Data Types

XINA has a fixed set of **data types** which apply to attributes and fields. They are intended to provide consistent behavior across MySQL, Java, and JavaScript data types.

Numeric Types

Numeric data types form the backbone of most XINA data sets. Particularly in high data volume environments it is worth considering the smallest byte size type needed, as storage savings and query performance impacts can be considerable.

int(n)

Signed integer values:

Type	Java	MySQL	JavaScript	Notes
`int(1)`	`byte`	`tinyint`	`number`	1 byte signed integer, -2^7 to 2^7-1
`int(2)`	`short`	`smallint`	`number`	2 byte signed integer, -2^{15} to $2^{15}-1$
`int(4)`	`int`	`int`	`number`	4 byte signed integer, -2^{31} to $2^{31}-1$
`int(8)`	`long`	`bigint`	`number`	8 byte signed integer, -2^{63} to $2^{63}-1$??

> ?? In JavaScript number primitives are stored as an 8 byte float, so only -2^{53} to $2^{53}-1$ is available with exact precision when working in a JavaScript client (including the XINA web application).

An error is returned if a value provided is outside the range a required type, or contains a fractional value.

float(n)

Standard floating point values:

Type	Java	MySQL	JavaScript	Notes
`float(4)`	`float`	`float`	`number`	IEEE 754 4 byte floating point
`float(8)`	`double`	`double`	`number`	IEEE 754 8 byte floating point

An error will be returned if a floating point value is provided outside the representable magnitude of the required type. (Some languages denote this as `Infinity` / `-Infinity`, but this is not supported in XINA). XINA also does not recognize a `NaN` (not a number) floating point literal.

Boolean Type

Simple `true` / `false` type.

Type	Java	MySQL	JavaScript
`boolean`	`boolean`	`tinyint`	`boolean`

Note that MySQL treats `0` as `false`, non-zero as `true`.

Character Types

Character data types are used to store text information.

Type	Java	MySQL	JavaScript	Notes
`utf8string(n)`	`string`	`char(n)`	`string`	n up to 128, uses `n*4` bytes, normalized
`utf8vstring(n)`	`string`	`varchar(n)`	`string`	n up to 128, uses up to `n*4` bytes, normalized
`utf8string`	`string`	`mediumtext`	`string`	up to 2^{24} bytes, normalized
`utf8text`	`string`	`mediumtext`	`string`	up to 2^{24} bytes, not normalized
`utf8filename`	`string`	`varchar(255)`	`string`	uses up to 255 bytes, file-safe characters only
`asciistring(n)`	`string`	`char(n)`	`string`	n up to 256, uses n bytes, normalized
`asciivstring(n)`	`string`	`varchar(n)`	`string`	n up to 256, uses up to n bytes, normalized
`asciistring`	`string`	`mediumtext`	`string`	up to 2^{24} bytes, normalized
`asciitext`	`string`	`mediumtext`	`string`	up to 2^{24} bytes, not normalized
`asciifilename`	`string`	`varchar(255)`	`string`	uses up to 255 bytes, file-safe characters only

Note, all string operations are **case-insensitive** by default. This can be overridden with the `COLLATE` expression by specifying a binary collation.

Character Encoding

XINA offers two types of character encoding:

UTF-8

UTF-8 is the recommended default encoding. It has a variable length of 1 to 4 bytes per character.

ASCII

ASCII is recommended for high volume datasets where storage or indexing is critical. It is a subset of UTF-8 with a fixed length of 1 byte per character.

SQL Types

MySQL provides several types for character data storage with different considerations.

char(n)

- data stored in the table
- fixed amount of space per row (n * max_bytes_per_character)
- fastest search and index
- requires most storage

varchar(n)

- data stored in the table
- variable amount of space per row (up to n * max_bytes_per_character)
- fast search and index

text

- data stored outside the table
- slowest search and index
- only as much storage as needed

Categories

string

Text is **normalized** before insertion:

- leading and trailing whitespace is trimmed
- all internal whitespace is reduced to a single space character

text

Text is kept exactly as input, preserving all whitespace.

filename

Text is **normalized** before insertion. Content is invalid if it contains common unsafe file characters or patterns. Specifically, it is checked against the following rules:

- 255 characters or less
- Must not end with a period
- Must not match a set of Windows-specific restricted names:
  - `(?!(aux|clock\\$|con|nul|prn|com[1-9]|lpt[1-9])(?:\\.|$))`
- Must only contain upper/lower case letters, digits, space, or the characters `.-[]()$+=;#@~,&`.

These rules are more restrictive than certain platforms, but are intended to provide a safe common baseline.

Temporal Types

Temporal data types store time data.

Instant

Instant types identify specific moment in time, independent of time zone. The value is stored in the database in Unix time, in one of three resolutions:

Type	Java	MySQL	JavaScript
`instant(s)`	`DateTime`	`bigint`	`date`
`instant(ms)`	`DateTime`	`bigint`	`date`
`instant(us)`	`DateTime`	`bigint`	`date`

Additionally, XINA supports standalone date and time components of instants:

Type	Java	MySQL	JavaScript
`date(s)`	`XDate`	`bigint`	`date`
`date(ms)`	`XDate`	`bigint`	`date`
`date(us)`	`XDate`	`bigint`	`date`
`time(s)`	`LocalTime`	`int`	`number`
`time(ms)`	`LocalTime`	`int`	`number`
`time(us)`	`LocalTime`	`bigint`	`number`

The date types are stored as the instant at start of date in UTC, as Unix time. The time types are stored as their respective unit limited to less than 24 hours.

XINA contains the following redundant legacy types, which will deprecated in a future release:

Type	Java	MySQL	JavaScript	Notes

`datetime`	`DateTime`	`bigint`	`date`	replaced by `instant(ms)`
`date`	`XDate`	`bigint`	`date`	replaced by `date(ms)`
`time`	`LocalTime`	`int`	`number`	replaced by `time(ms)`

Instant Parsing

Instant types support parsing from either Unix time or ISO style formatted timestamps. The general goal is to allow as many formats as possible while avoiding ambiguity.

Unix Parsing

A value provided as a JSON number will be interpretted with automatic precision detection based on the following rules:

- `n > 1e16` ? nanoseconds
- `n > 1e14` ? microseconds
- `n > 1e11` ? milliseconds
- `n > 1e8` ? seconds
- `n <= 1e8` ? error

This approach allows time values in the range of roughly 1973 to 5138.

A string value will be treated as Unix time if it matches the following format:

- `[+-]?` (optional sign)
- `\d[\d,]*` (digit followed by zero or more digits or commas)
- `(\.\d*)?` (optional period followed by zero or more digits)
- `(e[+-]?\d[\d,]*)?` (optional exponent)
- `([mun]?s)?` (optional precision)

Examples:

- `1609459200` ? 2021-01-01T00:00:00Z (auto-detected as seconds)
- `1,609,459,200` ? 2021-01-01T00:00:00Z (auto-detected as seconds, commas permitted and ignored)
- `1609459200s` ? explicit seconds
- `1609459200000ms` ? milliseconds
- `1609459200000000us` ? microseconds
- `1609459200000000000ns` ? nanoseconds
- `0` ? error (outside range supported by auto-detection)
- `0s` ? 1970-01-01T00:00:00Z
- `−31536000s` ? 1969-01-01T00:00:00Z

ISO Parsing

String values not matching the numeric format described above will be parsed as permissive ISO style formatted timestamps with an explicit offset. The ISO formats are detailed below in the **local** types section. The offset can be provided as:

- `±[hh]:[mm]`
- `±[hh][mm]`
- `±[hh]`
- `Z` (UTC shorthand)

Local

Local temporal types store values as ISO 8601 formatted strings without a time zone. This has the advantage of being plain-text searchable.

Type	Java	MySQL	JavaScript	Format
`localdate`	`LocalDate`	`char(10)`	`string`	`yyyy-MM-dd`
`localtime(s)`	`LocalTime`	`char(8)`	`string`	`HH:mm:ss`
`localtime(ms)`	`LocalTime`	`char(12)`	`string`	`HH:mm:ss.SSS`
`localtime(us)`	`LocalTime`	`char(15)`	`string`	`HH:mm:ss.SSSSSS`
`localdatetime(s)`	`LocalDateTime`	`char(19)`	`string`	`yyyy-MM-ddTHH:mm:ss`
`localdatetime(ms)`	`LocalDateTime`	`char(23)`	`string`	`yyyy-MM-ddTHH:mm:ss.SSS`
`localdatetime(us)`	`LocalDateTime`	`char(26)`	`string`	`yyyy-MM-ddTHH:mm:ss.SSSSSS`

`localdate` and `localdatetime(*)` are directly comparable in the database. A `localdate` and `localtime(*)` can be merged into a `localdatetime(*)` with `CONCAT([localdate], 'T', [localtime])`.

XINA contains the following redundant legacy types, which will deprecated in a future release:

Type	Java	MySQL	JavaScript	Notes
`localtime`	`LocalTime`	`char(12)`	`string`	replaced by `localtime(ms)`
`localdatetime`	`LocalDateTime`	`char(23)`	`string`	replaced by `localdatetime(ms)`

Local Parsing

Local date and time parsers use a strict resolution style with ISO chronology to ensure date validity (e.g., February 30th would be rejected). Additionally, all parsers support multiple separator characters for maximum flexibility:

- **Date separators**: `-`, `_`, ` `, `.`
- **Time separators**: `:`, `_`, ` `, `.`
- **Date-time separators**: `T`, `_`, ` `, `.`
- **Fraction separators**: `.`, `_`, ` `, `,`

Note that local types are stored in the database in the standard ISO format regardless of the import format.

Date Parsing

Dates can be parsed from the standard ISO format:

- **Pattern**: `YYYY[separator]MM[separator]DD`
- **Separators**: `-`, `_`, ` ` (space), `.`
- **Examples**:
  - `2011-12-03` (standard ISO)
  - `2011_12_03` (underscore)

    - `2011 12 03` (space)
    - `2011.12.03` (period)

Or an ordinal (day-of-year) ISO format:

- **Pattern**: `YYYY[separator]DDD`
- **Separators**: `-`, `_`, ` `, `.`
- **Examples**:
    - `2011-124` (124th day of 2011)
    - `2011_124`
    - `2011 124`
    - `2011.124`

Time Parsing

- **Pattern**: `HH[sep]MM[sep]SS[frac_sep]SSSSSSSSS`
- **Time Separators**: `:`, `_`, ` `, `.`
- **Fraction Separators**: `.`, `_`, ` `, `,`
- **Optional components**: seconds, fractional seconds (0-9 digits)
- **Examples**:
    - `10:15` (hour and minute only)
    - `10:15:30` (with seconds)
    - `10:15:30.123456789` (with nanoseconds)
    - `10_15_30` (underscore separator)
    - `10.15.30,123` (mixed separators)

Datetime Parsing

Combines local date and local time parsers.

Again, this supports standard ISO format:

- **Pattern**: `{LOCAL_DATE}[date_time_separator]{LOCAL_TIME}`
- **Date-Time Separators**: `T`, `_`, ` `, `.`
- **Examples**:
    - `2011-12-03T10:15:30` (standard ISO)
    - `2011-12-03T10:15:30.123456789` (with nanoseconds)
    - `2011_12_03_10_15_30` (all underscores)
    - `2011-12-03 10:15:30` (space separator)
    - `2011.12.03.10.15.30` (all periods)

Or an ordinal (day-of-year) ISO format:

- **Pattern**: `{ORDINAL_DATE}[date_time_separator]{LOCAL_TIME}`
- **Examples**:
    - `2011-124T10:15:30` (ordinal date)
    - `2011_124_10_15_30`
    - `2011-124 10:15:30.123`

Duration

Duration types contain a postive or negative integer duration of a specified unit.

Type	Java	MySQL	JavaScript
`duration(s)`	`Duration`	`bigint`	`number`
`duration(ms)`	`Duration`	`bigint`	`number`
`duration(us)`	`Duration`	`bigint`	`number`

JSON Types

JSON data types store JSON data directly in the database.

Type	Java	MySQL	JavaScript
`jsonarray`	`JsonArray`	`json`	`array`
`jsonobject`	`JsonObject`	`json`	`object`

Enum Types

Enum types map a series of discrete numeric integer values to text names. Though additional values may be added in the future, existing values will not change names or IDs.

notification_level

ID	Name	Notes
`0`	`none`	default level, no associated formatting
`1`	`success`	green
`2`	`info`	cyan
`3`	`notice`	yellow
`4`	`warning`	red
`5`	`primary`	blue, elevated over `none`
`6`	`secondary`	grey, below `none`

notification_type

ID	Name	Notes
`0`	`post`	
`1`	`task`	
`2`	`request`	request received
`3`	`response`	response to request received

`post_level`

ID	Name	Notes
`0`	`none`	default level, no associated formatting
`1`	`success`	green
`2`	`info`	cyan
`3`	`notice`	yellow
`4`	`warning`	red
`5`	`primary`	blue, elevated over `none`
`6`	`secondary`	grey, below `none`

System Types

ID

Wall ID

Path

Structured Data Types

UUID

HTML

XML

Collection Types

List

Set

Null Value

Although not a discrete XINA data type, `null` values may appear in any type depending on the context.

Type	Java	MySQL	JavaScript	JSON
`null`	`null`	`NULL`	`null`	`null`, `""` (empty string)

Broadly, the XINA interpretation of a `null` value is simply "no value". For this reason XINA treats empty strings in JSON as equivalent to `null` (because they contain no actual information). The implication of this concept in SQL is that operations on `null` values return `null`:

- `null + 1` ? `null`
- `null == null` ? `null`

Record Formatting

> UNDER CONSTRUCTION, details may change

A record can be formatted to a string by wrapping each field name in brackets. For example, the record:

`{ "foo": "bar" }`

with the format:

`"Foo: {foo}"`

would resolve to `"Foo: bar"`.

Additionally, values can be passed through **pipes**, allowing custom formatting to be used, or modifying the value entirely. The pipe format is:

`| @<pipe name> <pipe args>`

Revision #27
Created 9 June 2022 15:57:28 by Nick Dobson
Updated 21 January 2026 20:20:45 by Nick Dobson