Contents

1 Module `XmlRpc` : XmlRpc Light.

XmlRpc Light is a minimal XmlRpc library based on Xml Light and Ocamlnet.

It provides a type for values, a client class with a simple calling interface, and low-level tools that can be used to implement a server.


```
val version : string
```

Version of XmlRpc-Light as a string.

1.1 High-level interface

Example:

```
let rpc = new XmlRpc.client "http://localhost:8000" in
let result = rpc#call "echo" [`String "hello!"] in
print_endline (XmlRpc.dump result)
```

```
exception Error of (int * string)
```

Raised for all errors including XmlRpc faults (code, string).

```
type value = [ `Array of value list
  | `Binary of string
  | `Boolean of bool
  | `DateTime of XmlRpcDateTime.t
  | `Double of float
  | `Int of int
  | `Int32 of int32
  | `Nil
  | `String of string
  | `Struct of (string * value) list ]
```

Polymorphic variant type for XmlRpc values:

- `` `Array ``: An ordered list of values
- `` `Binary ``: A string containing binary data
- `` `Boolean ``: A boolean
- `` `DateTime ``: A date/time value
- `` `Double ``: A floating-point value
- `` `Int ``: An integer
- `` `Int32 ``: A 32-bit integer
- `` `Nil ``: A null value
- `` `String ``: A string
- `` `Struct ``: An association list of (name, value) pairs

Note that base64-encoding of `` `Binary `` values is done automatically. You do not need to do the encoding yourself.

```
class client : ?debug:bool -> ?headers:(string * string) list -> ?insecure_ssl:bool -> ?timeo
  object
```

```
val url : string
```

Url of the remote XmlRpc server.

```
val mutable debug : bool
```

If true, Xml messages will be printed to standard error.

```
val mutable headers : (string * string) list
```

List of custom HTTP headers to send with each request.

```
val mutable insecure_ssl : bool
```

If true, SSL will be allowed even if the certificate is self-signed.

```
val mutable timeout : float
```

Maximum time to wait for a request to complete, in seconds.

`val mutable useragent : string`

User-agent to send in request headers.

`method url : string`

Gets `url`.

`method debug : bool`

Gets `debug`.

`method set_debug : bool -> unit`

Sets `debug`.

`method headers : (string * string) list`

Gets `headers`.

`method set_headers : (string * string) list -> unit`

Sets `headers`.

`method insecure_ssl : bool`

Gets `insecure_ssl`.

`method set_insecure_ssl : bool -> unit`

Sets `insecure_ssl`.

`method timeout : float`

Gets `timeout`.

`method set_timeout : float -> unit`

Sets `timeout`.

`method useragent : string`

Gets `useragent`.

`method set_useragent : string -> unit`

Sets `useragent`.

`method set_base64_encoder : (string -> string) -> unit`

Sets an alternate Base-64 binary encoding function.

```
    method set_base64_decoder : (string -> string) -> unit
```

Sets an alternate Base-64 binary decoding function.

```
    method set_datetime_encoder : (XmlRpcDateTime.t -> string) -> unit
```

Sets an alternate ISO-8601 date/time encoding function.

```
    method set_datetime_decoder : (string -> XmlRpcDateTime.t) -> unit
```

Sets an alternate ISO-8601 date/time decoding function.

```
    method call : string -> XmlRpc.value list -> XmlRpc.value
```

`call name params` invokes an XmlRpc method and returns the result, or raises `XmlRpc.Error`[1.1] on error.

```
  end
```

Class for XmlRpc clients. Takes a single mandatory argument, `url`.

If `url` is of the form "http://username:password@...", basic authentication will be used.

If `url` starts with "https", Curl will be used instead of Ocamlnet. The "curl" command-line program must be in your path for this to work. You can use the `insecure_ssl` setting to allow connections to servers with self-signed certificates; by default this is false and certificates must be valid.

`timeout` can be used to specify the maximum amount of time elapsed before a connection is cancelled. It defaults to 300.0 seconds.

`headers` may contain an array of (name, value) pairs of additional headers to send with each request.

The `useragent` setting provides a convenient way to change the User-Agent header, which defaults to "XmlRpc-Light/<version>".

The `debug` setting, if true, will enable verbose debugging output to the standard error stream.

```
class multicall : client ->
  object
```

```
    method call : string -> XmlRpc.value list -> XmlRpc.value Lazy.t
```

Adds a call to this `multicall` instance. If the call has already executed, the following exception will be raised: Failure "multicall#call: already executed".

```
    method execute : unit -> unit
```

Forces the call to execute immediately. If the call has already executed and completed successfully, the following exception will be raised: Failure "multicall#execute: already completed".

```
    method result : int -> XmlRpc.value
```

Returns a `multicall` result, executing the call if necessary. The results are numbered starting with zero.

`method executed : bool`

True if the call has executed, whether or not it succeeded.

`method completed : bool`

True of the call has executed and completed successfully.

`end`

Convenience class for `system.multicall` calls.

Instances take an `XmlRpc.client`[1.1] as an argument:

```
# let mc = new XmlRpc.multicall client;;
val mc : XmlRpc.multicall = <obj>
```

The "call" method works like `client#call`, but it returns a lazy value:

```
# let a = mc#call "demo.addTwoNumbers" [‘Int 3; ‘Int 4];;
val a : XmlRpc.value Lazy.t = <lazy>
# let b = mc#call "demo.addTwoNumbers" [‘Int 42; ‘String "oh noes!"];;
val b : XmlRpc.value Lazy.t = <lazy>
# let c = mc#call "demo.addTwoNumbers" [‘Double 3.0; ‘Double 4.0];;
val c : XmlRpc.value Lazy.t = <lazy>
```

At this point, the call has not been executed yet:

```
# mc#executed;;
-- : bool = false
```

As soon as one of the return values is forced, the call is executed:

```
# Lazy.force a;;
-- : XmlRpc.value = ‘Int 7
# mc#executed;;
-- : bool = true
```

Once a call has been executed, this instance cannot be used to make any further calls; instead, a new `multicall` instance must be created:

```
# mc#call "demo.addTwoNumbers" [‘Int 2; ‘Int 2];;
Exception: Failure "multicall#call: already executed".
```

If an XmlRpc fault occurred, the exception will be thrown when the lazy value is forced:

```
# Lazy.force b;;
Exception: XmlRpc.Error (-32602, "server error. invalid method parameters").
```

This will not prevent further methods from executing successfully:

```
# Lazy.force c;;
-- : XmlRpc.value = `Double 7.
```

It is possible for a `multicall` to be executed but not completed, for example if a transport error occurs. Aside from catching the exception, the `completed` property indicates if the call actually went through or not:

```
# mc#completed;;
-- : bool = true
```

It is not necessary to use lazy values. Instead, the call can be executed explicitly, and the results can be retrieved by number:

```
# let mc = new XmlRpc.multicall client;;
val mc : XmlRpc.multicall = <obj>
# ignore (mc#call "demo.addTwoNumbers" [`Int 2; `Int 2]);;
-- : unit = ()
# ignore (mc#call "demo.addTwoNumbers" [`Int 3; `Int 3]);;
-- : unit = ()
# mc#result 1;;
-- : XmlRpc.value = `Int 6
```

1.2 Utility functions

```
val dump : value -> string
```

Converts an XmlRpc value to a human-readable string.

1.3 Low-level interface

```
type message =
  | MethodCall of (string * value list)
  | MethodResponse of value
  | Fault of (int * string)
```

Type for XmlRpc messages.

```
val message_of_xml_element :
  ?base64_decoder:(string -> string) ->
  ?datetime_decoder:(string -> XmlRpcDateTime.t) -> Xml.xml -> message
```

Converts an Xml Light element to an XmlRpc message.

```
val xml_element_of_message :
  ?base64_encoder:(string -> string) ->
  ?datetime_encoder:(XmlRpcDateTime.t -> string) -> message -> Xml.xml
```

Converts an XmlRpc message to an Xml Light element.

```
val value_of_xml_element :
  ?base64_decoder:(string -> string) ->
  ?datetime_decoder:(string -> XmlRpcDateTime.t) -> Xml.xml -> value
```

Converts an Xml Light element to an XmlRpc value.

```
val xml_element_of_value :
  ?base64_encoder:(string -> string) ->
  ?datetime_encoder:(XmlRpcDateTime.t -> string) -> value -> Xml.xml
```

Converts an XmlRpc value to an Xml Light element.

1.4 Server tools

```
val serve :
  ?base64_encoder:(string -> string) ->
  ?base64_decoder:(string -> string) ->
  ?datetime_encoder:(XmlRpcDateTime.t -> string) ->
  ?datetime_decoder:(string -> XmlRpcDateTime.t) ->
  ?error_handler:(exn -> message) ->
  (string -> value list -> value) -> string -> string
```

Creates a function from string (Xml representing a `MethodCall`) to string (Xml representing a `MethodResult` or `Fault`) given a function of the form: (`name` → `params` → `result`), where `name` is the name of the method, `params` is a list of parameter values, and `result` is the result value.

This function can be used to build many different kinds of XmlRpc servers since it makes no assumptions about the network library or other communications method used.

If an exception other than `XmlRpc.Error`[1.1] occurs, the exception is passed to `error_handler`. If `error_handler` returns a message, the message will be used as the result. If an `XmlRpc.Error`[1.1] is raised by either the main function or `error_handler`, it will be converted to an XmlRpc `Fault`. Any other exception raised by `error_handler` is allowed to escape.

For a full-featured, easy-to-use, network-capable server implementation, see the `XmlRpcServer`[2] module.

```
val serve_message :
  ?error_handler:(exn -> message) ->
  (string -> value list -> value) ->
  message -> message
```

Performs the same function as `serve`, but operates on typed messages instead of strings.

```
val default_error_handler : exn -> message
```

The default error handler for `serve`.

This error handler catches all exceptions and converts them into faults by wrapping them in `XmlRpc.Error`.

```
val quiet_error_handler : exn -> message
```

A "quiet" error handler for `serve`.

This error handler simply re-raises the exception. Use this if you want exceptions to remain unhandled so that they will escape to the error log. The client will receive a generic "transport error", which is more secure since it does not reveal any information about the specific exception that occurred.

2 Module `XmlRpcServer` : XmlRpc Light server.

Example:

```
let server = new XmlRpcServer.cgi () in
server#register "demo.sayHello"
  (fun _ -> `String "Hello!");
server#run ()
```

By inheriting from `XmlRpcServer.base`[2.1], all servers provide the following introspection functions by default: `system.listMethods`, `system.getCapabilities`. To prevent their use, use `server#unregister`.

Additionally, the methods `system.methodHelp` and `system.methodSignature` will be made available if at least one method help or method signature is provided.

```
type param_type = [ `Array
  | `Binary
  | `Boolean
  | `DateTime
  | `Double
  | `Int
  | `String
  | `Struct
  | `Undefined ]
```

Type of parameters used in method signatures.

2.1 Base classes

```
class virtual base :
  object
```

```
    val methods : (string, XmlRpc.value list -> XmlRpc.value) Hashtbl.t
```

Hashtable mapping method names to implementation functions.

```
    val mutable base64_encoder : string -> string
```

Base-64 binary encoding function.

```
    val mutable base64_decoder : string -> string
```

Base-64 binary decoding function.

```
    val mutable datetime_encoder : XmlRpcDateTime.t -> string
```

ISO-8601 date/time encoding function.

```
    val mutable datetime_decoder : string -> XmlRpcDateTime.t
```

ISO-8601 date/time decoding function.

```
    val mutable error_handler : exn -> XmlRpc.message
```

Handler for unhandled exceptions.

```
    method set_base64_encoder : (string -> string) -> unit
```

Sets an alternate Base-64 binary encoding function.

```
    method set_base64_decoder : (string -> string) -> unit
```

Sets an alternate Base-64 binary decoding function.

```
    method set_datetime_encoder : (XmlRpcDateTime.t -> string) -> unit
```

Sets an alternate ISO-8601 date/time encoding function.

```
    method set_datetime_decoder : (string -> XmlRpcDateTime.t) -> unit
```

Sets an alternate ISO-8601 date/time decoding function.

```
    method set_error_handler : (exn -> XmlRpc.message) -> unit
```

Sets an alternate handler for unhandled exceptions. See `XmlRpc.default_error_handler`[1.4] and `XmlRpc.quiet_error_handler`[1.4] for examples.

```
    method serve :
      (string -> XmlRpc.value list -> XmlRpc.value) -> string -> string
```

For use in subclasses; calls `XmlRpc.serve`[1.4] with the current encoders, decoders, and error handler.

```
method serve_message :
  (string -> XmlRpc.value list -> XmlRpc.value) ->
  XmlRpc.message -> XmlRpc.message
```

Like `serve`, but operates on messages instead of strings.

```
method register :
  string ->
  ?help:string ->
  ?signature:XmlRpcServer.param_type list ->
  ?signatures:XmlRpcServer.param_type list list ->
  (XmlRpc.value list -> XmlRpc.value) -> unit
```

Registers a method with the server.

If a `help` string is specified, its contents will be returned for calls to `system.methodHelp` for this method.

If `signature` is specified, this method's signature will be published by `system.methodSignature` and (shallow) type-checking will be enabled for parameters passed into this method.

Multiple signatures can be supplied via `signatures` if desired to provide for overloaded methods.

Signatures are of the form `return-type; param1-type; param2-type; ...` where each type is an instance of the `XmlRpcServer.param_type`[2] variant.

```
method unregister : string -> unit
```

Removes a method from the server.

```
method virtual run : unit -> unit
```

Starts the main server process.

```
end
```

Abstract base class for XmlRpc servers.

```
class type server =
  object
```

```
inherit XmlRpcServer.base [2.1]
method run : unit -> unit
```

Starts the main server process.

```
  end
```

Type of concrete XmlRpc server classes.

2.2 Server implementations

`class cgi : unit -> server`

CGI XmlRpc server based on Netcgi2.

`class netplex : ?parallelizer:Netplex_types.parallelizer -> ?handler:string -> unit -> server`

Stand-alone XmlRpc server based on Netplex.

2.3 Utility functions

`val invalid_method : string -> 'a`

Raise an `XmlRpc.Error`[1.1] indicating a method name not found.

`val invalid_params : unit -> 'a`

Raise an `XmlRpc.Error`[1.1] indicating invalid method parameters.

3 Module `XmlRpcDateTime` : Date/time type.

3.1 Types

`exception Parse_error of string`

Raised by `XmlRpcDateTime.of_string`[3.6] if a string could not be parsed. The exception contains the input string.

`type t = int * int * int * int * int * int * int`

Type of XmlRpc-compatible date/time values. (year, month, day, hour, minute, second, time zone offset in minutes)

3.2 Comparison

`val compare : t -> t -> int`

Standard comparator for date/time values. Converts all values to UTC before comparing to ensure correct behavior with values of differing time zones.

`val equal : t -> t -> bool`

Standard equality function for date/time values. Converts all values to UTC before comparing.

`val hash : t -> int`

Standard hash function for date/time values. Converts values to UTC before hashing.

3.3 Current date and time

`val now : unit -> t`

Returns the current date and time in the local time zone.

`val now_utc : unit -> t`

Returns the current date and time in UTC.

3.4 Time zone adjustments

`val set_tz_offset : int -> t -> t`

Adjusts the time zone offset, preserving equality.

`val fix_tz_offset : int -> t -> t`

Forces the time zone offset to a different value, ignoring all other fields. Use this to correct the time zone of a date/time value that was received without a time zone offset and is known not to be UTC.

3.5 Conversion

`val from_unixfloat : float -> t`

Builds a date/time value from epoch seconds in the local time zone.

`val from_unixfloat_utc : float -> t`

Builds a date/time value from epoch seconds in UTC.

`val to_unixfloat : t -> float`

Converts a date/time value to epoch seconds in the local time zone.

`val to_unixfloat_utc : t -> float`

Converts a date/time value to epoch seconds in UTC.

`val from_unixtm : Unix.tm -> t`

Builds a date/time value from a Unix.tm value in the local time zone.

`val from_unixtm_utc : Unix.tm -> t`

Builds a date/time value from a Unix.tm value in UTC.

`val to_unixtm : t -> Unix.tm`

Converts a date/time value to a Unix.tm value in the local time zone.

`val to_unixtm_utc : t -> Unix.tm`

Converts a date/time value to a Unix.tm value in UTC.

3.6 ISO-8601 parsing and generation

`val of_string : string -> t`

Parses an (XmlRpc-flavor) ISO-8601 date/time value from a string.

`val to_string : t -> string`

Generates an ISO-8601 string from a date/time value.

4 Module `XmlRpcBase64` : Base64 codec.

8-bit characters are encoded into 6-bit ones using ASCII lookup tables. Default tables maps 0..63 values on characters A-Z, a-z, 0-9, '+' and '/' (in that order).

`exception Invalid_char`

This exception is raised when reading an invalid character from a base64 input.

`exception Invalid_table`

This exception is raised if the encoding or decoding table size is not correct.

`type encoding_table = char array`

An encoding table maps integers 0..63 to the corresponding char.

`type decoding_table = int array`

A decoding table maps chars 0..255 to the corresponding 0..63 value or -1 if the char is not accepted.

`val str_encode : ?tbl:encoding_table -> string -> string`

Encode a string into Base64.

`val str_decode : ?tbl:decoding_table -> string -> string`

Decode a string encoded into Base64, raise `Invalid_char` if a character in the input string is not a valid one.

`val encode : ?tbl:encoding_table -> char Stream.t -> char Stream.t`

Generic base64 encoding over a character stream.

`val decode : ?tbl:decoding_table -> char Stream.t -> char Stream.t`

Generic base64 decoding over a character stream.

`val make_decoding_table : encoding_table -> decoding_table`

Create a valid decoding table from an encoding one.