server returns the type you asked for.

```
ZnClient new
   enforceAcceptContentType: true;
   accept: ZnMimeType textPlain;
   get: 'http://zn.stfx.eu/zn/numbers.txt'.
```

The above code indicates to the server that we want a `text/plain` type resource by means of the `Accept` header. When the response comes back and it is not of that type, the client will raise a `ZnUnexpectedContentType` exception. Again, this will be handled by the `ifFail:` block, when specified.

2.14 Headers

HTTP meta data, both for requests and for responses, is specified using headers. These are key/-value pairs, both strings. A large number of predefined headers exists, see this List of HTTP

header fieldshttp://en.wikipedia.org/wiki/HTTP_header. The exact semantics of each header, especially their value, can be very complicated. Also, although headers are key/value pairs, they are more than a regular dictionary. There can be more values for the same key and keys are often written using a canonical capitalization, like 'Content-Type'.

HTTP provides for a way to do a request, just like a regular GET but with a response that contains only the meta data, the status line and headers, but not the actual resource or entity. This is called a HEAD request.

```
ZnClient new
   head: 'http://zn.stfx.eu/zn/small.html';
   response.
```

Since there is no content, we have to look at the `headers` of the response object. Note that the content-type and content-length headers will be set, as if there was an entity, although none is transferred.

`ZnClient` allows you to easily specify custom headers for which there is not yet a predefined accessor, which is most of them. At the framework level,`ZnResponse` and `ZnRequest` offer some more predefined accessors, as well as a way to set and query any custom header by accessing their headers sub object. The following are all equivalent:

```
ZnClient new accept: 'text/*'.
ZnClient new request setAccept: 'text/*'.
ZnClient new request headers at: 'Accept' put: 'text/*'.
ZnClient new request headers at: 'ACCEPT' put: 'text/*'.
ZnClient new request headers at: 'accept' put: 'text/*'.
```

Once a request is executed, you can query the response headers like this:

```
client response isConnectionClose.
(client response headers at: 'Connection' ifAbsent: [ '' ])
   sameAs: 'close'.
```

2.15 Entities, Content Readers and Writers

As mentioned before, `ZnMessages` (`ZnRequests` and `ZnResponses`) can hold an optional `ZnEntity` as body. By now we used almost all concrete subclasses of `ZnEntity`:

- `ZnStringEntity`
- `ZnByteArrayEntity`
- `ZnApplicationFormUrlEncodedEntity`
- `ZnMultiPartFormDataEntity`
- `ZnStreamingEntity`

Like all other fundamental Zn domain model objects, these can and are used both by clients and servers. All `ZnEntities` have a content type (a mime-type) and a content length (in bytes). Their basic behavior is that they can be written to or read from a binary stream. All but the last one are classic, in-memory objects.

`ZnStreamingEntity` is special: it contains a read or write stream to be used once in one direction only. If you want to transfer a 10 Mb file, using a normal entity, this would result in the 10 Mb being taken into memory. With a streaming entity, a file stream is opened to the file, and the data is then copied using a buffer of a couple of tens of Kb. This is obviously more efficient. The limitation is that this only works if the exact size is known upfront.

Knowing that a `ZnStringEntity` has a content type of XML or JSON is however not enough to interpret the data correctly. You might need a parser to convert the representation to Pharo or a writer to convert Pharo into the proper representation. That is where the `ZnClient` options `contentReader` and `contentWriter` are useful.

If the content reader is nil (the default), `contents` will return the `contents` of the response object, usually a `String` or `ByteArray`.

To customize the content reader, you specify a block that will be given the incoming entity and that is then supposed to parse the incoming representation, for example as below:

```
ZnClient new
   systemPolicy;
   url: 'http://zn.stfx.eu/zn/numbers.txt';
   accept: ZnMimeType textPlain;
   contentReader: [ :entity |
      entity contents lines
         collect: [ :each | each asInteger ] ];
  get.
```

In this example, `get` (which returns the same as `contents`) will no longer return a `String` but a collection of numbers. Note also that by using `systemPolicy` in combination with an `accept:` we handle most error cases before the content reader start doing its work, so it does no longer have to check for good incoming data. In any case, when the `contentReader` throws an exception, it can be caught by the `ifFail:` block.

If the content writer is nil (the default), `contents:` will take a Pharo object and pass it to `ZnEntity` class' `with:` instance creation method. This will create either a `text/plain` `String` entity or an `application/octectstream` `ByteArray` entity.

You could further customize the entity by sending`contentType:` with another mime type. Or you could completely skip the `contents:` mechanism and supply your own entity to `entity:`.

To customize the content writer, you need to pass a one-argument block to the `contentWriter:` message. The block should create and return an entity. A theoretical example is given next.

```
ZnClient new
   url: 'http://internet-calculator.com/sum';
   contentWriter: [ :numberCollection |
      ZnEntity text:
         (Character space join:
            (numberCollection collect: [ :each | each asString ])) ];
   contentReader: [ :entity | entity contents asNumber ];
   post.
```

Assuming there is a web service at `http://internet-calculator.com` where you can send numbers to, we send a whitespace separated list of numbers to its sum URI and expect a number back. Exceptions occuring in the content writer can be caught with the `ifFail:` block.

2.16 Downloading, Uploading and Signalling Progress

Often, you want to download a resource from some internet server and store its contents in a file. The well known curl and wget Unix utilities are often used to do this in scripts. There is a handy convenience method in `ZnClient` to do just that.

```
ZnClient new
   url: 'http://zn.stfx.eu/zn/numbers.txt';
   downloadTo: FileLocator imageDirectory.
```

The example will download the URL and save it in a file named `numbers.txt` next to your image. The argument to `downloadTo:` can be a `FileReference` or a path string, designating either a file or a directory. When it is a directory, the last component of the URL will be used to create a new file in that directory. When it is a file, that file will be used as given. Additionally, the `downloadTo:` operation will use streaming so that a large file will not be taken into memory all at once, but will be copied in a loop using a buffer.

The inverse, uploading the raw contents of file, is just as easy thanks to the convenience method `uploadEntityFrom:`. Given a file reference or a path string, it will set the current request entity to a `ZnStreamingEntity` reading bytes from the named file. The content type will be guessed based on the file name extension. If needed you can next override that mime type using `contentType:`. Here is a hypothetical example uploading the contents of the file`numbers.txt` using a POST to the URL specified, again using an efficient streaming copy.

```
ZnClient new
   url: 'http://cloudstorage.com/myfiles/';
   username: 'john@foo.co.uk' password: 'asecret';
   uploadEntityFrom: FileLocator imageDirectory / 'numbers.txt';
   post.
```

Some HTTP operations, particularly those involving large resources, might take some time, especially when slower networks or servers are involved. During interactive use, Pharo often indicates progress during operations that take a bit longer. `ZnClient` can do that too using the `signalProgress` option. By default this is off. Here is an example.

```
UIManager default informUserDuring: [ :bar |
   bar label: 'Downloading latest Pharo image...'.
   [ ^ ZnClient new
         signalProgress: true;
         url: 'http://files.pharo.org/image/stable/latest.zip';
         downloadTo: FileLocator imageDirectory ]
   on: HTTPProgress
   do: [ :progress |
         bar label: progress printString.
         progress isEmpty ifFalse: [ bar current: progress
    percentage ].
         progress resume ] ]
```

2.17 Client Options, Policies and Proxies

To handle its large set of options, `ZnClient` implements a uniform, generic option mechanism using the `optionAt:put:` and `optionAt:ifAbsent:` methods (this last one always defines an explicit default), storing them lazily in a dictionary. The method category `options` includes all accessors to actual settings.

Options are generally named after their accessor, a notable exception is `beOneShot`. For example, the timeout option has a getter named `timeout` and setter named `timeout:` whose implementation defines its default

```
^ self
   optionAt: #timeout
   ifAbsent: [ ZnNetworkingUtils defaultSocketStreamTimeout ]
```

The set of all option defaults defines the default policy of `ZnClient`. For certain scenarios, there are policy methods that set several options at once. The most useful one is called `systemPolicy`. It specifies good practice behavior for when system level code does an HTTP call:

```
ZnClient>>systemPolicy
   self
      enforceHttpSuccess: true;
      enforceAcceptContentType: true;
      numberOfRetries: 2
```

Also, in some networks you do not talk to internet web servers directly, but indirectly via a proxy. Such a proxy controls and regulates traffic. A proxy can improve performance by caching often used resources, but only if there is a sufficiently high hit rate.

Zn client functionality will automatically use the proxy settings defined in your Pharo image. The UI to set a proxy host, port, username or password can be found in the Settings browser under the Network category. Accessing localhost will bypass the proxy. To find out more about Zn's usage of the proxy settings, start by browsing the `proxy` method category of `ZnNetworkingUtils`.

2.18 Conclusion

Zinc is a solid and very flexible HTTP library. This chapter only presented the client-side of Zinc i.e., how to use it to send HTTP requests and receive responses back. Through several code examples, we demonstrated some of the possibilities of Zinc and also its simplicity. Zinc relies on a very good object-centric decomposition of the HTTP concepts. It results in an easy to understand and extensible library.

CHAPTER 3

Zinc HTTP: The Server Side

Zinc is both a client and server HTTP library written and maintained by Sven van Caekenberghe. HTTP clients and servers are each others' mirror: An HTTP client sends a request and receives a response. An HTTP server receives a request and sends a response. Hence the fundamental Zn framework objects are used to implement both clients and servers.

This chapter focuses on the server-side features of Zinc and demonstrates through small, elegant and robust examples some possibilities of this powerful library. The client side is described in Chapter 2

3.1 Running a Simple HTTP Server

Getting an independent HTTP server up and running inside a Pharo image is surprisingly easy.

```
ZnServer startDefaultOn: 1701.
```

Don't try this just yet. To be able to see what is going on, it is better to enable logging, as follows:

```
(ZnServer defaultOn: 1701)
   logToTranscript;
   start.
```

This starts the default HTTP server, listening on port 1701. We use 1701 in the example because using a port below 1024 requires special OS level privileges, and ports like 8080 might already be in use. Visiting http://localhost:1701 with a browser yields the Zn welcome page. The Transcript produces output related to the server's activities, for example:

```
2015-06-11 18:06:31 001 565881 Server Socket Bound 0.0.0.0:1701
2015-06-11 18:06:31 002 275888 Started ZnManagingMultiThreadedServer
    HTTP port 1701
2015-06-11 18:06:35 003 565881 Connection Accepted 127.0.0.1
2015-06-11 18:06:35 004 097901 Request Read a ZnRequest(GET /) 0ms
2015-06-11 18:06:35 005 097901 Request Handled a ZnRequest(GET /) 0ms
2015-06-11 18:06:35 006 097901 Response Written a ZnResponse(200 OK
    text/html;charset=utf-8 977B) 2ms
```

```
2015-06-11 18:06:35 007 097901 GET / 200 977B 2ms
2015-06-11 18:06:35 008 097901 Request Read a ZnRequest(GET
    /favicon.ico) 129ms
2015-06-11 18:06:35 009 097901 Request Handled a ZnRequest(GET
    /favicon.ico) 0ms
2015-06-11 18:06:35 010 097901 Response Written a ZnResponse(200 OK
    image/vnd.microsoft.icon 318B) 2ms
2015-06-11 18:06:35 011 097901 GET /favicon.ico 200 318B 2ms
2015-06-11 18:06:35 012 097901 Request Read a ZnRequest(GET
    /favicon.ico) 32ms
2015-06-11 18:06:35 013 097901 Request Handled a ZnRequest(GET
    /favicon.ico) 0ms
2015-06-11 18:06:35 014 097901 Response Written a ZnResponse(200 OK
    image/vnd.microsoft.icon 318B) 0ms
2015-06-11 18:06:35 015 097901 GET /favicon.ico 200 318B 0ms
2015-06-11 18:07:05 016 097901 Server Read Error ConnectionTimedOut:
    Data receive timed out.
2015-06-11 18:07:05 017 097901 Server Connection Closed 127.0.0.1
```

You can see the server starting and initializing its server socket on which it listens for incoming connections. When a connection comes in, it starts executing its request-response loop. Then it gets a `GET` request for / (the home page), to which it answers a 200 OK response with 997 bytes of HTML. The browser also asks for a `favicon.ico`, which the server supplies. The request-response loop is kept alive for some time and usually closes when the other end does. Although it looks like an error, it actually is normal, expected behavior.

The example uses the default server: Zn manages a default server to ease interactive experimentation. The server object is obtained by:`ZnServer default`. The default server also survives image save and restart cycles and needs to be stopped with `ZnServer stopDefault`. The Transcript output will confirm what happens:

```
2015-06-11 18:11:07 018 565881 Server Socket Released 0.0.0.0:1701
2015-06-11 18:11:07 019 275888 Stopped ZnManagingMultiThreadedServer
    HTTP port 1701
```

Due to its implementation, the server will print a debug notification: `Wait for accept timed out`, every 5 minutes. Again, although it looks like an error, it is by design and normal, expected behavior.

3.2 Server Delegate, Testing and Debugging

The functional behavior of a `ZnServer` is defined by an object called its delegate. A delegate implements the key method `handleRequest:` which gets the incoming request as parameter and has to produce a response as result. The delegate only needs to reason in terms of a `ZnRequest` and a `ZnResponse`. The technical side of being an HTTP server, like the protocol itself, the networking and the (optional) multiprocessing, is handled by the server object.

This allows us to write what is arguably the simplest possible HTTP server behavior:

```
(ZnServer startDefaultOn: 1701)
   onRequestRespond: [ :request |
      ZnResponse ok: (ZnEntity text: 'Hello World!') ].
```

Now go to http://localhost:1701 or do:

```
ZnEasy get: 'http://localhost:1701'.
```

This server does not look at the incoming request. It always answers `200 OK` with a `text/plain` string `Hello World!`. The `onRequestRespond:` method accepts a block that takes a request and that should produce a response. It is implemented using the helper object `ZnValueDelegate`, which converts `handleRequest:` to `value:` on a wrapped block.

3.3 The Default Server Delegate

Out of the box, a `ZnServer` will have a certain functionality that is related to testing and debugging. The `ZnDefaultServerDelegate` object implements this behavior. Assuming a server is running locally on port 1701, this is the list of URLs that are available.

- http://localhost:1701/ the default for `/`, equivalent to `/welcome`
- http://localhost:1701/bytes a collection of bytes
- http://localhost:1701/dw-bench a dynamically generated page for benchmarking
- http://localhost:1701/echo a textual response echoing the request
- http://localhost:1701/favicon.ico nice Zn favicon used by browsers
- http://localhost:1701/form-test-1 to `/form-test-3` are form test pages
- http://localhost:1701/help this list of URLs
- http://localhost:1701/random a random string of characters
- http://localhost:1701/session information about the session
- http://localhost:1701/status a textual page showing some server internals
- http://localhost:1701/unicode a UTF-8 encoded page listing the first 591 Unicode characters
- http://localhost:1701/welcome the standard Zn greeting page

The random handler normally returns 64 characters, you can specify your own size as well. For example, `/random/1024` will respond with a 1Kb random string. The random pattern consists of hexadecimal digits and ends with a linefeed. The standard, slower UTF-8 encoding is used instead of the faster LATIN-1 encoding.

The bytes handler has a similar size option. Its output is in the form of a repeating BCDA pattern. When requesting equally sized byte patterns repeatably, some extra server side caching will improve performance.

3.4 Testing and Debugging

The echo handler is used extensively by the unit tests. It not only lists the request headers as received by the server, but even the entity if there is one. In case of a non-binary entity, the textual contents will be included. This is really useful to debug PUT or POST requests.

In general, to help in debugging a server, enabling logging is important to learn what is going on. Breakpoints can be put anywhere in the server, but interrupting a running server can sometimes be a bit hard or produce strange results. This is because the server and its spawned handler subprocesses are different from the UI process.

When logging is enabled, the server will also keep track of the last request and response it processed. You can inspect these to find out what happened, even if there was no debugger raised.

3.5 Server Authenticator

Similar to the delegate, a `ZnServer` also has an authenticator object whose function is to authenticate requests. An authenticator has to implement the`authenticateRequest:do:` method whose first argument is the incoming request and second argument a block. This method has to produce a response, like`handleRequest:` does. If the request is allowed, the block should be evaluated, which will produce the response. If the request is denied, the authenticator should generate a 401 Unauthorized response. One simple authenticator is available to add basic HTTP authentication:

```
(ZnServer startDefaultOn: 1701)
   authenticator: (ZnBasicAuthenticator username: 'admin' password:
    'secret').
```

Now, when you try to visit the server at http://localhost:1701 you will have to provide a username and password. Note that it is also possible to use `ZnEasy` to send a get request to this URL with these credentials.

```
ZnEasy
   get: 'http://localhost:1701'
   username: 'admin'
   password: 'secret'.
```

Using `ZnBasicAuthenticator` or implementing an alternative authenticator is only one of several possibilities to address the problem of adding security to a web site or web application.

3.6 Logging

Log output consists of a log message preceded by a number of fixed fields. Here is an example of a server log.

```
2015-06-11 10:19:59 001 220937 Server Socket Bound 0.0.0.0:1701
2015-06-11 10:19:59 002 233075 Started ZnManagingMultiThreadedServer
    HTTP port 1701
2015-06-11 10:25:36 003 220937 Connection Accepted 127.0.0.1
2015-06-11 10:25:36 004 879540 Request Read a ZnRequest(GET /help)
    2ms
2015-06-11 10:25:36 005 879540 Request Handled a ZnRequest(GET
    /help) 0ms
2015-06-11 10:25:36 006 879540 Response Written a ZnResponse(200 OK
    text/html;charset=utf-8 867B) 0ms
2015-06-11 10:25:36 007 879540 GET /help 200 867B 0ms
2015-06-11 10:25:38 008 879540 Request Read a ZnRequest(GET /help)
    1770ms
2015-06-11 10:25:38 009 879540 Request Handled a ZnRequest(GET
    /help) 0ms
2015-06-11 10:25:38 010 879540 Response Written a ZnResponse(200 OK
    text/html;charset=utf-8 867B) 0ms
2015-06-11 10:25:38 011 879540 GET /help 200 867B 0ms
2015-06-11 10:25:44 012 879540 Request Read a ZnRequest(GET
    /unicode) 6082ms
2015-06-11 10:25:44 013 879540 Request Handled a ZnRequest(GET
    /unicode) 5ms
```

```
2015-06-11 10:25:44 014 879540 Response Written a ZnResponse(200 OK
    text/html;charset=utf-8 11454B) 2ms
2015-06-11 10:25:44 015 879540 GET /unicode 200 11454B 7ms
```

The first two fields are the date and time in a fixed sized format. The next field is the id of the log entry. The next number is a fixed sized hash of the process ID. Note how 3 different processes are involved: the one starting the server (probably the UI process), the actual server listening process, and the client worker process spawned to handle the request.

Both `ZnClient` and `ZnServer` implement logging using a similar mechanism based on the announcements framework. `ZnLogEvents` are subclasses of the `Announcement` class and are sent by an HTTP server or client containing logging information. A log event has a TimeStamp, an id, and a message.

To log something, a server or client uses its own log methods. For example, a server receives a `logConnectionAccepted:` message with the socket that will process the request as argument. In `ZnSingleThreadedServer`, the implementation of `logConnectionAccepted:` is:

```
logConnectionAccepted: socket
   logLevel < 3 ifTrue: [ ^ nil ].
   ^ (self newLogEvent: ZnConnectionAcceptedEvent)
      address: ([ socket remoteAddress ] on: Error do: [ nil ]);
      emit
```

This logging mechnism can be easily customized by implementing subclasses of `ZnLogEvent`. For example, `ZnConnectionAcceptedEvent` is a subclass of `ZnLogEvent` customized for connection acceptation.

You can also provide your own listener for `ZnLogEvents`. The following example shows how to log events in a file named`zn.log`, next to the image.

```
| logger |
loggerStream := (FileSystem workingDirectory / 'zn.log') writeStream.
ZnLogEvent announcer
   when: ZnLogEvent
   do: [ :event | loggerStream lf; print: event ].
(ZnServer defaultOn: 1701) start.
```

3.7 Server Variants and Life Cycle

The class side of `ZnServer` is actually a factory to instantiate a particular concrete `ZnServer` subclass, as can be seen in `defaultServerClass`. The hierarchy looks as follows.

```
ZnServer
   + ZnSingleThreadedServer
      + ZnMultiThreadedServer
         + ZnManagedMultiThreadedServer
```

`ZnServer` is an abstract class. `ZnSingleThreadedServer` implements the core server functionality. It runs in one single process, which means it can only handle one request at a time, making it easier to understand and debug. `ZnMultiThreadedServer` spawns a new process on each incoming request, possibly handling multiple request/response cycles on the same connection. `ZnManagedMultiThreadedServers` keeps explicit track of which connections are alive so that they can be stopped when the server stops instead of letting them die out.

Server instances can be started and stopped using `start` and `stop`. By registering a server instance, by sending it `register`, it becomes managed. That means it will survive image save and

restart. This only happens automatically with the `default` server, for other server instances it needs to be enabled manually.

The main parameter a server needs is the port on which it will listen. Additionally, you can restrict the network interface the server should listen on by setting its `bindingAddress:` to some IP address. The default, which is `nil` or `#[0 0 0 0]`, means to listen on all interfaces. With `#[127 0 0 1]`, the server will not respond to requests over its normal network, but only to requests coming from the same host. This is often used to increase security while proxying.

```
(ZnServer defaultOn: 1701)
   bindingAddress: #[127 0 0 1];
   logToTranscript;
   start.
```

3.8 Static File Server

When most people think about a web server, they imagine what is technically called static file serving. There is a directory full of HTML, image, CSS, and other files, somewhere on a machine, and the web server serves these files over HTTP to web browser clients anywhere on the network. This is indeed what Apache does in its most basic form.

Zn can do this by using a `ZnStaticFileServerDelegate`. Given a directory and an optional prefix, this delegate will serve all files it finds in that directory, for example:

```
(ZnServer startDefaultOn: 1701)
   delegate: (
      ZnStaticFileServerDelegate new
         directory: '/var/www' asFileReference;
         prefixFromString: 'static-files';
         yourself).
```

If we suppose the contents of `/var/www` is

- index.html
- small.html

You can access these files with these URLs

- http://localhost:1701/static-files/index.html
- http://localhost:1701/static-files/small.html

The prefix is added in front of all files being served, the actual directory where the files reside is of course invisible to the end web user. If no prefix is specified, the files will be served directly.

Note how all other URLs result in a 404 Not found error. Note that while the `ZnStaticFileServerDelegate` is very simple, it does have a couple of capabilities. Most importantly, it will do what most people expect with respect to directories. Consider the following URLs:

- http://localhost:1701/static-files
- http://localhost:1701/static-files/

The first URL above will result in a redirect to the second. The second URL will look for either an `index.html` or `index.htm` file and serve that. Automatic generation of an index page when there is no index file is not implemented.

As a static file server, the following features are implemented:

- automatic determination of the content mime-type based on the file extension
- correct setting of the content length based on the file length
- usage of streaming
- addition of correct modification date based on the files' last modification date
- correct reaction to the if-modified-since protocol
- optional expiration and caching control

Here is a more complex example:

```
(ZnServer startDefaultOn: 1701)
   logToTranscript;
   delegate: (
      ZnStaticFileServerDelegate new
         directory: '/var/www' asFileReference;
         mimeTypeExpirations: ZnStaticFileServerDelegate
    defaultMimeTypeExpirations;
         yourself);
   authenticator: (
      ZnBasicAuthenticator username: 'admin' password: 'secret').
```

In the above example, we add the optional expiration and caching control based on default settings. Note that it is easy to combine static file serving with logging and authentication.

3.9 Dispatching

Dispatching or routing is HTTP application server speak for deciding what part of the software will handle an incoming request. This decision can be made on any of the properties of the request: the HTTP method, the URL or part of it, the query parameters, the meta headers and the entity body. Different applications will prefer different kinds of solutions to this problem.

Zinc HTTP Components is a general framework that offers all the necessary components to build your own dispatcher. Out of the box, there are the different delegates that we discussed before. Most of these have hand coded dispatching in their `handleRequest:` method.

`ZnDefaultServerDelegate` can be configured to perform dispatching as it uses a prefix map internally that maps URI prefixes to internal methods. Configuration is by installing a block as the value to a prefix, which accepts the request and produces a response. Here is an example of using that capability:

```
| staticFileServerDelegate |

ZnServer startDefaultOn: 8080.

(staticFileServerDelegate := ZnStaticFileServerDelegate new)
   prefixFromString: 'zn';
   directory: '/home/ubuntu/zn' asFileReference.

ZnServer default delegate prefixMap
   at: 'zn'
   put: [ :request | staticFileServerDelegate handleRequest: request
    ];
   at: 'redirect-to-zn'
```

```
   put: [ :request | ZnResponse redirect: '/zn/index.html' ];
   at: '/'
   put: 'redirect-to-zn'.
```

This is taken from the configuration of what runs at http://zn.stfx.eu. A static web server is set up under the `zn` prefix pointing to the directory `/home/ubuntu/zn`. The prefix map of the default delegate is kept as is, with its standard functionality, but is modified, such that

- anything with a `zn` prefix is directly forwarded to the static file server
- a special `redirect-to-zn` prefix is set up which will issue a redirect to `/zn/index.html`
- the default `/` handler is linked to `redirect-to-zn` instead of the default `welcome:`

Another option is to use `ZnDispatcherDelegate`.

```
(ZnServer startDefaultOn: 9090) delegate: (
   ZnDispatcherDelegate new
      map: '/hello'
      to: [ :request :response |
            response entity: (ZnEntity html: '<h1>hello!</h1>') ]).
```

You configure the dispatcher using `map:to:` methods. First argument is the prefix, second argument is a block taking two arguments: the incoming request and an already instantiated response.

3.10 Character Encoding

Proper character encoding and decoding is crucial in today's international world. Pharo encodes characters and strings using Unicode. The primary internet encoding is UTF-8, but a couple of others are used as well. To translate between these two, a concrete `ZnCharacterEncoding` subclass like `ZnUTF8Encoder` is used.

`ZnCharacterEncoding` is an extension and reimplementation of regular `TextConverter`. It only works on binary input and generated binary output and it adds the ability to compute the encoded length of a source character, a crucial operation for HTTP. It is more correct and will throw proper exceptions when things go wrong.

Character encoding is mostly invisible. Here are some code snippets using the encoders directly, feel free to substitute any Unicode character to make the test more interesting.

```
| encoder string |
encoder := ZnUTF8Encoder new.
string := 'any Unicode'.
self assert: (encoder decodeBytes: (encoder encodeString: string))
    equals: string.
encoder encodedByteCountForString: string.
```

There are no automatic conversions in Zinc, so no defaults are assumed. Instead you should specify a proper Content-Type header including the charset information. Otherwise Zinc has no chance of knowing what to use and the default NullEncoder will make your string wrong.

Consider the following example:

```
ZnServer startDefaultOn: 1701.

ZnClient new
```

```
   url: 'http://localhost:1701/echo';
   entity: (ZnEntity with: 'An der schönen blauen Donau');
   post.

ZnClient new
   url: 'http://localhost:1701/echo';
   entity: (
      ZnEntity
         with: 'An der schönen blauen Donau'
        type: (ZnMimeType textPlain charSet: #'iso-8859-1';
    yourself));
   post;
   yourself.
```

In the first case, a UTF-8 encoded string is POST-ed and correctly returned (in a UTF-8 encoded response).

In the second case, an ISO-8859-1 encoded string is POST-ed and correctly returned (in a UTF-8 encoded response).

In both cases the decoding was done correctly, using the specified charset (if that is missing, the ZnNullEncoder is used). Now, ö is not a perfect test example because its Unicode encoding value is 246 in decimal, U+00F6 in hex, still fits in 1 byte and hence survives null encoding/decoding (it would not be the case with € for example). That is why the following still works, although it is wrong to drop the charset.

```
ZnClient new
   url: 'http://localhost:1701/echo';
   entity: (
      ZnEntity
         with: 'An der schönen blauen Donau'
         type: (ZnMimeType textPlain clearCharSet; yourself));
   post;
   yourself.
```

3.11 Resource Protection Limits, Content and Transfer Encoding

Internet facing HTTP servers will come under attack by malicious clients. Good security is thus important. The first step is a correct and safe implementation of the HTTP protocol. Another way a server protects itself is by implementing some resource limits.

Zinc HTTP Components currently implements and enforces the following limits:

- maximumLineLength (4Kb), impacting mainly the size of a header pair
- maximumEntitySize (16Mb), the size of incoming entities
- maximumNumberOfDictionaryEntries (256), which is used in headers, URLs and some entities

Of course these values may be customized if one needs to.

Also, Zn implements two important techniques used by HTTP servers when they send entity bodies to clients: Gzip encoding and chunked transfer encoding. The first one adds compres-

sion. The second one is used when the size of an entity is not known up front. Instead chunks of certain sizes are sent until the entity is complete.

All this is handled internally and invisibly. The main object dealing with content and transfer encoding is `ZnEntityReader`. When necessary, the binary socket stream is wrapped with either a `ZnChunkedReadStream` and/or a `GZipReadStream`. Zn also makes use of a `ZnLimitedReadStream` to make sure there is no read beyond the boundaries of one single request's body, provided the content length is set.

3.12 Seaside Adaptor

Seasidehttp://www.seaside.st/ is a well known, cross platform, advanced web application framework. It does not provide its own HTTP server but relies on an existing one by means of an adaptor. It works well with Zn, through the use of a `ZnZincServerAdaptor`. It comes already included with certain Seaside distributions and on Pharo it is the default.

Starting this adaptor can be done using the Seaside Control panel in the normal way. Alternatively, the adaptor can be started programmatically.

```
ZnZincServerAdaptor startOn: 8080.
```

Since Seaside does its own character conversions, the Zn adaptor is configured to work in binary mode for maximum efficiency. There is complete support for POST and PUT requests with entities in form URL, multipart or raw encoding.

There is even a special adaptor that combines being a Seaside adaptor with static file serving, which is useful if you don't like the WAFileLibrary machinery and prefer plain static files served directly.

```
ZnZincStaticServerAdaptor startOn: 8080 andServeFilesFrom:
    '/var/www/'.
```

3.13 Scripting a REST Web Service with Zinc

As a last example of the use of Zinc HTTP, we now show the implementation of REST web services, both the client and the server parts. REST orRepresentational State Transferhttp://en.wikipedia.org/wiki/Representational_state_transfer*state*transfer) is an architectural style most easily described as using HTTP verbs and URIs to deal with encoded resources. Some kind of framework is needed to successfully implement a non-trivial REST service. There is one available in the Zinc-REST-Server package, for example. Here we will implement a very small, simplified example by hand, for educational purposes.

The service will allow arbitrary JSONhttp://www.json.org/ objects to be stored on the server, each identified by an URI allocated by the server. Here is the REST API exposed by the server:

`GET /` Returns a list of all known stored object URIs;

`GET /n` Returns the JSON object known under URI `/n`;

`POST /` Creates a new entry with JSON as contents, returns the new URI;

`PUT /n` Updates (replaces) the contents of an existing JSON object known under URI `/n`;

`DELETE /n` Removes the JSON object known under URI `/n`.

3.14 The Server Code

A proper implementation should best use a couple of classes. However for brevity, the following implementation is written in a workspace, not using any classes. It requires STON and starts by creating two global variables to hold the stored objects and the last ID used. The former is a standard dictionary mapping string URIs to objects.

```
JSONStore := Dictionary new.
ServerLastId := 0.
```

The server implementation uses two helper objects: a `jsonEntityBuilder` and a `mapper`. Both make use of block closures.

```
| jsonEntityBuilder mapper |

jsonEntityBuilder := [ :object |
   ZnEntity
      with: ((String streamContents: [ :stream |
         STON jsonWriter
            on: stream;
            prettyPrint: true;
            nextPut: object.
         stream cr ])
         replaceAll: Character cr with: Character lf)
      type: ZnMimeType applicationJson ].
```

The `jsonEntityBuilder` block helps in transforming objects to a JSON entity. We use the STON writer and reader here because they are backwards compatible with JSON. We use linefeeds to improve compatibility with internet conventions as well as pretty printing to help human interpretation of the data.

```
mapper := {
   [ :request |
      request uri isSlash and: [ request method = #GET ] ]
   ->
   [ :request |
      ZnResponse
         ok: (jsonEntityBuilder value: JSONStore keys asArray) ].
"-------------------------------------------------------------"
   [ :request |
      request uri pathSegments size = 1 and: [ request method = #GET
    ] ]
   ->
   [ :request | | uri |
      uri := request uri pathPrintString.
      JSONStore
         at: uri
         ifPresent: [ :object |
            ZnResponse ok: (jsonEntityBuilder value: object) ]
         ifAbsent: [ ZnResponse notFound: uri ] ].
"-------------------------------------------------------------"
   [ :request |
      (request uri isSlash
```

```
         and: [ request method = #POST ])
         and: [ request contentType = ZnMimeType applicationJson ] ]
   ->
   [ :request | | uri |
      uri := '/', (ServerLastId := ServerLastId + 1) asString.
      JSONStore at: uri put: (STON fromString: request contents).
      (ZnResponse created: uri)
         entity: (jsonEntityBuilder value: 'Created ', uri);
         yourself ].
"-------------------------------------------------------------"
   [ :request |
      (request uri pathSegments size = 1
         and: [ request method = #PUT ])
         and: [ request contentType = ZnMimeType applicationJson ]]
   ->
   [ :request | | uri |
      uri := request uri pathPrintString.
      (JSONStore includesKey: uri)
         ifTrue: [
            JSONStore
               at: uri
               put: (STON fromString: request contents).
            ZnResponse ok: (jsonEntityBuilder value: 'Updated') ]
         ifFalse: [ ZnResponse notFound: uri ] ].
"-------------------------------------------------------------"
   [ :request |
      request uri pathSegments size = 1
         and: [ request method = #DELETE ] ]
   ->
   [ :request | | uri |
      uri := request uri pathPrintString.
      (JSONStore removeKey: uri ifAbsent: [ nil ])
         ifNil: [ ZnResponse notFound: uri ]
         ifNotNil: [
            ZnResponse ok: (jsonEntityBuilder value: 'Deleted') ] ].
}.
```

The mapper is a dynamically created array of associations (not a dictionary). Each association consists of two blocks. The first block is a condition: it tests a request and returns true when it matches. The second block is a handler that is evaluated with the incoming request to produce a response (if and only if the first condition matched).

The associations in the mapper follow exactly the list of the REST API as shown earlier. The server is set up with a block based delegate using the`onRequestRepond:` method. Again, a more object-oriented implementation would use a proper delegate object here, but for this example, the block is sufficient.

The server logic thus becomes: find a matching entry in the mapper and invoke it. If no matching entry is found, we have a bad request. Error handling is of course rather limited in this small example.

```
(ZnServer startDefaultOn: 1701)
   logToTranscript;
   onRequestRespond: [ :request |
      (mapper
            detect: [ :each | each key value: request ]
            ifNone: [ nil ])
         ifNil: [ ZnResponse badRequest: request ]
         ifNotNil: [ :handler | handler value value: request ] ].
```

3.15 Using the Server

Here is an example command line session using the Unix utility curlhttp://en.wikipedia.org/wiki/CURL, interacting with the server.

```
$ curl http://localhost:1701/
[ ]

$ curl -X POST -d '[1,2,3]' -H'Content-type:application/json'
    http://localhost:1701/
"Created /1"

$ curl http://localhost:1701/1
[
    1,
    2,
    3
]

$ curl -X POST -d '{"bar":-2}' -H'Content-type:application/json'
    http://localhost:1701/
"Created /2"

$ curl http://localhost:1701/2
{
    "bar" : -2
}

$ curl -X PUT -d '{"bar":-1}' -H'Content-type:application/json'
    http://localhost:1701/2
"Updated /2"

$ curl http://localhost:1701/2
{
    "bar" : -1
}

$ curl http://localhost:1701/
[
    "/1",
```

```
    "/2"
]

$ curl -X DELETE http://localhost:1701/2
"Deleted /2"

$ curl http://localhost:1701/2
Not Found /2
```

3.16 A Zinc Client

It is trivial to use `ZnClient` to have the same interaction. But we can do better: using a contentWriter and contentReader, we can customise the client to do the JSON conversions automatically.

```
| client |

client := ZnClient new
   url: 'http://localhost:1701';
   enforceHttpSuccess: true;
   accept: ZnMimeType applicationJson;
   contentWriter: [ :object |
      ZnEntity
         with: (String streamContents: [ :stream |
                  STON jsonWriter on: stream; nextPut: object ])
         type: ZnMimeType applicationJson ];
   contentReader: [ :entity |  STON fromString: entity contents ];
   yourself.
```

Now we can hold the same conversation as above, only in this case in terms of real objects.

```
client get: '/'
>>> #()

client post: '/' contents: #(1 2 3)
>>> 'Created /1'

client get: '/1'
>>> #(1 2 3)

client post: '/' contents: (Dictionary with: #bar -> -2)
>>> 'Created /2'

client put: '/2' contents: (Dictionary with: #bar -> -1)
>>> 'Updated'

client get: '/2'
>>> a Dictionary('bar'->-1 )

client get: '/'
```

```
>>> #('/1' '/2')

client delete: '/2'
>>> 'Deleted'

client get: '/2'
>>> throws a ZnHttpUnsuccessful exception
```

3.17 Conclusion

Zinc HTTP Components was written with the explicit goal of allowing users to explore the implementation. The test suite contains many examples that can serve as learning material. This carefulness while writing Zinc HTTP Components code now enable users to customize it to their need or to build on top of it. Zinc is indeed an extremely malleable piece of software.

CHAPTER 4

Tips and Tricks

This chapter will evolve over time and collect nice approaches to various practical problems you may encounter.

4.1 DNS over HTTPS (DoH)

Firefox switched over to using 'DNS over HTTPS (DoH)' by default (https://blog.mozilla.org/netpolicy/2020/02/25/the-facts-mozillas-dns-over-https-doh/).

We can do this in Pharo as well, even out of the box (minus the interpretation of the results, but still).

4.2 First, what is this?

A good description can be found at https://developers.cloudflare.com/1.1.1.1/dns-over-https/. Using the Cloudflare server, we can do the following in Pharo, using the JSON wire format.

```
ZnClient new
 url: 'https://cloudflare-dns.com/dns-query';
 accept: 'application/dns-json';
 queryAt: #name put: 'pharo.org';
 queryAt: #type put: 'A';
 contentReader: [ :entity | STONJSON fromString: entity contents ];
 get.
```

The actual address can be accessed inside the returned result.

```
SocketAddress fromDottedString: (((ZnClient new
 url: 'https://cloudflare-dns.com/dns-query';
 accept: 'application/dns-json';
 queryAt: #name put: 'pharo.org';
 queryAt: #type put: 'A';
 contentReader: [ :entity | STONJSON fromString: entity contents ];
```

```
  get) at: #Answer) first at: #data).
```

If you load the following code, https://github.com/svenvc/NeoDNS. It is just as easy to use the binary UDP wire format.

```
ZnClient new
 url: 'https://cloudflare-dns.com/dns-query';
 accept: 'application/dns-message';
 contentWriter: [ :message |
   ZnEntity with: message asByteArray type:
    'application/dns-message' ];
 contentReader: [ :entity |
   DNSMessage readFrom: entity readStream ];
 contents: (DNSMessage addressByName: 'pharo.org');
 post.
```

Again, the actual address can be accessed inside the returned object.

```
(ZnClient new
  url: 'https://cloudflare-dns.com/dns-query';
  accept: 'application/dns-message';
  contentWriter: [ :message |
    ZnEntity with: message asByteArray type:
     'application/dns-message' ];
  contentReader: [ :entity |
    DNSMessage readFrom: entity readStream ];
  contents: (DNSMessage addressByName: 'pharo.org');
  post) answers first address.
```

Incidentally, a more robust answer can be got as follows:

```
NeoSimplifiedDNSClient default addressForName: 'pharo.org'.
```

4.3 Serving static files

Is it possible a Zinc server returns static files within a specific url path (like the ZnStaticFileServerDelegate) and also returns other logics as shown with `map:#otherPath to: MyWebapp new` ?

The `ZnDefaultServerDelegate` can already do a lot, including the request. Consider part of the startup file of *http://zn.stfx.eu* which serves its own website, together with all the default builtin responses.

So the following are all possible:

- http://zn.stfx.eu
- http://zn.stfx.eu/zn/index.html
- http://zn.stfx.eu/xn/small.html
- http://zn.stfx.eu/welcome
- http://zn.stfx.eu/dw-bench
- http://zn.stfx.eu/help

(the last one lists all configured prefixes).

```
(ZnServer defaultOn: 8180)
  logToTranscript;
  logLevel: 1;
  start.
```

```
(staticFileServerDelegate := ZnStaticFileServerDelegate new)
  prefixFromString: 'zn';
  directory: '/home/stfx/zn' asFileReference.

ZnServer default delegate prefixMap
  at: 'zn'
  put: [ :request | staticFileServerDelegate handleRequest: request
    ];
  at: 'redirect-to-zn'
  put: [ :request | ZnResponse redirect: '/zn/index.html' ];
  at: '/'
  put: 'redirect-to-zn'.
```

There is `ZnPrefixMappingDelegate` you can use.

```
server
   delegate: (ZnPrefixMappingDelegate
      map: 'static' to: staticFileDelegate;
      map: 'app' to: myApp)
```

There is also `ZnStaticFileDecoratorDelegate` if you want to mimick another typical setup where all urls that resolve to a file get served from disk and any other will be forwarded to you app.

```
server
   delegate: (ZnStaticFileDecoratorDelegate
      decorate: myApp
      servingFilesFrom: 'static/')
```