timotheecour {.deprecated: [existsFile: fileExists].} (#14735) ... ✓ History

13 contributors +1

439 lines (380 sloc) | 15.3 KB

```nim
#
#
#            Nim's Runtime Library
#        (c) Copyright 2015 Dominik Picheta
#    See the file "copying.txt", included in this
#    distribution, for details about the copyright.
#

## This module implements an asynchronous FTP client. It allows you to connect
## to an FTP server and perform operations on it such as for example:
##
## * The upload of new files.
## * The removal of existing files.
## * Download of files.
## * Changing of files' permissions.
## * Navigation through the FTP server's directories.
##
## Connecting to an FTP server
## ===========================
##
## In order to begin any sort of transfer of files you must first
## connect to an FTP server. You can do so with the ``connect`` procedure.
##
## .. code-block::nim
##    import asyncdispatch, asyncftpclient
##    proc main() {.async.} =
##      var ftp = newAsyncFtpClient("example.com", user = "test", pass = "test")
##      await ftp.connect()
##      echo("Connected")
##    waitFor(main())
##
## A new ``main`` async procedure must be declared to allow the use of the
## ``await`` keyword. The connection will complete asynchronously and the
## client will be connected after the ``await ftp.connect()`` call.
##
## Uploading a new file
## ====================
##
## After a connection is made you can use the ``store`` procedure to upload
## a new file to the FTP server. Make sure to check you are in the correct
## working directory before you do so with the ``pwd`` procedure, you can also
## instead specify an absolute path.
##
## .. code-block::nim
##    import asyncdispatch, asyncftpclient
##    proc main() {.async.} =
##      var ftp = newAsyncFtpClient("example.com", user = "test", pass = "test")
##      await ftp.connect()
##      let currentDir = await ftp.pwd()
##      assert currentDir == "/home/user/"
##      await ftp.store("file.txt", "file.txt")
##      echo("File finished uploading")
##    waitFor(main())
##
## Checking the progress of a file transfer
## ========================================
##
## The progress of either a file upload or a file download can be checked
## by specifying a ``onProgressChanged`` procedure to the ``store`` or
## ``retrFile`` procedures.
##
## .. code-block::nim
##    import asyncdispatch, asyncftpclient
##
##    proc onProgressChanged(total, progress: BiggestInt,
##                           speed: float): Future[void] =
##      echo("Uploaded ", progress, " of ", total, " bytes")
##      echo("Current speed: ", speed, " kb/s")
##
##    proc main() {.async.} =
##      var ftp = newAsyncFtpClient("example.com", user = "test", pass = "test")
##      await ftp.connect()
##      await ftp.store("file.txt", "/home/user/file.txt", onProgressChanged)
##      echo("File finished uploading")
##    waitFor(main())


import asyncdispatch, asyncnet, nativesockets, strutils, parseutils, os, times
```

```
from net import BufferSize

type
  AsyncFtpClient* = ref object
    csock*: AsyncSocket
    dsock*: AsyncSocket
    user*, pass*: string
    address*: string
    port*: Port
    jobInProgress*: bool
    job*: FtpJob
    dsockConnected*: bool

  FtpJobType* = enum
    JRetrText, JRetr, JStore

  FtpJob = ref object
    prc: proc (ftp: AsyncFtpClient, async: bool): bool {.nimcall, gcsafe.}
    case typ*: FtpJobType
    of JRetrText:
      lines: string
    of JRetr, JStore:
      file: File
      filename: string
      total: BiggestInt         # In bytes.
      progress: BiggestInt      # In bytes.
      oneSecond: BiggestInt     # Bytes transferred in one second.
      lastProgressReport: float # Time
      toStore: string           # Data left to upload (Only used with async)

  FtpEventType* = enum
    EvTransferProgress, EvLines, EvRetr, EvStore

  FtpEvent* = object             ## Event
    filename*: string
    case typ*: FtpEventType
    of EvLines:
      lines*: string             ## Lines that have been transferred.
    of EvRetr, EvStore:          ## Retr/Store operation finished.
      nil
    of EvTransferProgress:
      bytesTotal*: BiggestInt    ## Bytes total.
      bytesFinished*: BiggestInt ## Bytes transferred.
      speed*: BiggestInt         ## Speed in bytes/s
      currentJob*: FtpJobType    ## The current job being performed.

  ReplyError* = object of IOError

  ProgressChangedProc* =
    proc (total, progress: BiggestInt, speed: float):
      Future[void] {.closure, gcsafe.}

const multiLineLimit = 10000

proc expectReply(ftp: AsyncFtpClient): Future[TaintedString] {.async.} =
  var line = await ftp.csock.recvLine()
  result = TaintedString(line)
  var count = 0
  while line[3] == '-':
    ## Multi-line reply.
    line = await ftp.csock.recvLine()
    string(result).add("\n" & line)
    count.inc()
    if count >= multiLineLimit:
      raise newException(ReplyError, "Reached maximum multi-line reply count.")

proc send*(ftp: AsyncFtpClient, m: string): Future[TaintedString] {.async.} =
  ## Send a message to the server, and wait for a primary reply.
  ## ``\c\L`` is added for you.
  ##
  ## **Note:** The server may return multiple lines of coded replies.
  await ftp.csock.send(m & "\c\L")
  return await ftp.expectReply()

proc assertReply(received: TaintedString, expected: varargs[string]) =
  for i in items(expected):
    if received.string.startsWith(i): return
  raise newException(ReplyError,
                     "Expected reply '$1' got: $2" %
                      [expected.join("' or '"), received.string])

proc pasv(ftp: AsyncFtpClient) {.async.} =
  ## Negotiate a data connection.
  ftp.dsock = newAsyncSocket()

  var pasvMsg = (await ftp.send("PASV")).string.strip.TaintedString
  assertReply(pasvMsg, "227")
  var betweenParens = captureBetween(pasvMsg.string, '(', ')')
  var nums = betweenParens.split(',')
  var ip = nums[0 .. ^3]
  var port = nums[^2 .. ^1]
  var properPort = port[0].parseInt()*256+port[1].parseInt()
  await ftp.dsock.connect(ip.join("."), Port(properPort))
  ftp.dsockConnected = true

proc normalizePathSep(path: string): string =
  return replace(path, '\\', '/')
```

```
proc connect*(ftp: AsyncFtpClient) {.async.} =
  ## Connect to the FTP server specified by ``ftp``.
  await ftp.csock.connect(ftp.address, ftp.port)

  var reply = await ftp.expectReply()
  if string(reply).startsWith("120"):
    # 120 Service ready in nnn minutes.
    # We wait until we receive 220.
    reply = await ftp.expectReply()

  # Handle 220 messages from the server
  assertReply(reply, "220")

  if ftp.user != "":
    assertReply(await(ftp.send("USER " & ftp.user)), "230", "331")

  if ftp.pass != "":
    assertReply(await(ftp.send("PASS " & ftp.pass)), "230")

proc pwd*(ftp: AsyncFtpClient): Future[TaintedString] {.async.} =
  ## Returns the current working directory.
  let wd = await ftp.send("PWD")
  assertReply wd, "257"
  return wd.string.captureBetween('"').TaintedString # "

proc cd*(ftp: AsyncFtpClient, dir: string) {.async.} =
  ## Changes the current directory on the remote FTP server to ``dir``.
  assertReply(await(ftp.send("CWD " & dir.normalizePathSep)), "250")

proc cdup*(ftp: AsyncFtpClient) {.async.} =
  ## Changes the current directory to the parent of the current directory.
  assertReply(await(ftp.send("CDUP")), "200")

proc getLines(ftp: AsyncFtpClient): Future[string] {.async.} =
  ## Downloads text data in ASCII mode
  result = ""
  assert ftp.dsockConnected
  while ftp.dsockConnected:
    let r = await ftp.dsock.recvLine()
    if r.string == "":
      ftp.dsockConnected = false
    else:
      result.add(r.string & "\n")

  assertReply(await(ftp.expectReply()), "226")

proc listDirs*(ftp: AsyncFtpClient, dir = ""): Future[seq[string]] {.async.} =
  ## Returns a list of filenames in the given directory. If ``dir`` is "",
  ## the current directory is used. If ``async`` is true, this
  ## function will return immediately and it will be your job to
  ## use asyncdispatch's ``poll`` to progress this operation.
  await ftp.pasv()

  assertReply(await(ftp.send("NLST " & dir.normalizePathSep)), ["125", "150"])

  result = splitLines(await ftp.getLines())

proc fileExists*(ftp: AsyncFtpClient, file: string): Future[bool] {.async.} =
  ## Determines whether ``file`` exists.
  var files = await ftp.listDirs()
  for f in items(files):
    if f.normalizePathSep == file.normalizePathSep: return true

proc createDir*(ftp: AsyncFtpClient, dir: string, recursive = false){.async.} =
  ## Creates a directory ``dir``. If ``recursive`` is true, the topmost
  ## subdirectory of ``dir`` will be created first, following the secondmost...
  ## etc. this allows you to give a full path as the ``dir`` without worrying
  ## about subdirectories not existing.
  if not recursive:
    assertReply(await(ftp.send("MKD " & dir.normalizePathSep)), "257")
  else:
    var reply = TaintedString""
    var previousDirs = ""
    for p in split(dir, {os.DirSep, os.AltSep}):
      if p != "":
        previousDirs.add(p)
        reply = await ftp.send("MKD " & previousDirs)
        previousDirs.add('/')
    assertReply reply, "257"

proc chmod*(ftp: AsyncFtpClient, path: string,
            permissions: set[FilePermission]) {.async.} =
  ## Changes permission of ``path`` to ``permissions``.
  var userOctal = 0
  var groupOctal = 0
  var otherOctal = 0
  for i in items(permissions):
    case i
    of fpUserExec: userOctal.inc(1)
    of fpUserWrite: userOctal.inc(2)
    of fpUserRead: userOctal.inc(4)
    of fpGroupExec: groupOctal.inc(1)
    of fpGroupWrite: groupOctal.inc(2)
    of fpGroupRead: groupOctal.inc(4)
    of fpOthersExec: otherOctal.inc(1)
    of fpOthersWrite: otherOctal.inc(2)
    of fpOthersRead: otherOctal.inc(4)

```

```
  var perm = $userOctal & $groupOctal & $otherOctal
  assertReply(await(ftp.send("SITE CHMOD " & perm &
                  " " & path.normalizePathSep)), "200")

proc list*(ftp: AsyncFtpClient, dir = ""): Future[string] {.async.} =
  ## Lists all files in ``dir``. If ``dir`` is ``""``, uses the current
  ## working directory.
  await ftp.pasv()

  let reply = await ftp.send("LIST" & " " & dir.normalizePathSep)
  assertReply(reply, ["125", "150"])

  result = await ftp.getLines()

proc retrText*(ftp: AsyncFtpClient, file: string): Future[string] {.async.} =
  ## Retrieves ``file``. File must be ASCII text.
  await ftp.pasv()
  let reply = await ftp.send("RETR " & file.normalizePathSep)
  assertReply(reply, ["125", "150"])

  result = await ftp.getLines()

proc getFile(ftp: AsyncFtpClient, file: File, total: BiggestInt,
             onProgressChanged: ProgressChangedProc) {.async.} =
  assert ftp.dsockConnected
  var progress = 0
  var progressInSecond = 0
  var countdownFut = sleepAsync(1000)
  var dataFut = ftp.dsock.recv(BufferSize)
  while ftp.dsockConnected:
    await dataFut or countdownFut
    if countdownFut.finished:
      asyncCheck onProgressChanged(total, progress,
          progressInSecond.float)
      progressInSecond = 0
      countdownFut = sleepAsync(1000)

    if dataFut.finished:
      let data = dataFut.read
      if data != "":
        progress.inc(data.len)
        progressInSecond.inc(data.len)
        file.write(data)
        dataFut = ftp.dsock.recv(BufferSize)
      else:
        ftp.dsockConnected = false

  assertReply(await(ftp.expectReply()), "226")

proc defaultOnProgressChanged*(total, progress: BiggestInt,
    speed: float): Future[void] {.nimcall, gcsafe.} =
  ## Default FTP ``onProgressChanged`` handler. Does nothing.
  result = newFuture[void]()
  #echo(total, " ", progress, " ", speed)
  result.complete()

proc retrFile*(ftp: AsyncFtpClient, file, dest: string,
               onProgressChanged: ProgressChangedProc = defaultOnProgressChanged) {.async.} =
  ## Downloads ``file`` and saves it to ``dest``.
  ## The ``EvRetr`` event is passed to the specified ``handleEvent`` function
  ## when the download is finished. The event's ``filename`` field will be equal
  ## to ``file``.
  var destFile = open(dest, mode = fmWrite)
  await ftp.pasv()
  var reply = await ftp.send("RETR " & file.normalizePathSep)
  assertReply reply, ["125", "150"]
  if {'(', ')'} notin reply.string:
    raise newException(ReplyError, "Reply has no file size.")
  var fileSize: BiggestInt
  if reply.string.captureBetween('(', ')').parseBiggestInt(fileSize) == 0:
    raise newException(ReplyError, "Reply has no file size.")

  await getFile(ftp, destFile, fileSize, onProgressChanged)
  destFile.close()

proc doUpload(ftp: AsyncFtpClient, file: File,
              onProgressChanged: ProgressChangedProc) {.async.} =
  assert ftp.dsockConnected

  let total = file.getFileSize()
  var data = newString(4000)
  var progress = 0
  var progressInSecond = 0
  var countdownFut = sleepAsync(1000)
  var sendFut: Future[void] = nil
  while ftp.dsockConnected:
    if sendFut == nil or sendFut.finished:
      # TODO: Async file reading.
      let len = file.readBuffer(addr(data[0]), 4000)
      setLen(data, len)
      if len == 0:
        # File finished uploading.
        ftp.dsock.close()
        ftp.dsockConnected = false

        assertReply(await(ftp.expectReply()), "226")
      else:
        progress.inc(len)
```

```
        progressInSecond.inc(len)
        sendFut = ftp.dsock.send(data)

    if countdownFut.finished:
      asyncCheck onProgressChanged(total, progress, progressInSecond.float)
      progressInSecond = 0
      countdownFut = sleepAsync(1000)

    await countdownFut or sendFut

proc store*(ftp: AsyncFtpClient, file, dest: string,
            onProgressChanged: ProgressChangedProc = defaultOnProgressChanged) {.async.} =
  ## Uploads ``file`` to ``dest`` on the remote FTP server. Usage of this
  ## function asynchronously is recommended to view the progress of
  ## the download.
  ## The ``EvStore`` event is passed to the specified ``handleEvent`` function
  ## when the upload is finished, and the ``filename`` field will be
  ## equal to ``file``.
  var destFile = open(file)
  await ftp.pasv()

  let reply = await ftp.send("STOR " & dest.normalizePathSep)
  assertReply reply, ["125", "150"]

  await doUpload(ftp, destFile, onProgressChanged)

proc rename*(ftp: AsyncFtpClient, nameFrom: string, nameTo: string) {.async.} =
  ## Rename a file or directory on the remote FTP Server from current name
  ## ``name_from`` to new name ``name_to``
  assertReply(await ftp.send("RNFR " & nameFrom), "350")
  assertReply(await ftp.send("RNTO " & nameTo), "250")

proc removeFile*(ftp: AsyncFtpClient, filename: string) {.async.} =
  ## Delete a file ``filename`` on the remote FTP server
  assertReply(await ftp.send("DELE " & filename), "250")

proc removeDir*(ftp: AsyncFtpClient, dir: string) {.async.} =
  ## Delete a directory ``dir`` on the remote FTP server
  assertReply(await ftp.send("RMD " & dir), "250")

proc newAsyncFtpClient*(address: string, port = Port(21),
    user, pass = ""): AsyncFtpClient =
  ## Creates a new ``AsyncFtpClient`` object.
  new result
  result.user = user
  result.pass = pass
  result.address = address
  result.port = port
  result.dsockConnected = false
  result.csock = newAsyncSocket()

when not defined(testing) and isMainModule:
  var ftp = newAsyncFtpClient("example.com", user = "test", pass = "test")
  proc main(ftp: AsyncFtpClient) {.async.} =
    await ftp.connect()
    echo await ftp.pwd()
    echo await ftp.listDirs()
    await ftp.store("payload.jpg", "payload.jpg")
    await ftp.retrFile("payload.jpg", "payload2.jpg")
    await ftp.rename("payload.jpg", "payload_renamed.jpg")
    await ftp.store("payload.jpg", "payload_remove.jpg")
    await ftp.removeFile("payload_remove.jpg")
    await ftp.createDir("deleteme")
    await ftp.removeDir("deleteme")
    echo("Finished")

  waitFor main(ftp)
```