Mango Quickref

Want to run a high-performance web-server with Servlet support?

Take a look at how unittest.testServletEngine() combines a ServletProvider (the servlet engine) with an HttpServer and a few instances of Servlet classes. It's very easy to build a web-server using servlets to construct dynamic content. For serving traditional static-pages, there's a convenient ServletResponse.copyTo() method that takes care of the grunt work.

Want to build your own HTTP-based server environment?

Look at how unittest.testHttpServer() binds a very simple HttpProvider to an HttpServer. You get much of the same functionality as a ServletProvider exposes, but at a slightly lower level.

Need to work with random-access files?

Unittest.testRandomAccess() provides an example of how this is done using FileConduit, and introduces formatted IO using the fundamental Reader & Writer pair.

Want to copy a file from one place to another (console; socket; other file)?

Unittest.testFile4() and testFile5() show you two ways to do this very efficiently. Mango.io prefers to read big chunks of data, and write in big chunks also. It happily handles cases where the output cannot accept as much as the input can read: e.g. file in, socket out.

Want to read & write formatted data on a memory buffer?

Mango.io supports both an iostream style syntax and a get/put syntax. See unittest.testBuffer() for a brief example of the two different styles. Formatted IO is supported via a gaggle of Readers and Writers.

Want to write a buffer without using a Writer?

Unittest.testAppend() bypasses the use of a Writer and appends directly to a buffer. This can be very handy for efficiently constructing strings. Use Buffer.toString() to extract the valid readable content from the buffer. Module mango.io.FilePath uses this approach to construct char[] output.

Need to read text as tokens?

Take a look at unittest.testToken1(), testToken2() and testToken3(). They show different ways to convert text-based input into a stream of tokens, performing data conversion where necessary.

Want to use Printf() for formatted output?

See unittest.testPrintf() for an example of how to bind Printf() onto an buffer. Note that Printf() uses an additional buffering mechanism.

Concerned about overhead involved with buffering?

Unittest.testDirectIO() shows how to perform traditional direct-IO to a local area of memory: performed by mapping a Buffer onto local memory instead of the heap.

How about memory-mapped IO on huge files?

FileConduit provides a method to bind an IBuffer onto a memory-mapped file, which can then be used to directly manipulate files up to 4GB in size (much larger on 64bit systems). Read/Write access follows the mode set for the file itself. Currently supported on Win32 platforms only. See unittest.testMappedFile() for an example.

Want to easily Read and/or Write the data from your own classes?

Use the IReadable and IWritable interfaces to make your own classes blend into the mango.io framework. See unittest.testClassIO() and unittest.testCompositeIO() for how to apply these interfaces. The latter uses a mechanism designed to extend up to transactional boundaries, whereas the former exposes the raw framework.

Need to serialize your classes (onto disk, or elsewhere) and then reconstruct them?

Unittest.testClassSerialization() shows what you need to get going. Classes are serialized using the IReadable and IWritable interfaces, but are processed to ensure their binary format will be interchangeable across machine boundaries.

Classes are later reconstituted via a combination of a Registry and a bit of support from the serializable class itself (each serializable class must expose a factory for itself).

Want to read a file (or any Conduit) as a set of lines?

Take a look at unittest.testFile1(), testFile2() and testFile3(). They show various methods to tokenize and process line-oriented content. While these examples all use a file as input, you can use any other Conduit type (sockets, console, etc.)

Need to recursively scan files and directories in a file-system?

Unittest.testScanFiles() has an example of how to use FilePath, FileProxy, and filelist-filtering. FilePath represents a parsed path, making it easy to extract various components or blend multiple paths together. FileProxy is a representative for a physical file, and supports operations such as isDirectory(), toList(), getModifiedTime(), getSize(), create(), remove() etc.

See unittest.testFileSize(), testFilePath(), and testFileList() for related examples.

How about writing output aligned to a set of columns?

There's a simple ColumnWriter that outputs each element to a particular column position, and resets itself when a newline is emitted. You can use this for output to files, console, or whatever.

Want to read input from the console?

The basic procedure is shown in unittest.testStdin(), though you'd likely use a tokenizing approach instead. Note that console input always waits for a CR.

Want to use regular expressions to tokenize input from a variety of sources?

Take a look at unittest.wordRegex(), testFileRegex() and testSocketRegex() to see how they obtain and process data. RegexTokenizer wraps the std.regexp module.

How about reading pages from the Internet?

The basic example is in unittest.testSocket(), which grabs the introduction page from www.digitalmars.com. Note that mango.io supports timeout on Internet reads. Extended examples include unittest.testHttpClient(), testHttpClient2() and testHttpClient3(). The latter introduces the HttpClient class, which takes care of many aspect of client-side HTTP processing (headers, parameters, response-line, etc). See unittest.testUri() for URI support.

Need a local cache of D objects?

You might consider trying the approach exposed via unititest.testQueuedCache(), which uses a MRU algorithm to ensure that popular items are made available at the expense of less popular items. Cached items should implement ICacheEntry, or extend CacheEntry. Note that cache entries are serializable also.

How about a level-1/level-2 cache of D objects?

Mango.cache provides a virtual-cache, whereby LRU objects are serialized out to disk, and then resurrected as necessary. For quantities of immutable objects, or low-value content, this can offload a tremendous strain from, say, a database server. See unittest.testVirtualCache() for a basic example.