the wiz book

Torben Schinke

Version 0.1, 2018-02-04

Table of Contents

Dedication

-

Preface

The idea of a robust, simple and scalable storage format superseeding the lowest denomiator filesystems, fascinated me already 15 years ago, however I never had the opportunity to actually start implementing such a project. When the time came, I started to design a paper based specification in 2015 which performs well for deduplicating large files, nested directory trees and continues snapshots. To solve the typical problems of a *multi file based document format* at work, I created a proprietary java based implementation from it, called wiz - which is just the opposite of a git, similarities are purely coincidental. For the original intention, it worked pretty well. But as requirements changed, the performance for a lot of additional use cases was disappointing. The main performance issues are caused by both, inherent format decisions and the necessity of a complex virtual machine. In practice, the latter caused also penalties on the probably most successful mobile platform of our time. To solve all of these issues I started to design an entirely new specification which addresses all of the new additional scenarios (and even more). Hereafter this new specification is actually *wiz version 3* or simply *wiz*. Therefore the proprietary existing wiz implementation is called *legacy wiz* and is not only implemented in a different language but also does a lot of things differently to improve performance, storage usage, reliability and system complexity. Today, the market for closed source commercial software libraries is nearly dead and gaining money or finding acceptance is not easy. Usually large companies dominate the market with a lot (but definity not all) high quality products.

Format specification

Wiz is both, an implementation and a specification. In this chapter only the specification matters and is described in a way that it can be implemented in any language and ecosystem.

Everything in the world of wiz is represented by a node, which always starts with a byte identifier. Besides that, there are no other common properties among node types.

Some nodes support compression for their payload, but it is not a generic feature. In contrast to that, encryption is a property of the pageing infrastructure itself. I believe that providing unencrypted insight into meta information is already an absolute security flaw. Examples for this are stacked filesystems like EncFs or eCryptfs, which provides plain information about the folder structure, amount of files and the file sizes. Keeping facts about nodes unencrypted and just encrypt payload would be the same kind of security flaw.

Pointers to nodes have a variadic size from 2 bytes upto 20 bytes, which boils down to a theoretical 128 bit addressing, in which the first 10 bytes refer to the id of a virtual device (vdev) and the last 10 bytes to a physical offset. The additonal two bytes are caused by the overhead of the varint LEB128 encoding. The vdev is usually a simple file in your host's filesystem. Using this technique, the pointer adapts itself to different use cases, like many small vdevs or a single large file. The usage of a varint favors storage costs over performance. Addressing millions of nodes, especially in independent vdev sets - I think of an online mirror storage - would waste a lot of space. In this scenario, one could argue that the performance penalties are also on the client side and will not stress the server at all. There are also use cases in which a high fragmentation is caused. In such scenarios it is cheaper to resolve physical addresses over another indirection. For this there are a

number of reserved *vdev* ids.

All texts should be encoded in UTF-8, however there is no simple answer for filepaths. When taking a look at Git and how it handles filenames, it works perfectly on Linux filesystems, which just treat a filename as a byte sequence but fails miserably when mixing platforms like MacOS and Windows. Some users expect an UTF-8 normalization and others excoriate that. So we also keep that decision to a preprocess. Moreover, instead of following the errornous c approach of zero termination, we also use a varint as a length encoding, which has the same overhead for most expected strings (up to 127 byte, runes may vary). We define a text payload always as UTF-8 (a varint prefixed array of uint8) and undefined text payload as a varint prefixed array of uint8.

Nodes need not to be aligned to physical sectors, but if you do so, one can expect better performance and recoverability but you will sacrifice space efficiency. Consider a 4k sector alignment for todays devices.

In general, byte order is the common network order - big endian. Most elements are already invariant to endianess, like varint or arrays. There is no expectation that a specialization to the host endianess will actually result in any performance gain, for an implementation of this specification.

In the following, offsets are always defined in hexadecimal values prefixed by 0x. Fixed array values are also declared using hex values, but with 0x omitted.

Table 1. specification of types

Size	Type	Description
1-10	varuint	unsigned, as defined in [varint]
1-10	varint	signed, as defined in [varint]
1-10	vdev id	varint id, must be unique within a vdev set
2-20	ndptr	a node pointer is a double varint 128 bit address, where the first 10 byte determine the vdev id and the last 10 byte determine the physical offset in the vdev.
1-*	utf8	UTF-8 sequences are encoded with a prefix of the type *varuint* followed by an arbritary amount of bytes
2-*	wbo	a wbo serialized object

Table 2. examples of varuint encodings using LEB128

Dezimal	Length	Binary
1	1	00000001
127	1	01111111
128	2	10000000 0000001
16383	2	11111111 01111111
16384	3	10000000 10000000 00000001
32767	3	11111111 11111111 00000001
65535	3	11111111 11111111 00000011
2147483647	5	11111111 11111111 11111111 11111111 00000111
4294967295	5	11111111 11111111 11111111 11111111 00001111
1099511627520	6	10000000 11111110 11111111 11111111 11111111 00011111

Dezimal	Length	Binary
17592186040320	7	10000000 11100000 11111111 11111111 11111111 11111111 00000011
9223372036854775807	9	11111111 11111111 11111111 11111111 11111111 11111111 11111111 01111111
18446744073709551615	10	11111111 11111111 11111111 11111111 11111111 11111111 11111111 00000001

As you can see from the example encoding table table one can expect that most *ndptr* values will be in the size of 1 for the *vdev* and 4-6 for the offset, so expect an average size of 5-7 byte per pointer. So in contrast to a naive encoding of a full 128bit pointer we only need 30% to 43% of the space which is even less than for an uncompressed 64bit pointer.

You can also use virtual addresses when using the reserved *vdev* id. This makes things like defragmentation and sector relocation a lot easier but introduces a significant overhead.

Table 3. Reserved vdev identifiers

Value	Description
0x00	Refers to the unique lookup table to resolve virtual node ids/addresses to physical ones.
0x01…0xF	Reserved for future use.

WBO specification

The *wiz binary object* serialization format is specified by the following BNF like declaration. It is somewhat comparable to the BSON format (see [bson]) but uses the packed varint format from above to improve space efficency. Due to the copy-on-write approach, we do not plan to update a distinct data field within a written structure. BSON cannot guarantee that either when increasing the length of a string.

Table 4. Pseudo BNF, types as uint8 in quotes

object ::= varuint varuint field_list	a WBO starts with the total object length in bytes (including nested objects), followed by the amount of field entries and the actual field_list
field_list ::= field field_list	the recursive definition
field_name ::= varuint (uint8*)	a varuint declares the number of (UTF-8) bytes to follow
field ::=	
"0x00" field_name uint8	byte / uint8
"0x01" field_name uint16	uint16
"0x02" field_name uint32	uint32
"0x03" field_name uint64	uint64
"0x04" field_name int8	int8
"0x05" field_name int16	int16
"0x06" field_name int32	int32
"0x07" field_name int64	int64
"0x08" field_name float32	float32
"0x09" field_name float64	float64
"0x0A" field_name complex64	complex64
"0x0B" field_name complex128	complex128

"0x0C" field_name varuint (uint8*)	a varuint declares the number of UTF-8 bytes to follow
"0x0D" field_name varuint (uint8*)	a varuint declares the number of bytes to follow
"0x0E" field_name varuint	a variable length unsigned integer in LEB 128 format (1 - 10 bytes)
"0x10" field_name varint	a variable length signed integer in LEB 128 format (1 - 10 bytes) with zigzag encoding
"0x11" field_name varuint varuint	the vdev id of two variable unsigned length integers
"0x13" field_name varuint type (type content bytes*)	an array with the bytes of the according type to follow. E.g. could be a list of float32 or object.
"0x14" field_name object	a field containing another (recursive) object definition

Record format

As already pointed out, the actual format is represented as nodes. However also nodes are organized within a higher level structure, named *record*. Records are used to introduce other properties which should be common to all nodes, like generic checksums, extra redundancy, encryption or message authentication. The first node, the *magic node*, contains all relevant parameters for the record configuration, including the first node itself. The type of a record is always the same for the entire *vdev-set*, and therefore needs not any overhead, like headers or size attributes. Currently only records of a fixed size of 4096 byte are allowed. This size is used primarily to match today's physical storage devices which works with 4k sectors. Also a lot of legacy filesystems are driven with 4k sectors, so this looks like a good fit. Actually there seems to be no real reason to

limit the record size, so it is a runtime configuration and this specification may be extended in the near future. The primary goal of a record is to verify that the contained data has not been tampered, either by a machine failure like random bitrots or due to a real attacker. Remember that protecting against attackers is only possible by using a *hmac* or similar authentication techniques. An implementation has to verify the record before evaluating the contained nodes. An implementation should provide recovery methods based on records by at least ignoring the defunct one and continue reading the next record. Due to the fixed record size but with different overhead, the actual payload size is different from type to type. The common property is that each record starts with a node and contains at least a valid one. However it is expected to put as many nodes into a record as possible to improve space efficiency. The minimal file size of a valid *vdev* depends on the record size and is at least one record.

Table 5. Record format type 0: This is the fastest and simplest record format.

Order	Size	Description
0	4098	The payload
1	4	A CRC32 value to detect some simple bit failures

Table 6. Record format type 1: This is a variant with a configurable 256 bit cryptographic hash.

Order	Size	Description
0	4064	The payload
1	32	The configured hash of 256 bit

TBD: encryption, hmac, reed solomon

Magic node

Marks a container and must be always the first node of a file and should not occur once again. If it does (e.g. for recovery purposes), it is not allowed to be contradictory. Wiz containers can simply be identified using the magic bytes *[00 03 77 69 7a 63]*.

Table 7. on-disk format of the magic node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x00	type *header*
0x01	4	uint32	version	0x03	this is the third version of the wiz format
0x05	4	[]uint8	magic	[77 69 7a 63]	the magic header value *wizc* for the container
0x06	1	uint8	encryption type	*	the kind of encryption algorithm for the pages
0x07	*	utf8	sub magic	*	the user defined sub magic header value as varuint prefixed UTF-8
#5	16	UUID	wiz file set identifier	*	the UUID of this wiz storage. Any vdev id and therefore ndptr is only valid within the same set of wiz files sharing the same UUID.
#6	1-10	varuint	vdev id	*	The unique vdev id of this wiz file within the file set. Should start with 0.

The *version* indicates which nodes and how they are defined. A node format may be changed in future revisions but should be extended in a backwards compatible manner. If such a thing is not possible (e.g. also by adding new kinds) the number increases. Because the

format depends on the node kind (and therefore the sizes to parse) an outdated reader can actually only use it's recovery options to continue reading.

Some notes to the version flag: Actually this is the third generation of the wiz format. The first only existed on paper, the second was implemented largely based on the paper based specification but is proprietary. So this is the first which is now open source. It is not only implemented in a different language but also does a lot of things differently to improve performance, storage usage, reliability and system complexity.

One of the basic ideas of wiz is to replace custom *on disk formats* with something better. Today, probably the most widespreaded format is the zip file format from pkware. Amongst others, it is used by the entire Microsoft Office suite for their **x files*. To easily identify such subformats, the wiz header defines an UTF-8 subformat specifier. In the following table one can see a list of known sub format identifiers. If you create your own identifier, use your reversed company or product internet domain, e.g. *com.mycompany.myproduct* to minimize collisions. You may also invent your own file extension, but as a rule of thumb, you should never rely on it and check the magic node instead.

Table 8. known sub format identifiers

Value	Description
0x04 [77 69 7a 61]	*wiza* the standard archive format of the command line tool
0x04 [77 69 7a 62]	*wizb* the format of the backup tool

The encryption formats are defined as follows:

Table 9. encryption format identifiers

Value	Description
0x00	no encryption, all nodes are written as they are, just in plain bytes
0x01	AES-256 CTR mode

See the encryption chapter for the detailed specification of each encryption mode.

A wiz storage may consist of multiple files or devices, which have each their own magic node but a unique vdev id. Any *ndptr* contains also that id, so referred nodes can be spreaded across vdevs. Use cases for this may be to improve performance, to create append-only / WORM (write once read many) storages or simply to attach additional storage volumes. To detect which vdevs belong to the same vdev set, a unique UUID is assigned to each set. You should not rely on a file name to identify a set, if the user has access to the files.

Choose wisely your trade-of when considering (large) file sets, especially when dealing with end users. A common expectation is that an application stores a document always in a single file.

It is a hard descision where to write and update the *super node*. Depending on the use case it is either unrealistic (linear growing amount of vdevs) or even impossible (WORM) to update existing vdevs, hence there is no definitive rule here.

Each application has to define where to write or update the *super node*.

In order to alleviate the situation, there are some well defined use cases. If a type matches your use case, apply one of the following rules.

Type 1

For single file formats (ever a single vdev) always update the ringbuffer.

Type 2

A performance optimized stripe vdev set (like RAID 0) only updates the ring buffer in the vdev with the lowest number (typical 16). Stripe sets are wobbly anyway. So actually *Type 1* is only a special case of a stripe set with a single vdev.

Type 3

For redundant vdevs (like mirrors / RAID 1 / RAID 5) always update the ringbuffer in every vdev.

Type 4

For WORM / append-only formats only write a new super node to the added vdev and never change an already written file.

Configuration node

The wiz repository (as defined by the file) may include different properties. These properties are important to open the repository properly, e.g. picking the correct hash algorithm. The hash algorithm has a fixed length, but not a fixed algorithm. However the algorithm configured is valid for the entire vdev set and must not change

between vdevs. It will be used for all hashed data structures. The configuration also may contain persistent optional settings for tweaking, which are represented in the wbo. This node directly follows the magic node.

Table 10. on-disk format of the configuration node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x01	type *configuration*
0x01	1	uint8	hash algorithm	*	the hash algorithm to use, which must always be 256 bits / 32 byte in length
0x02	1 - 10	varuint	reserved	*	the reserved space for the wbo object.
0x01	*	wbo	configuration	*	key value properties in wbo format

By default, the reserved space for the *wbo* should be the difference between the actual size of the magic node and the first physical sector at offset 0x1000. However, a writer may decide to ignore that and not to provide any reserved space or even provide more sectors.

A configuration node should provide some space to allow changes to the *wbo*, so that permanent changes are possible without rewriting the entire file.

In general, the configuration node is not intended to be modified on a regular basis, and therefore there is no infrastructure to provide any resilence here. The settings here are intended to be written either at creation time or for recovering or debugging purposes.

Table 11. hash algorithm identifiers

Value	Description
0x00	SHA-256
0x01	SHA-512/256
0x02	SHA3-256

Super node

The super node is a ring buffer having a variable amount of transaction entries which are written in a round-robin manner. The minimum valid capacity is 1 and the maximum amount if 255. The larger the ring buffer, the more possiblities to recover older states are available. Consider e.g. a capacity of 128 for single file formats, but 1 when appending only new immutable vdevs. Otherwise provide at least the space for 2 nodes. The transaction node with the highest transaction id and a valid checksum is the transaction node to use. If something went wrong, older transactions may be used for recovery, but the usefulness depends on the kind of damage. Usually one would expect that if the transaction is written to the ring buffer and the underlying file system crashes, it hopefully will loose the data in the same order (the transaction node is always the last thing written), however there is no guarantee on that. Also fsync cannot protect us from that, because it is broken on many filesystems, even by design (see also [btrfs-fsync]). Today, I don't know how to solve that properly.

To get the best resilence, you should never overwrite any data and instead create a new vdev for every transaction and fsync the file contents and the directory in the right order.

The super node must be the third node after the *configuration node* and should be located at file offset 0x1000. But remember, that

depending on the reserved space of the wbo in the configuration node, there is no guarantee for that.

The super node is rewritten for each transaction and has a high write amplification. It should always match the physical addressing of the file system or the raw device to optimize performance.

Table 12. on-disk format of the super node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x02	type *super*
0x01	1	uint8	size	*	# entries in ring buffer as *n*
0x02	*n* * sizeof(tx-node)	[]tx-node	array	*	ring buffer of *n* transaction nodes

Transaction node

The transaction node is the entry point which defines an applied transaction and all references to nodes which describe the valid state of the entire storage. When applying changes to the storage all changes are made using COW (copy on write) techniques. Even a simple delete will cause a write cascade, from a leaf to the root, to represent the change. Afterwards the new commit is referenced by a new transaction node. As soon as the transaction node has made it to disk, at least the predecessor still points to a valid state, however the pre-predecessor may now point to overwritten data, so the possibilities of recovery are limited (comparable to [zfs-magic]), due to the usage of free areas as declared by the *free space tree*. Note

that a writer may implement various algorithms to lower fragmentation by deferring writes and by prefer writing to new areas instead of filling holes. Also a writer may defragment storage by rewriting nodes and updating all related *ndptrs*, which is probably one of the most expensive operations. On the other side one can use virtual addresses. But keep in mind, that using the reserved *vdev* for the indirect address table, lookups will double the amount of required in-memory space and doubles the amount of initial I/O to resolve values from disk, which slows down everything else. Depending on the use case, this may be a good choice to support faster defragmentation.

The transaction id is a strict monotonic number.

Table 13. on-disk format of the transaction node

Order	Size	Type	Name	Value	Description
#0	1	uint8	node type	0x03	type *transaction*
#1	8	uint64	transaction id	*	increasing number. If the id overflows, all preceeding transactions are simply zeroed out, to invalidate them.
#2	16	ndptr	vtable tree	*	reference to the virtual address table. If the offset (the last 8 byte) are 0x00, no virtual addresses are in use yet (and have never been used) or when disabled.
#3	16	ndptr	commit tree	*	an uncompressed 128 bit node pointer to the tree of named commits (tags or branches). If the value is 0x00 there is no tree yet and the storage is empty.

Order	Size	Type	Name	Value	Description
#4	16	ndptr	free space tree	*	an uncompressed 128 bit node pointer to the tree of free areas. A value of 0x00 indicates no free space, e.g. when newly created or when disabled.
#5	16	ndptr	reference count tree	*	an uncompressed 128 bit node pointer to the tree of reference counts. A value of 0x00 indicates that there is no reference tree yet, e.g. when newly created or disabled.
#6	16	ndptr	flat checksum tree	*	an uncompressed 128 bit node pointer to the tree of checksums for each node. This just keeps simple hash values of each node, to detect corruptions. This is not a hash tree. Is 0x00 if disabled.
#7	16	ndptr	hash tree	*	an uncompressed 128 bit node pointer to the hash tree. This is a merkle tree e.g. used for blockchain features or other use cases. Is 0x00 when disabled. In contrast to a simple hash of the entire node, it is calculated on the logical content (children hashes) and not any actual pointer values.
#8	32	hash	checksum	*	the hash of fields #0 - #7

As you can see, the size of a transaction node comes at 121 byte. This size needs not to be discussed because the amount of

transaction nodes is constant and the minimal size should be 2 anyway. In a perfect world, this will protect us when we get interrupted while writing the transaction. There

Node relation overview

The following figure illustrates the basic node relations. The *vtable tree*, *free space tree*, *ref count tree*, *flat checksum tree* and *hash tree* are all optional. Values of 0x00 in the transaction node signals that no such tree has been defined yet. However if a certain tree should be used, is configured through the *wbo* in the configuration node. The optional trees are possibilities to optimize certain use cases but are just bloat for others.

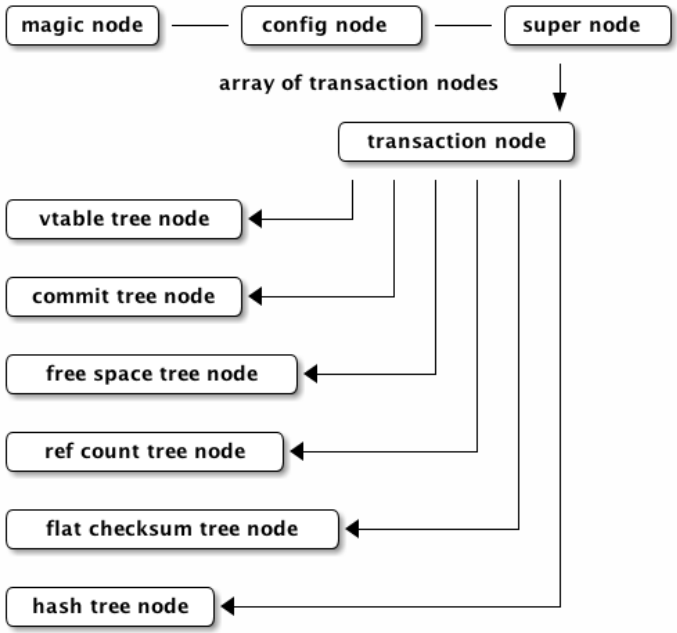


Figure 1. Node order and references within a vdev

The *wbo* configuration options are defined as follows.

Table 14. wbo configuration for optional transaction trees

Name	Type	Description
vtable_tree	bool	true if *ndptr* should be virtual, false if they should be direct
space_tree	bool	true if freed memory segments are tracked using this tree
reference_count_tree	bool	true if nodes should be reference counted

Name	Type	Description
flat_checksum_tree	bool	true if nodes are checksumed in a flat way
hash_tree	bool	true if nodes are hashed using a merkle tree

Commit node

A commit incorporates a bunch of parent commits, a list of named trees, a message and a unix timestamp. The most important thing is that it does never contain pointers but the hashed values of the trees and parents. When calculating the hash of a node it should be always prefixed (e.g. type) and postfixed (e.g. length) as it is done by e.g. git and recommended by the Sakura hash tree mode (see [sakura]) to create a strong hash tree and to form a merkle tree.

Note that depending on the chosen data, stream nodes are not hashed directly and therefore are not part of the merkle tree. This is an explicit design decision to give writers the freedom to distribute data nodes at will to e.g. improve copy-on-write efficiency or e.g. remote delta updates by desired redundancy in different pack files.

Table 15. on-disk format of the commit node

Order	Size	Type	Name	Value	Description
#0	1	uint8	node type	0x04	type *commit*
#1	8	int64	timestamp	*	milliseconds since epoch. This is considered as a hint for humans, not for the system. Timed order is defined by referring to parent commits.

Order	Size	Type	Name	Value	Description
#2	1-*	varuint	len	*	Length of the message array in bytes as *len*
#3	*len*	[]uint8	payload	*	byte array containing a message payload. The format of this message is part of the application's domain and not specified, e.g. it may just be an utf8 string, but also a binary blob.
#4	1-*	varuint	count	*	Amount of trees in this commit as count
#5	*count** named tree	[]named tree	trees	*	sorted list of trees, each with a byte array as name, followed by it's hash.

A named tree is just a varuint prefixed byte array as the key, followed by the 32 byte of the hash. The key (or name) may be an arbitary byte sequence, but it makes sense to keep it as an UTF8 string.

Table 16. on-disk format of a named tree

Order	Size	Type	Name	Value	Description
#0	1-*	varuint	count	*	length of the key as *len*
#1	*len*	[]uint8	payload	*	the key or name of the following tree
#2	32	hash	hash of tree	*	the hash of the referenced tree

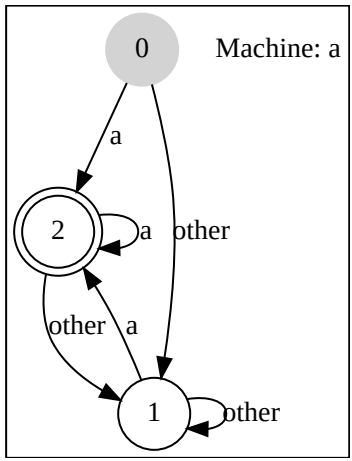
In the following some reserved and predefined keys are explained. The *f* or 0x66 is used as the default *filesystem* anchor. The *r* or 0x72 is used as the default *relational* anchor, using the same hierarchical key/value logic as the filesystem. Relational data is stored in the *wbo*

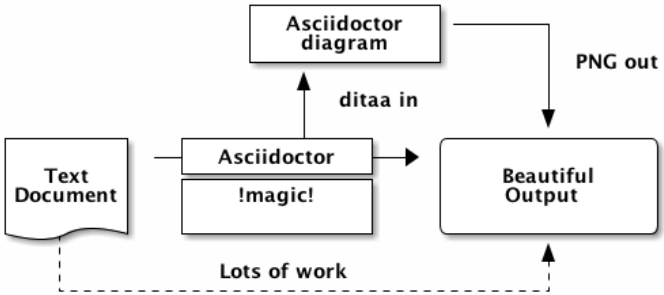
format without any schema.

Table 17. known keys for named trees

Hex	ASCII	Description
0x66	f	the default filesystem tree
0x72	r	the default relational tree

TODO: separate merkle tree in the transaction seems not to make sense and is not worth the effort.




Asciidoctor
diagram
PNG out
ditaa in
Text
Document
Asciidoctor
!magic!
Beautiful
Output
Lots of work

Example Bibliography

The bibliography list is a style of AsciiDoc bulleted list.

Books

- [taoup] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley. ISBN 0-13-142901-9.
- [walsh-muellner] Norman Walsh & Leonard Muellner. *DocBook - The Definitive Guide*. O'Reilly & Associates. 1999. ISBN 1-56592-580-7.
- [zfs-spec] http://www.giis.co.in/Zfs_ondiskformat.pdf
- [btrfs-fsync] https://btrfs.wiki.kernel.org/index.php/FAQ#Does_Btrfs_have_data.3Dordered_mode_like_Ext3.3F
- [varint] https://developers.google.com/protocol-buffers/docs/encoding
- [bson] http://bsonspec.org/spec.html
- [zfs-magic] https://blogs.oracle.com/ahrens/is-it-magic
- [sakura] https://keccak.team/files/Sakura.pdf

Articles

- [abc2003] Gall Anonim. *An article*, Whatever. 2003.

Example Glossary

Glossaries are optional. Glossaries entries are an example of a style of AsciiDoc labeled lists.

A glossary term

The corresponding (indented) definition.

A second glossary term

The corresponding (indented) definition.

Example Colophon

Text at the end of a book describing facts about its production.

Example Index