**Path Module for Node**

**Presented by Kendall Abraham and Vivian Maganas**

**Intro:**

The core node module Path is a vital to creating portable code. It lets you manipulate and navigate through directories and their arrays without having to worry about which platform your code will be run on. To include this module, use require('path').

If you don’t want your code breaking on you, read on!

**Crash Course in Representation of Paths by operating systems:**

Unix-like Systems: ex. /foo/bar/baz/asdf.exe;

Root: space to left most slash

Directories: foo, bar, baz

Basename: asdf (could be directory or file, in this case it’s a file)

Extension: .exe

Directory Seperator: ‘/’

Path Seperator: ‘;’

Windows: ex. C:\\foo\\bar\\baz\\asdf.exe:

Root: C:

Directories: foo, bar, baz

Basename: asdf (could be directory or file, in this case it’s a file)

Extension: .exe

Directory Seperator: ‘\’

Path Seperator: ‘:’

**METHODS**

**Path normalize**

path.normalize() function normalizes a given path by collapsing (or removing) redundant separators and up-level references. It accepts only one argument which is a path. It takes a path containing ‘..’ or ‘./’ and returns it without any of the ‘..’ or ‘./’.

ex. path.normalize('/foo/bar//baz/asdf/quux/..')

// returns

'/foo/bar/baz/asdf'

**Path Join**

path.join() joins all given paths and normalizes them. It accepts multiple arguments. All paths that need to be joined together are given as arguments.

ex. path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')

// returns

'/foo/bar/baz/asdf'

**Path Resolve**

path.resolve() returns an absolute path. It accepts multiple arguments. The last argument is the path to which a path needs to be calculated from another path. Any of the arguments before the last argument are optional and joined to form a path from which a path is calculated to the last argument. If the last argument is absolute, it will return the last this function returns a path calculated by joining all arguments at the end of the current working directory.

**Path Relative**

path.relative() calculates a relative path between two absolute paths. It requires two arguments. The first argument is the absolute path from which a relative path has to be calculated. The second argument is the absolute path to which a relative path has to be calculated.

ex. path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb')

// returns

'../../impl/bbb'

Path Dirname

path.dirname() returns the path of the parent directory of a given path. It requires only one argument which is the path whose parent directory is required.

ex. path.dirname('/foo/bar/baz/asdf/quux')

// returns

'/foo/bar/baz/asdf'

**Path isAbsolute**

Determines whether path is an absolute path. An absolute path contains the [root](http://www.computerhope.com/jargon/r/root.htm) [directory](http://www.computerhope.com/jargon/d/director.htm) and all other [subdirectories](http://www.computerhope.com/jargon/s/subdirec.htm) in which a file or folder is contained. The absolute path will always lead to the same location, regardless of the working directory (whereas the working directory affects the relative path).

path.isAbsolute('/foo/bar') // true  
path.isAbsolute('/baz/..') // true  
path.isAbsolute('qux/') // false  
path.isAbsolute('.') // false

path.isAbsolute('//server') // true  
path.isAbsolute('C:/foo/..') // true  
path.isAbsolute('bar\\baz') // false  
path.isAbsolute('.') // false

**Path Basename**

path.basename() returns the last part of a path which is usually the file name. It accepts two arguments – the first argument is the path whose last part needs to be returned, the second argument is optional and is the extension. If an extension is given, path.basename() will return the file name in a path without the specified extension.

ex. path.basename('/foo/bar/baz/asdf/quux.html')

// returns

'quux.html'

**path.extname(p)**

The purpose of this method is to return the extension of a path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns as ‘ ’.

Ex.

*Case 1: Returns from the last ‘.’ to the end of the string*

path.extname('picasso.html')

/ returns

'.html'

path.extname('index.picasso.md')

/ returns

'.md'

path.extname('picasso.')

/ returns

'.'

*Case 2: No ‘.’ in the last portion of the path, or the first character is ‘.’*

path.extname('picasso')

/ returns

''

path.extname('.picasso')

/ returns

''

**path.parse(pathString)**

Returns an object from a path string, breaking the path into its different components: root, directory, base, extension, name.

Ex.

*On \*nix (Unix-like system):*

path.parse('/home/user/dir/picasso.txt')

/ returns

{

root : "/",

dir : "/home/user/dir",

base : "picasso.txt",

ext : ".txt",

name : "picasso"

}

*On Windows*

path.parse('C:\\path\\dir\\picasso.txt')

/ returns

{

root : "C:\\",

dir : "C:\\path\\dir",

base : "picasso.txt",

ext : ".txt",

name : "picasso"

}

**path.format(pathObject)**

Returns a path string from an object. It is the result of path.parse.

*On \*nix (Unix-like system):*

{

root : "/",

dir : "/home/user/dir",

base : "picasso.txt",

ext : ".txt",

name : "picasso"

}

/ returns

'/home/user/dir/picasso.txt'

*On Windows*

{

root : "C:\\",

dir : "C:\\path\\dir",

base : "picasso.txt",

ext : ".txt",

name : "picasso"

}

/ returns

'C:\\path\\dir\\picasso.txt'

**METHODS**

**path.sep**

Each operating system uses a different file path separator for identifying and/or creating file paths.

UNIX-like systems use the forward slash, and Windows uses the backslash. Unfortunately, most Windows applications cannot handle the forward slash.

As a fix to the cross-platform path separator problem, Node provides the sep property of the path module. Sep is used when building directory paths because it determines what file separator is needed based on the OS. An example appears below:

*On \*nix:*

'foo/bar/baz'.split(path.sep)

/ returns

['foo', 'bar', 'baz']

*On Windows:*

'foo\\bar\\baz'.split(path.sep)

// returns

['foo', 'bar', 'baz']

**In this example, Node determines which separator is being used, where it occurs, and breaks the path into its directories.**

**path.delimiter**

The platform-specific path delimiter which looks for ‘;’ or ':,' and splits the path at each instance.

Each operating system uses a different path separator for identifying paths in a path environment variable. The path environment variable is a colon-separated list of directories in which the shell will look when it tries to find an executable file that matches a command name.

UNIX-like systems use the colon, and Windows uses the semi-colon.

Node provides the delimiter property of the path module to correctly parse out the paths for the path variable, regardless of the operating-system that it’s running from.

*On \*nix (‘:’):*

console.log(process.env.PATH)

// '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter)

/ returns

['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']

*On Windows (‘;’):*

console.log(process.env.PATH)

// 'C:\Windows\system32;C:\Windows;C:\Program Files\node\'

process.env.PATH.split(path.delimiter)

// returns

['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\node\\']

**path.posix**

Provide access to the path methods mentioned but always interact in a posix compatible way.

POSIX: Portable Operating System Interface, a set of standards specified for maintaining compatibility between operating systems which defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

**path.win32**

Provide access to path methods mentioned but always interact in a win32 compatible way.

Win32: the main set of Microsoft Windows APIs used for developing 32-bit applications, used for functions in things like diagnostics, security and Windows UI