API Reference

Overview

The **XINA API** (or **XAPI**) provides programmatic access to a XINA server.

> Note that client applications do not connect directly to the server. The XINA Tunnel utility performs the actual server connection, authentication, and security, and provides a local server for local client to connect.

XAPI is built on the **XINA Protocol** (or **XProtocol**), a TCP format used to communicate with the XINA server. It is designed to be simple and easy to implement across many languages and environments, using standard UTF-8 character encoding and JSON data structures.

Tokens

XProtocol is intended to be parsed in-place as a stream is read. This is achieved with variable length **tokens** of the following format:

- prefix length: one byte UTF-8 digit indicating length of the prefix in bytes
- prefix: UTF-8 digit(s) indicating the length of the content in bytes
- content: UTF-8 encoded string or binary data of (prefix count) bytes

For example:

- `14cake` = `"cake"`
- `213big hamburger` = `"big hamburger"`

The maximum allowed length of a single token is 2GB. A token may also be empty, which can be denoted with either the prefix `10` or the shorthand prefix `0`.

> Note that the content length is specified in *bytes*, not *characters*. Because UTF-8 is a variable length encoding format, it is recommend to first convert string data to bytes before creating a token for an accurate count.

Packets

Tokens are combined together into **packets**, which form all communication between the client and server.

Client Packets

Client packets are sent from the client to server. They use the following format:

- packet type :: one byte UTF-8 character
- header token :: JSON object containing header information (used primarily for system purposes, typically empty)
- content token :: UTF-8 encoded JSON object, binary data, or empty depending on token type

Client packets use the following packet types:

Type	Code	Description
`ACTION`	`A`	contains an API action (most common packet type)
`BINARY`	`B`	contains binary data (used for transmitting file data)
`CLOSE`	`X`	closes the connection
`CONTINUE`	`C`	prompts continuing a data stream from the server
`END`	`E`	indicates the end of a series of binary packets
`INIT`	`I`	initializes the connection
`KEEPALIVE`	`K`	ignored by both server and client, keeps connection open
`OBJECT`	`O`	indicates the start of a binary object

Server Packets

Server packets, inversely, are sent from the server to clients. They use the following format:

- packet type :: one byte UTF-8 character
- status code :: three byte UTF-8 numeric status code
- header token :: JSON object containing header information (only used currently for system functions, typically empty)
- status token :: JSON object containing status information
- content token :: UTF-8 encoded JSON object, binary data, or empty depending on token type

Server packets use the following packet types:

Type	Code	Description
`KEEPALIVE`	`K`	ignored by both server and client, keeps connection open
`SERVER`	`S`	primary server packet type, used for all functions

The server packet status token is a JSON object in the following format:

```
{
 "type"   : "OK" or "ER",
 "code"   : <int>,
```

```
  "message" : <string, optional>
}
```

The `"type"` indicates if an action succeeded (`"OK"`) or failed (`"ER"`). The `"code"` is a numeric identifier for the status and will be in the range of 100 to 500.

Code	Description
`1XX`	Success, more data available
`2XX`	Success, data ended
`4XX`	Content error
`5XX`	Server error

The optional `"message"` contains a plain text description of the status or error.

Control Flow

In practice the general design of XProtocol is call and response. Each packet (of most types) sent by a client will receive a single server packet in response. The exception to this rule is the binary object upload procedure, detailed below.

Initialization

When an application opens a connection with the XINA Tunnel, it must first send a single `INIT` packet containing a JSON object:

```
{ "version": "3.0" }
```

Currently the only attribute for this object is the XProtocol version number, which is currently 3.0. More attributes may be added in the future. The XINA Tunnel will then respond with a server packet indicating if the initialization is accepted. If it is not, the connection will then be closed by XINA Tunnel. If it is accepted the application may then begin sending other XAPI packets.

Actions

The bulk of the XAPI communication consists of a collection of discrete **actions**. Actions are fully **transactional**; any changes performed by an action must **all** be successful or **no** changes will be committed.

Each action is encoded as a single JSON object, with the exact format dependent on the action type. There are two categories of actions; data actions, which read or manipulate data, and administrative actions, which alter data structures or perform other administrative tasks.

All actions have a standard server response. If an action returns no data (such as a write action), or if the returned data fits in a single `SERVER` packet, the server will respond with a single `SERVER` packet, indicating success or failure for the action in the status token, and with any returned data in the content token. If a `SERVER` packet contains data, it will always be formatted at a single JSON object.

client		server
`ACTION` token	?	
	?	`SERVER` token (`2XX` status code)

In some cases the server may send the results of a query over multiple packets. This is indicated by a `1XX` status code in a `SERVER` packet. The client may send a `CONTINUE` token to receive the next packet, until a `2XX` code is received, indicating the data has been sent.

client		server
`ACTION` token	?	
	?	`SERVER` token (`1XX` status code)
`CONTINUE` token	?	
	?	`SERVER` token (`1XX` status code)
`CONTINUE` token	?	
	?	`SERVER` token (`2XX` status code)

In this case, the complete data response can be aggregated from the JSON objects according to the following rules:

- properties appearing in a single object are included as-is
- properties appearing in multiple objects are merged based on the data type
  - if the first instance of a property is a JSON array, successive arrays are concatenated and non-arrays are appended
  - otherwise, the values are appended individually into a JSON array

For the purposes of this merge operation, an explicit `null` value should be included in merged results, whereas an explicit or implicit `undefined` should not.

Example

Given the following three server packets:

```
{
 "a": 0,
 "b": 1
}
```

```
{
 "a": undefined,
 "b": [2]
 "c": [4, 5, 6]
}
```

```
{
  "b": null,
  "c": [7, 8, 9]
}
```

The correctly merged result would be:

```
{
  "a": 0,
  "b": [1, [2], null],
  "c": [4, 5, 6, 7, 8, 9]
}
```

Binary Object Upload

The `OBJECT`, `BINARY`, and `END` packet types allow a client to upload binary objects (such as files) to XINA. Binary objects received by the server are assigned a unique ID, which is returned to the client. The client may then use the ID to refer to the cached object in a future action. Cached objects are deleted if not used within 24 hours.

client		server	notes
`OBJECT` token	?		initializes the object
`BINARY` token	?		contains binary data
`...`	?		contains additional binary data as needed
`END` token	?		ends the object
	?	`SERVER` token (`2XX`)	content contains `object_id`

The `SERVER` token content will be a JSON object in the format:

```
{ "object_id": "<id>" }
```

Unlike other packet types, the server will not respond to each client packet, but only once the `END` packet is received. If the client sends any packet other than an `END` or `BINARY` packet after an `OBJECT` or `BINARY` packet any loaded data will be discarded. If an `END` packet is received without any binary data, an `object_id` will not be returned.

JSON Implementation

XINA API actions and responses are encoded in JSON, a common, widely supported, human-readable format. For the most part, the implementation of JSON in XINA is intended to match the format specification as closesly as possible. In the interest of increased flexibility and greater clarity, the XINA JSON parser does support some extensions and enforce additional conditions which are not part of the standard. However, **all XINA API JSON outputs will always be valid standard JSON**.

Empty Strings

XINA treats an empty string as equal to the JSON literal `null`. For example, the following two objects are equal in XINA:

```
{ "foo": "" }
```

```
{ "foo": null }
```

This is based on the XINA interpretation of `null` as meaning "no value", and an empty string also effectively representing "no value". Therefore, XINA does not permit an empty string as an object key, because `null` would not be a valid object key. For example, this is valid JSON, but not valid in XINA:

```
{ "": "foo" }
```

Duplicate Keys

The JSON standard permits (but discourages) duplicate keys in an object. XINA does not permit this, as the interpretation would be ambiguous. For example, this is valid JSON but not valid in XINA:

```
{
 "foo": "bar",
 "foo": "not bar?!"
}
```

Case Insensitive Keys

The JSON standard treats object keys as case sensitive, but XINA treats them as case-insensitive, for consistency with the overall case-insentitive design of the API, and to reduce abiguity. For example, this object would cause an error, due the keys being duplicates:

```
{
 "foo": true,
 "FOO": false
}
```

Normalized Keys

XINA normalizes object keys, meaning any leading and trailing white space is trimmed, and any internal white space is reduced to a single space. Again, this is to reduce ambiguity. For example, this object would cause an error, due the keys being duplicates:

```
{
 "foo bar": true,
 " foo  bar ": false
}
```

Extra Commas

XINA permits extra commas after the last item in arrays and objects. This does not affect the interpretation of the data.

```
[
 "foo",
 "bar",
]
```

```
{
 "foo": true,
 "bar": false,
}
```

Numeric Parsing

The JSON standard does not explicitly define rules around numeric parsing, aside from the text format for numbers. XINA initially parses numeric values with (virtually) unlimited precision. If the value is a whole number, it must fit in the range of a signed 64-bit integer (greater or equal to -2^{63} and less or equal to $2^{63}-1$). This applies even if the value has a fraction component, so long as it is zero. Otherwise it must fit into range representable by a 64-bit floating point value, $\pm(2-2^{-52})\cdot 2^{1023}$.

In summary:

- whole numbers must be precisely representable by a 64-bit signed integer

- decimal number magnitude must be accurately representable by a 64-bit floating point, but precision may be lost

Explicit Undefined

JSON does not include the JavaScript `undefined` literal, but XINA will parse it. For example, the following two objects are parsable and equivalent in XINA:

```
{
 "foo": true,
 "bar": undefined
}
```

```
{
 "foo": true
}
```

In practice this is rarely needed. In most cases a `null` literal will essentially serve this role, except in XINA record objects, where an explicit `null` refers to a `null` field value, and `undefined` refers to nothing.

Binary Objects

Binary objects are used to import any type of binary data. This may be used in conjuction with the low level API upload function, or the data may be embedded directly.

Uploaded Objects

When referencing an uploaded binary object, the object is specified as a string by the server-provided `"object_id"`. The object must have been uploaded prior to the API action in which it is being referenced.

Example

```
{
 "file": "<object ID>"
}
```

Embedded Objects

Alternatively, the content of an object may be embedded directly in JSON API actions.

Text

A `text` binary object contains UTF-8 encoded plain text.

Example

```
{
 "file": {
  "type": "text",
  "content": "<plain text>"
 }
}
```

Base64

A `base64` binary object must be encoded with the standard RFC 4648 format.

Example

```
{
 "file": {
```

```
  "type": "base64",
  "content": "<base64 encoded text>"
 }
}
```

Data Actions

Data actions read from or write to XINA databases.

Read Actions

SELECT

The primary read action in XINA. It closely mirrors the MySQL `SELECT` query, and returns data as a header of columns and list of rows. The full syntax for the `SELECT` object is available here.

Property	Value	Req	Default
action	`"select"`	?	
select	select	?	
rows	integer		`10,000`
use_strings	boolean		`false`
echo	boolean		`false`

The server response to a `SELECT` action will start with a header packet, containing a JSON array of JSON object(s) indicating the `name` of each column as a `string` and the XINA data type of each column as a `string`. This will be followed by packet(s) containing the data, as a JSON array of of JSON array(s) of values.

The optional `rows` property sets the limit of rows per packet. Note that this *does not* limit the total number of rows returned, this is set by the `limit` property of the select object.

If the `use_strings` property is `true`, all values will be stored as JSON strings instead of their associated JSON type.

If the `echo` property is `true`, the generated SQL query will be included in the header object in the `"query"` property. This is provided to support query debugging; it does not affect the query itself.

Example

Given a table `t` with two columns, `a` (`int(4)`), and `b` (`utf8text`), and three rows:

a	b
`0`	`"x"`
`1`	`"y"`
`2`	`"z"`

The following `SELECT` action:

```
{
 "action": "select",
 "select": {
  "from": "t"
 },
 "rows": 2
}
```

Would return three server packets.

First, the header information:

`100`

```
[
 {
  "name": "a",
  "type": "int(4)"
 },
 {
  "name": "b",
  "type": "utf8text"
 }
]
```

Second, the first two rows (limited to two by the `rows` property):

`100`

```
[
 [ 0, "x" ],
 [ 1, "y" ]
]
```

Third, the last remaining row (with the status code `200` indicating the end of the data):

`200`

```
[
 [ 2, "z" ]
]
```

FETCH

Reads specific types of data in a more structured format than the SELECT action. Although the syntax and response format differs depending on fetch type, all fetch actions share the boolean property `"count"`, which if true, overrides the default action output with a single value indicating the total count result for the current selection.

```
{
 "count": <integer>
}
```

FETCH RECORDS

Fetches records from a database.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"records"`	?	
database	database specifier	?	
records	records specifier		
fields	array of record field names		
attributes	arrray of attribute names		
where	expression		
order	array of order terms		default database order
limit	integer		1,000 (see below)
offset	integer		
children	boolean		`true`
count	boolean		`false`

Fetched records are returned as JSON objects, with each attribute and field name as a property key with the corresponding value. The `fields` and `attributes` property can be used to specify which fields and attributes (e.g. insert_at) are returned in the record. If the array is empty, all will be filtered out. If not provided or null, no filtering occurs. The `record_id` and `database_id` attribute is always included in the record and can't be filtered. In databases with the `file` feature enabled, each record will also include a generated presigned URL in the `"file_url"` property, and an S3 key reference in the `"file_key"` property.

If `children` is `true` and the specified database contains one or more child databases, all child records for each record will be included in the result. Each record with children will contain a `"children"` property, a JSON object with each key containing the name of the child database, and each value a JSON array of child record(s).

If both `records` and `where` are provided, they will be combined with a boolean `AND` operation.

The default limit for this operation is 1,000. Unlike the `SELECT` action which streams data directly from the underlying database, `FETCH` involves additional server overhead processing and formatting the result, so a limit

is enforced to maintain system performance. Exceeding the default limit explicitly is permitted but may cause performance issues depending on server configuration.

FETCH MULTIRECORDS

Fetches records from several databases at once.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"multirecords"`	?	
databases	databases specifier	?	
where	expression		
order	array of order terms		default order of first database
limit	integer		1,000 (see below)
offset	integer		
children	boolean		`true`
count	boolean		`false`

Fetched records are returned as JSON objects, in the same format as the standard FETCH RECORDS action, with the same behavior for file and child records. Each top level record will also include a `"database_id"`, with the numeric database ID of the database of origin for the record.

Internally this action uses the multi-database source, which creates a SQL UNION of the record tables of each database as a single virtual table. This is achived by unioning each column by field name. As such this action works best with databases with the same set of fields (names and data types). If databases each have fields with the same names but different types unpredictable server or client side errors may result.

FETCH PSEUDORECORDS

Fetches data from an arbirary query formatted as though it represents a set of records.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"pseudorecords"`	?	
select	select	?	
where	expression		
order	array of order terms		default order of first database
limit	integer		
offset	integer		
count	boolean		`false`

Each row of the result will be formatted as a JSON object, with each property key taken from the `SELECT` response header.

FETCH CRONS

Fetches all crons.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"crons"`	?	

FETCH FOLLOWS

Fetches all follows for a single user.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"follows"`	?	
user	user specifier		current user
count	boolean		`false`

FETCH KEYS

Fetches all keys for a single user. Fetching keys for a different user requires the `SUPER` privilege.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"keys"`	?	
user	user specifier		current user
count	boolean		`false`

FETCH LOGS

Fetches record logs from a database.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"logs"`	?	
database	database specifier	?	
records	records specifier	?	

FETCH NOTIFICATIONS

Fetches notifications for a single user.

Property	Value	Req	Default

action	`"fetch"`	?	
fetch	`"notifications"`	?	
user	user specifier		current user
type	notification type		
seen	boolean		

Notifications will always be returned ordered by time, descending.

If `type` is provided, only notifications of the same type will be returned.

If `seen` is `true`, only seen notifications will be returned. If `seen` is `false`, only unseen notifications will be returned.

FETCH POSTS

Fetches wall posts.

Property	Value	Req	Default
fetch	`"posts"`	?	
wall	wall specifier		all walls
following	boolean		`false`
threads	boolean		`false`
post	post ID		
children	boolean		`false`
records	boolean		`false`

FETCH PREFS

Fetches preferences for a single user.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"prefs"`	?	
user	user specifier		current user

FETCH PREF DEFS

Fetches server preference definitions.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"pref_defs"`	?	

FETCH TASKS

Fetches task information.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"tasks"`	?	
from	task ID		
user	user specifier		
text	string		
where	expression		
order	array of order terms		recent first
limit	integer		1,000 (see below)
offset	integer		
count	boolean		`false`

FETCH TEAM SUBSCRIPTIONS

Fetches subscriptions for a single team.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"user_subscriptions"`	?	
team	team specifier	?	

FETCH SEQS

Fetches all task sequences.

Property	Value	Required	Default
action	`"fetch"`	?	
fetch	`"seqs"`	?	

FETCH USERS

Fetches user information.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"users"`	?	
users	users specifier		all users

Property	Value	Req	Default
order	array of order terms		recent first
limit	integer		1,000 (see below)
offset	integer		

FETCH USER SUBSCPTIONS

Fetches subscriptions for a single user.

Property	Value	Req	Default
action	`"fetch"`	?	
fetch	`"user_subscriptions"`	?	
user	user specifier		current user

DOWNLOAD

The download action generates a signed URL to download a file stored in the XINA system. Note that this does not actually perform the download; the returned URL can be used outside the XINA API to download the file.

DOWNLOAD RECORD

Generates a signed URL for a record file.

Property	Value	Req	Default
action	`"download"`	?	
download	`"record"`	?	
database	database specifier	?	
record	record specifier	?	
version	integer		most recent

DOWNLOAD POST

Generates a signed URL for a post file.

Property	Value	Required	Default
action	`"download"`	?	
download	`"post"`	?	
post	post ID	?	

Write Actions

INSERT

The INSERT action inserts one or more records into a XINA database.

By default, the action will fail if any records being inserted have duplicate key values already in the database. If a different `on_duplicate` property is set, duplicate records will be updated according to the rules in the table. Only fields explicitly set in the `INSERT` will be changed. This is analogous to an `INSERT ... ON DUPLICATE KEY UPDATE` MySQL query.

Property	Value	Req	Default
action	`"insert"`	?	
database	database specifier	?	
records	records data	?	
on_duplicate	`"fail"` or `"update"`		`"fail"`
fail_no_op	`boolean`		`false`

Examples

Given a starting database containing key field `k`, fields `f1`, `f2`, and `f3`, with tags enabled, containing the following two records:

k	f1	f2	f3	tags
a	1	2	3	t1
b	1	2	3	t1

And inserting records:

```
[
 { "k": "a", "f1": 4, "f2": null, "tags": ["t2"] },
 { "k": "c", "f1": 1, "f2": null, "tags": ["t2"] }
]
```

on_duplicate: `"fail"`

Action fails due to duplicate key value `"a"`. No change occurs.

on_duplicate: `"update"`

Record with key value `"a"` is updated, and record with key value `"c"` is inserted. Note that field `f3` of `"a"` is unaffected because no inserted records specified an explicit value for `f3`.

k	f1	f2	f3	tags

a	**4**	**null**	3	t1, **t2**
b	1	2	3	t1
c	**1**	**null**	**null**	**t2**

REPLACE

The REPLACE action inserts one or more records into a XINA database and **overwrites** any existing records with duplicate keys.

Property	Value	Req	Default
action	`"replace"`	?	
database	database specifier	?	
records	records data	?	
on_duplicate	`"update"`, `"delete"`, or `"trash"` (if trash enabled for database)		`"update"`
fail_no_op	`boolean`		`false`

Examples

Given a starting database containing key field `k`, fields `f1`, `f2`, and `f3`, with tags enabled, containing the following two records:

k	f1	f2	f3	tags
a	1	2	3	t1
b	1	2	3	t1

And replacing records:

```
[
 { "k": "a", "f1": 4, "f2": null, "tags": ["t2"] },
 { "k": "c", "f1": 1, "f2": null, "tags": ["t2"] }
]
```

on_duplicate: `"update"`

Record with key value `"a"` is updated, and record with key value `"c"` is inserted. Note that `f3` of `"a"` is now `null` and `t1` is removed because all fields are overridden by the incoming record.

k	f1	f2	f3	tags
a	**4**	**null**	**null**	**t2**
b	1	2	3	t1
c	**1**	**null**	**null**	**t2**

on_duplicate: `"trash"` or `"delete"`

Existing record with key value `"a"` is deleted (or trashed), and new records `"a"` and `"c"` are inserted.

k	f1	f2	f3	tags
b	1	2	3	t1
a	**4**	**null**	**null**	**t2**
c	**1**	**null**	**null**	**t2**

If `"trash"` is used, the trash table now contains the original `"a"` record.

k	f1	f2	f3	tags
a	**1**	**2**	**3**	**t1**

SET

The SET action sets a database to contain the provided, and *only* the provided, records. Other records already present in the database are removed (either trashed or deleted, depending on the provided configuration).

Property	Value	Req	Default
action	`"set"`	?	
database	database specifier	?	
records	records data	?	
on_duplicate	`"update"`, `"delete"`, or `"trash"` (if trash enabled for database)		`"update"`
on_remove	`"delete"` or `"trash"` (if trash enabled for database)		`"trash"` if enabled, `"delete"` otherwise
fail_no_op	boolean		`false`

Examples

Given a starting database containing key field `k`, fields `f1`, `f2`, and `f3`, with tags enabled, containing the following two records:

k	f1	f2	f3	tags
a	1	2	3	t1
b	1	2	3	t1

And setting records:

```
[
 { "k": "a", "f1": 4, "f2": null, "tags": ["t2"] },
```

```
  { "k": "c", "f1": 1, "f2": null, "tags": ["t2"] }
]
```

on_duplicate: `"update"`

Record `"a"` is updated, record `"c"` is inserted, and record `"b"` is deleted (or trashed, depending on `on_remove`). Note that `f3` of `"a"` is now `null` and `t1` is removed because all fields are overridden by the incoming record.

k	f1	f2	f3	tags
a	**4**	**null**	**null**	**t2**
c	**1**	**null**	**null**	**t2**

on_duplicate: `"trash"` or `"delete"`

All existing records are deleted (or trashed, depending on `on_remove`), and new records `"a"` and `"c"` are inserted.

k	f1	f2	f3	tags
a	**4**	**null**	**null**	**t2**
c	**1**	**null**	**null**	**t2**

UPDATE

The UPDATE action updates the values of one or more fields and/or attached files of one or more records in a single database.

> This documentation applies to XINA 9.2 and above.

Property	Value	Req	Default
action	`"update"`	?	
database	database specifier	?	
records	records specifier	?	
fields	`jsonobject` map of fields to values to update (see below)		
expressions	`jsonobject` map of fields to expressions to update (see below)		
file	string object ID of file to update (see below)		
fail_no_op	boolean		`false`

The `fields` and `expressions` properties are JSON objects, where each key is interpretted as a field specifier in the context of the current database. For the `fields` property, each value is interpretted as a literal JSON value for the type of the specified field. For the `expressions` property, each value is interpretted as an expression, with the evaluated result of the expression stored with the record. These are provided separately because expressions would otherwise not be distinguishable from JSON object value literals.

The `file` property may be provided for databases with the `file` feature enabled. When the file is updated, associated file record attributes (`file_size`, `file_type`, etc) will be updated automatically from the new file.

If a single field is referenced more than once across the `fields` and `expressions` object, the action will fail, as the result would be ambiguous.

By default, if no records are found matching the records specifier, or no values are provided for `fields`, `expressions`, and `file`, the action will complete successfully without any changes occuring. If `fail_no_op` is `true`, the action will fail. Note, however, that `fail_no_op` will only detect these specific no-op conditions; it is possible that no changes will occur if provided update(s) do not actually change any fields of matched record(s).

DELETE

The DELETE action deletes one or more records from a database.

*Note that deleted records and all associated data are **permanently deleted** and cannot be restored.*

This action requires the `DELETE` database privilege.

Property	Value	Req	Default
action	`"delete"`	?	
database	database	?	
records	records	?	
fail_no_op	boolean		`false`

TRASH

The `TRASH` action moves one or more records into the trash table of a database. This is only available in databases with the trash feature enabled, otherwise the action will fail.

This action requires the `TRASH` database privilege.

Property	Value	Req	Default
action	`"trash"`	?	
database	database	?	
records	records	?	
fail_no_op	boolean		`false`

RESTORE

The `RESTORE` action moves one or more records from the trash table of a database into the record table. This is only available in databases with the trash feature enabled, otherwise the action will fail.

If any records being restored have duplicate keys as other records currently in the database the action will fail.

This action requires the `TRASH` database privilege.

Property	Value	Req	Default
action	`"restore"`	?	
database	database	?	
records	records	?	
fail_no_op	boolean		`false`

DISPOSE

The `DISPOSE` action deletes one or more records from the trash table of a database. This is only available in databases with the trash feature enabled, otherwise the action will fail.

*Note that disposed records and all associated data are **permanently deleted** and cannot be restored.*

This action requires the `DELETE` database privilege.

Property	Value	Req	Default
action	`"dispose"`	?	
database	database	?	
records	records	?	
fail_no_op	boolean		`false`

Admin Actions

Administrative actions create, modify, or delete XINA data structures, perform user management, or other system functions.

Schema Actions

SCHEMA

Returns the complete environment schema as a JSON object.

Property	Value	Required	Default
action	`"schema"`	yes	

Example

```
{
  "action" : "schema"
}
```

The server will return a JSON object:

```
{
  "groups" : [ ... ]
}
```

CREATE

The CREATE action is used to create new groups, databases, teams, and users.

CREATE GROUP

Creates a new group.

Property	Value	Required	Default
action	`"create"`	yes	
create	`"group"`	yes	
group	group definition	yes	

Property	Value	Required	Default
parent	group specifier	no	
teams	group teams association (see below)	no	

If a `parent` group is provided, the created group will be a child of the parent; otherwise the group will be a root level group.

The `teams` property is used to associate the group with one or more teams on creation. This may either be a JSON array of team specifier(s), and the group will be added to those teams with the default group privileges as specified by each team, or may be a JSON object, with each key interpretted as a team specifier, and each value containing a JSON object of group privilege(s) to boolean values, overriding the default team privileges.

CREATE DATABASE

Creates a new database.

Property	Value	Required	Default
action	`"create"`	yes	
create	`"database"`	yes	
database	database definition	yes	
parent	group specifier	yes	
teams	database teams association (see below)	no	

The `teams` property is used to associate the database with one or more teams on creation. This may either be a JSON array of team specifier(s), and the database will be added to those teams with the default database privileges as specified by each team, or may be a JSON object, with each key interpretted as a team specifier, and each value containing a JSON object of database privilege(s) to boolean values, overriding the default team privileges.

CREATE TEAM

Creates a new team.

Property	Value	Required	Default
action	`"create"`	yes	
create	`"team"`	yes	
team	team definition	yes	

CREATE USER

Creates a new user.

Property	Value	Required	Default
action	`"create"`	yes	

Property	Value	Required	Default
create	`"user"`	yes	
user	user definition	yes	

ALTER

The ALTER action is used to edit group, database, field, team, or user properties.

ALTER GROUP SET

Alters one or more group parameters. Requires the `alter` privilege on the specified group.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"group"`	yes	
op	`"set"`	yes	
group	group specifier	yes	
set	JSON object map of parameter(s) to value(s)	yes	

ALTER GROUP OBJECTS

Inserts, updates, or deletes group objects. Requires the `alter` privilege on the specified group.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"group"`	yes	
op	`"objects"`	yes	
group	group specifier	yes	
objects	JSON object map of key(s) to object value(s)	yes	

Any properties in the `objects` JSON object with a `null` value will be deleted, if they exist in the group objects.

ALTER GROUP FILES

Inserts, updates, or deletes group files. Requires the `alter` privilege on the specified group.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"group"`	yes	
op	`"files"`	yes	

Property	Value	Required	Default
group	group specifier	yes	
files	JSON object map of key(s) to object ID(s)	yes	

Any properties in the `files` JSON object with a `null` value will be deleted, if they exist in the group files.

ALTER DATABASE SET

Alters one or more database parameters. Requires the `alter` privilege on the specified database.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"set"`	yes	
database	database specifier	yes	
set	JSON object map of parameter(s) to value(s)	yes	

ALTER DATABASE OBJECTS

Inserts, updates, or deletes database objects.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"objects"`	yes	
database	database specifier	yes	
objects	JSON object map of key(s) to object value(s)	yes	

Any properties in the `objects` JSON object with a `null` value will be deleted, if they exist in the database objects.

ALTER DATABASE FILES

Inserts, updates, or deletes database files. Requires the `alter` privilege on the specified database.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"files"`	yes	
database	database specifier	yes	
files	JSON object map of key(s) to object ID(s)	yes	

Any properties in the `files` JSON object with a `null` value will be deleted, if they exist in the database files.

ALTER DATABASE ADD FIELDS

Adds one or more fields to an existing database. Requires the `alter` privilege on the specified database.

This operation modifies the database table(s) and may take several hours for very large databases.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"add_fields"`	yes	
database	database specifier	yes	
fields	JSON array of field definitions	yes	
first	`boolean`	no	`false`
after	field specifier	no	

The action will fail if any of the new fields have the same name or label as eachother or any existing field in the database.

By default, new fields are added at the end of the existing fields. If `first` is true, new fields will be added at the front of the existing fields. If `after` is provided, new fields will be added immediately after the specified field, and before any following fields. If both `first` is `true` and `after` is provided, the action will fail.

ALTER DATABASE DROP FIELDS

Drops one or more fields from an existing database. Requires the `alter` privilege on the specified database.

This operation modifies the database table(s) and may take several hours for very large databases.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"drop_fields"`	yes	
database	database specifier	yes	
fields	fields specifier	yes	

The action will fail if any of the specified fields is a key field, or if the action would drop all fields from a database.

ALTER DATABASE ORDER FIELDS

Specifies ordering of one or more fields in a database. Requires the `alter` privilege on the specified database.

This operation does not modify the database table(s), only the field order as indicated by the XINA schema.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"order_fields"`	yes	
database	database specifier	yes	
fields	JSON array of field specifiers	yes	
after	field specifier	no	

Fields will be ordered based on the order provided in the `fields` property, with any fields not specified maintaining their original order after the specified set. If `after` is provided, the ordered block will start following that specified field, with any non-specified fields before the `after` field maintaining their original order. If the `after` field is included in the `fields` property, the action will fail.

Example

Given a database `db` with the fields order `f1`, `f2`, `f3`, `f4`, `f5`, `f6`, the action:

```
{
  "action": "alter",
  "alter": "database",
  "database": "db",
  "op": "order_fields",
  "fields": ["f4", "f2"]
}
```

The resulting field order would be `f4`, `f2`, `f1`, `f3`, `f5`, `f6`.

Given the same intial setup and action but adding `"after": "f3"`, the resulting order would be: `f1`, `f3`, `f4`, `f2`, `f5`, `f6`

ALTER DATABASE RESET PARTITIONS

Resets one or more partitions of a database record table. Requires the `alter` privilege on the specified database.

*This action permanently deletes **all data in the specified partitions**.

Unlike the DELETE action, this action immediately frees storage space in the underlying database system.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"database"`	yes	
op	`"reset_partitions"`	yes	
database	database specifier	yes	

Property	Value	Required	Default
partitions	partitions specifier	yes	

ALTER FIELD SET

Alters one or more field parameters. Requires the `alter` privilege on the specified database.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"field"`	yes	
op	`"set"`	yes	
database	database specifier	yes	
field	field specifier	yes	
set	JSON object map of parameter(s) to value(s)	yes	

ALTER FIELD OBJECTS

Inserts, updates, or deletes field objects. Requires the `alter` privilege on the specified database.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"field"`	yes	
op	`"objects"`	yes	
database	database specifier	yes	
field	field specifier	yes	
objects	JSON object map of key(s) to object value(s)	yes	

Any properties in the `objects` JSON object with a `null` value will be deleted, if they exist in the field objects.

ALTER FIELD FILES

Inserts, updates, or deletes field files. Requires the `alter` privilege on the specified database.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"field"`	yes	
op	`"files"`	yes	
database	database specifier	yes	
field	field specifier	yes	

Property	Value	Required	Default
files	JSON object map of key(s) to object ID(s)	yes	

Any properties in the `files` JSON object with a `null` value will be deleted, if they exist in the field files.

ALTER USER SET

Alters one or more user parameters. Requires the `super` privilege to alter any user other than the current user.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"user"`	yes	
op	`"set"`	yes	
user	user specifier	yes	
set	JSON object map of parameter(s) to value(s)	yes	

ALTER USER OBJECTS

Inserts, updates, or deletes user objects. Requires the `super` privilege to alter any user other than the current user.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"user"`	yes	
op	`"objects"`	yes	
user	user specifier	yes	
objects	JSON object map of key(s) to object value(s)	yes	

Any properties in the `objects` JSON object with a `null` value will be deleted, if they exist in the user objects.

ALTER USER FILES

Inserts, updates, or deletes user files. Requires the `super` privilege to alter any user other than the current user.

Property	Value	Required	Default
action	`"alter"`	yes	
alter	`"user"`	yes	
op	`"files"`	yes	
user	user specifier	yes	
files	JSON object map of key(s) to object ID(s)	yes	

Any properties in the `files` JSON object with a `null` value will be deleted, if they exist in the user files.

DROP

Permanently delete teams, groups, databases, or users.

DROP GROUP

Drops a group. This action requires the `super` privilege.

*This action permanently deletes **all data in the specified group**.*

Property	Value	Required	Default
action	`"drop"`	yes	
drop	`"group"`	yes	
group	group specifier	yes	
children	`boolean`	no	`false`

By default, if the specified group has any child groups or databases the action will fail. If `children` is `true`, all child groups and databases will also be dropped.

DROP DATABASE

Drops a database. This action requires the `super` privilege.

*This action permanently deletes **all data in the specified database**.*

Property	Value	Required	Default
action	`"drop"`	yes	
drop	`"database"`	yes	
database	database specifier	yes	
children	`boolean`	no	`false`

By default, if the specified database has any child databases the action will fail. If `children` is `true`, all child databases will also be dropped.

DROP TEAM

Drops a team. This action requires the `super` privilege.

*This action permanently deletes **all data in the specified team**.*

Property	Value	Required	Default
action	`"drop"`	yes	

Property	Value	Required	Default
drop	`"team"`	yes	
team	team specifier	yes	

DROP USER

Drops a user. This action requires the `super` privilege.

*This action permanently deletes **all data associated with the specified user**.*

Property	Value	Required	Default
action	`"drop"`	yes	
drop	`"user"`	yes	
user	user specifier	yes	

JOIN

The JOIN action joins groups, databases, or users to one or more teams.

JOIN GROUPS

JOIN DATABASES

JOIN USERS

LEAVE

The LEAVE action removes groups, databases, or users from one or more teams.

LEAVE GROUPS

LEAVE DATABASES

LEAVE USERS

User Actions

GRANT

REVOKE

REQUEST

Request an arbitrary action to be performed by a user with required permissions.

RETRACT

Retract one or more user requests.

APPROVE

REJECT

Task Actions

Task actions provide features for running and interacting with asynchronous tasks managed by the XINA Run application or AWS Lambda platform.

Task Reference

Task Actions may reference existing tasks by either their numeric Task ID or a JSON object with the following format:

```
{
  "type": "ref",
  "ref": [<ref_id assigned when creating the task>]
}
```

The `ref_id` is an arbitrary, optional value defined in the Task Definition when a task is created. The `ref_id` does not have to be unique, implying the Task Action will be performed on all tasks that match the `ref_id`.

Task Actions

RUN

Run one or more asynchronous tasks.

Property	Value	Req	Default
action	`"run"`	?	

Example

```
{
  "action" : "run",
  "tasks"  : [ <task definition or reference>, ... ]
}
```

Task Definition

A task definition is used to define a new Task. It has a JSON object with the following format:

```
{
  "name"     : <string, task name>,
  "conf"     : <JSON object, format depends on task>,
  "parent"   : <long, parent task ID, optional>,
  "seq"      : <string, sequence name, optional>,
  "auto"     : <boolean, optional, default false>,
  "archive"  : <boolean, optional, default false - if true, task workspace will not be deleted>,
  "open"     : <boolean, optional, default false>,
  "desc"     : <string, optional>,
  "priority" : <int, optional>,
  "timeout"  : <int, ms, optional>,
  "ref_id"   : <long, optional - arbitrary value for Task Referencing>
}
```

CONCLUDE

Explicitly conclude a task. This is currently only used by AWS Lambda tasks, to notify the XINA server that the task has concluded. If a value is provided for `"delay"`, the server will wait that many milliseconds before concluding the task. This supports tasks that may have a longer cleanup or import period following the immediate task completion.

Property	Value	Req	Default
action	`"conclude"`	?	

Example

```
{
  "action" : "conclude",
  "task"   : <task ID>,
  "delay"  : <number, ms, 0-5000, optional, default 0>
}
```

CANCEL

Cancel one or more tasks.

Property	Value	Req	Default
action	`"cancel"`	?	

Only non-concluded tasks may be canceled. If "ignore" is false, and any specified tasks have concluded, an error will be thrown and no changes will occur. If "ignore" is true, all non-concluded specified tasks will be canceled,

and any concluded tasks will be ignored.

Example

```
{
  "action" : "cancel",
  "tasks"  : [ <task reference>, ... ],
  "ignore" : <boolean, optional, default false>
}
```

CLEAN

Permanently delete one or more asynchronous task records and any associated files.

Property	Value	Req	Default
action	`"clean"`	?	

Only concluded tasks may be cleaned. If `"ignore"` is `false`, and any specified tasks have not concluded, an error will be thrown and no changes will occur. If `"ignore"` is `true`, all concluded specified tasks will be cleaned, and any non-concluded tasks will be ignored.

Example

```
{
  "action" : "clean",
  "tasks"  : [ <task reference>, ... ],
  "ignore" : <boolean, optional, default false>
}
```

DESTROY

Cancel and clean one or more asynchronous tasks. Unlike the CANCEL and CLEAN actions, this will apply to all tasks regardless of the current task state.

Example

```
{
  "action" : "clean",
  "tasks"  : [ <task reference>, ... ]
}
```

Sequence Actions

PAUSE

Pause execution of one or more asynchronous task threads. This does not affect any task currently running in the thread, but future tasks assigned to the thread will not run until the thread is resumed.

Example

```
{
  "action"  : "pause",
  "threads" : [ <string, thread name>, ... ]
}
```

RESUME

Resume execution of one or more asynchronous task threads. If `"continue"` is `true` and a continuable task is currently locked on the thread, that task will be continued.

Example

```
{
  "action"   : "pause",
  "threads"  : [ <string, thread name>, ... ],
  "continue" : <boolean, optional, default false>

}
```

Struct Actions

Struct actions are complex data actions designed to be used with XINA Structs. Unlike most API actions, they may involve complex multi-step operations, and are dependent on the structs configuration of groups and databases.

Data Actions

STRUCT BUFFER IMPORT

Imports a buffer data file into a pipe.

Property	Value	Req	Default
action	`"struct_buffer_import"`	?	
pipe	pipe group specifier	?	
file	binary object	?	
conf	`jsonobject`		
t_import	`instant`		

When a data set is imported the XINA server will run the following steps:

- For each row:
  - validate time and value
  - process mnemonic
    - If mnemonic ID
      - If found in definitions database:
        - If mnemonic is deprecated, throw error
        - Else, use mnemonic for row
      - Else, throw error due to unrecognized ID
    - Else, parse name and optional unit
      - If name match found:
        - If unit is provided and does not match, throw error
        - If mnemonic is deprecated, throw error
        - Else, use mnemonic for row
      - Else, mnemonic is new, create new temporary mnemonic definition based on provided information
- If any rows contain same mnemonic and time, throw error
- Check for time overlap in database
  - If found
    - If `on_overlap` = `fail`, throw error
    - Else If `on_overlap` = `delete`, delete all data from database in time range of imported data
    - Else (`on_overlap` = `ignore`), do nothing
- If new mnemonic definitions created, insert into mnemonic definition database
- Insert data into mnemonic database

STRUCT MN ALIAS

Adds one or more aliases of name/unit pairs to a single existing mnemonic.

Property	Value
`action`	`"struct_mn_alias"`
`database`	mnemonic definition database specifier
`mn`	mnemonic ID
`aliases`	`string[]`, name/unit pair alias(es) for mnemonic

STRUCT MN EDIT

Edits one or more properties of a single existing mnemonic.

Property	Value	Required
`action`	`"struct_mn_edit"`	?
`database`	mnemonic definition database specifier	?
`mn`	mnemonic ID	?
`name`	`string`, new name/unit pair for mnemonic	
`state`	`string`	

STRUCT MN MERGE

Merges one or more existing mnemonics into a single existing mnemonic.

STRUCT EVENT

Performs context-aware event operations.

Unlike typical record operations, these actions support event definition lookup and creation. Event records or updates may specify a `"name"` property, as if it were a database field. This will be used to lookup a corresponding event ID from the event definitions associated with the database, and create a new definition with the name if one is not found. Alternatively, the `"name"` may reference an event definition by external ID, by starting with the `$` character.

If an event specifies both a `"name"` and `"e_id"`, the action will fail, as the outcome is ambiguous. If the `"name"` property value is numeric or numeric text, it will interpretted as a direct event ID reference (as if it had been provided as `"e_id"`).

`"e_id"` values are validated against existing event definitions, and the action will fail if the event ID is not found.

STRUCT EVENT INSERT

Inserts one or more events into a single event database.

Property	Value	Req	Default
action	`"struct_event"`	?	
op	`"insert"`	?	
database	event database	?	
events	event records	?	

If the event database has an associated event change database, the event change database will be checked for any `update` records, and the changes will be applied to the incoming events before they are inserted.

If any inserted UEIDs are already present in the database, the action will fail.

STRUCT EVENT CLOSE

Closes one or more open interval event(s).

Property	Value	Req	Default
action	`"struct_event"`	?	
op	`"close"`	?	
database	event database	?	
t	`instant(us)` closing time		now
events	events specifier	?	
fields	field value map		

The closing time is specified by the `t` property.

The `events` property is an extension of the standard records specifier, but may include UEID(s) as strings. Only currently open intervals in the specified database will be affected.

If the `fields` property is provided, updates the value(s) of the specified field(s) in the map for all events being closed.

STRUCT EVENT UPDATE

Updates one or more events.

Property	Value	Req	Default
action	`"struct_event"`	?	
op	`"update"`	?	
database	event database	?	
t	`instant(us)` update time		now

Property	Value	Req	Default
events	events specifier		
fields	field(s) to update	?	

If the event database is a child of a pipe, an event change record is inserted in the associated event change database for each event UEID matching the specifier. Additionally, if any updated fields are not configured to permit updating, the action will fail.

STRUCT EVENT CLEAR

Schema Actions

STRUCT CREATE

The STRUCT CREATE action is used to create a variety of XINA Structs compatible schema elements.

STRUCT CREATE CATEGORY

Creates a structs category group.

Property	Value	Req	Default
action	`"struct_create"`	?	
create	`"category"`	?	
parent	group specifier	?	
name	`string`	?	
label	`string`		name
desc	`string`		label
group_teams	team group privilege map		
database_teams	team database privilege map		

The `parent` group must be either a project group or category group, or the action will fail. The `name` (and `label`, if provided) must not be in use by any group siblings, or the action will fail.

STRUCT CREATE MODEL

Creates a structs model group.

Property	Value	Req	Default
action	`"struct_create"`	?	
create	`"model"`	?	
parent	group specifier	?	

Property	Value	Req	Default
name	`string`	?	
label	`string`		name
desc	`string`		label
event	`boolean`		`false`
eventf	`boolean`		`false`
eventfs	`boolean`		`false`
group_teams	team group privilege map		
database_teams	team database privilege map		

The `parent` group must be either a project group or category group, or the action will fail. The `name` (and `label`, if provided) must not be in use by any group siblings, or the action will fail.

STRUCT CREATE PIPE

Creates a struct pipe group.

Property	Value	Required	Default
action	`"struct_create"`	?	
create	`"pipe"`	?	
model	group specifier	?	
name	`string`	?	
label	`string`		name
desc	`string`		label
group_teams	team group privilege map		
database_teams	team database privilege map		
partition	`boolean` or `{"from": <start year>, "to": <end year>}`		`false`
	See the pipe definition for other supported properties		

The `parent` group must be either a project group or category group, or the action will fail. The `name` (and `label`, if provided) must not be in use by any group siblings, or the action will fail.

STRUCT CREATE DEF

Creates a structs definitions group, with associated databases.

Property	Value	Req	Default
action	`"struct_create"`	?	
create	`"def"`	?	
parent	group specifier	?	

The `parent` group must be either a project, category, or model group, or the action will fail.

STRUCT CREATE EVENT

Creates a new structs event database.

Property	Value	Req	Default
action	`"struct_create"`	?	
create	`"event"`	?	
group	group specifier	?	
type	`"none"`, `"file"`, or `"files"`		`"none"`
name	string		`"event"`, `"eventf"`, or `"eventfs"`
label	string		name
desc	string		label
singular	string		`"event"`
plural	string		singular`s`
conf	JSON object		
label_type	string or text XINA type		`utf8vstring(128)`
fields	array of field definitions		
teams	team database privilege map		

STRUCT CREATE NOTEBOOK

Creates a new structs notebook database.

Property	Value	Req	Default
action	`"struct_create"`	?	
create	`"notebook"`	?	
parent	group specifier	?	
name	string	?	
label	string		name
desc	string		label
fields	array of field definitions		
teams	team database privilege map		

STRUCT CREATE PROJECT

Creates a structs project group.

Property	Value	Req	Default

action	`"struct_create"`	?	
create	`"project"`	?	
parent	group specifier		
name	`string`	?	
label	`string`		name
desc	`string`		label
group_teams	team group privilege map		
database_teams	team database privilege map		

If a `parent` group is specified, it may not include a structs definition (since project groups must be at the top level of a struct heirarchy). The `name` (and `label`, if provided) must not be in use by any group siblings, or the action will fail.

System Actions

System actions are used for internal functions, client-specific features, and debugging.

ACCESS

Returns a temporary websocket access ID. Only used in the web client.

Property	Value	Req	Default
action	`"access"`	?	

Response

Property	Value
access_id	`string`
expires	`number` (Unix ms)

ECHO

Returns a string sent in the action. Used for debugging.

Property	Value	Req	Default
action	`"echo"`	?	
echo	`string`	?	

Response

Property	Value
echo	`string`

IF

Performs a write-only action depending on query based conditional check. The value of the first column of the first row is checked for each query, and the first returning `true` is executed. If none match an optional `else` action will be executed. Otherwise, no action occurs.

Property	Value	Req	Default

action	`"if"`	?	
do	`condition[]`	?	
else	action		

Condition

Property	Value	Req
if	select	?
then	action	?

SYNC

Returns a string sent in the action. Used for debugging. Redundant with ECHO but preserved for backwards compatibility.

Property	Value	Req	Default
action	`"echo"`	?	
id	`string`	?	

Response

Property	Value
id	`string`

VIEW

Marks tasks or notifications as `seen`, for purposes of client side notification.

WAIT

Waits a specified number of milliseconds and returns. Can be set to fail, or fail if resolved before a particular instant.

Property	Value	Req	Default
action	`"wait"`	?	
ms	`number`		`1000`
fail	`boolean`		`false`
fail_before	`instant`		

Specifier Syntax

Specifiers are objects which specify schema or data elements.

In general a specifier is an object with a `type` property indicating the type of the specifier. Some specifiers provide a shorthand version by substituting a different JSON data type.

Common

There are several common specifiers used by multiple components.

All

Specifies all elements in the current context (for example, as a records specifier all would include all records in the selected database).

Property	Value
`type`	`"all"`

Example

```
{ "type": "all" }
```

ID

Specifies an element by numeric ID. The value is provided directly as a JSON number.

Example (JSON number)

```
123
```

Name

Specifies an element by name. The value is provided directly as a JSON string.

Example (JSON string)

```
"foo"
```

Where

Specifies element(s) meeting a condition provided by an expression.

The source against which the expression is used depends on the context, but in general can be represented as

```
SELECT [elements] FROM [source] WHERE [expression]
```

where source is the table containing the element.

Property	Value
`type`	`"where"`
`where`	expression

Array

Specifies elements using an array of singular specifiers. The value is provided directly as a JSON array.

The types of the individual specifiers depend on the element, but in general unless otherwise noted all singular specifier types for the element may be used. Specifier types may also be intermingled.

Example (JSON array)

```
[ 123, "foo", {"type": "where", "where", "..."} ]
```

Schema Elements

Groups

Specifies one or more groups. Group specifiers are also valid groups specifiers.

- All
- Array

Group

Specifies a single group.

- ID
- Name

Databases

Specifies one or more databases. Database specifiers are also valid databases specifiers.

- All
- Array

Database

Specifies a single database.

- ID
- Name

Fields

Specifies one or more fields. Field specifiers are also valid fields specifiers.

- All
- Array

Field

Specifies a single field.

- ID
- Name

Walls

Specifies one or more walls. Wall specifiers are also valid walls specifiers.

- `Array`

Wall

Specifies a single wall.

Group Wall

Specifies the wall of single group.

Property	Value
`type`	`"group"`
`group`	group specifier

Example

```
{
 "type"  : "group",
 "group" : "foo"
}
```

Database Wall

Specifies the wall of single database.

Property	Value
`type`	`"database"`
`database`	database specifier

Example

```
{
 "type"     : "database",
 "database" : "foo"
}
```

Record Wall

Specifies the wall of single record.

Property	Value
`type`	`"record"`
`database`	database specifier
`record`	record specifier

Example

```
{
 "type"     : "database",
 "database" : "foo",
 "record"   : 123
```

```
}
```

User Wall

Specifies the wall of single user.

Property	Value
`type`	`"user"`
`user`	user specifier

Example

```
{
 "type" : "user",
 "user" : "foo"
}
```

Data Elements

Records

Specifies a set of records in a single database. Record specifiers are also valid records specifiers.

- All
- Array
- Where

Record

Specifies a single record in a database.

- `ID`

Key

Specifies a single record by a set of key value(s).

{ "type" : "key", "key" : <[[XINA API :: Data Syntax#Fields|fields]]> }

Each key must specify a non-null value for each key field of the database.

Posts

Specifies a set of posts. Post specifiers are also valid posts specifiers.

- `All`
- `Array`
- `Where`

Post

Specifies a single post.

- `ID`

Administrative

Users

Specifies a set of users. User specifiers are also valid users specifiers.

- `All`
- `Array`
- `Where`

User

Specifies a single user. Note that `name` in this case refers to the username, not the user's full name.

- `ID`
- `Name`

Group Privileges

Specifies a set of group privileges.

- `All`
- `Array`

Group Privilege

Specifies a single group privilege as a JSON string. The valid group privileges are:

- "select"
- "post"
- "reply"
- "alter"
- "grant"

Database Privileges

Specifies a set of database privileges.

- All
- Array

Database Privilege

Specifies a single database privilege as a JSON string. The valid database privileges are:

- "select"
- "post"
- "reply"
- "update"
- "insert"
- "trash"
- "delete"
- "lock"
- "alter"
- "grant"

Record Syntax

JSON Format

A single record may be encoded as a JSON object:

Property	Value
<field name / label>	field type appropriate value / `null`
`"expressions"`	JSON object mapping field name/label to expression
`"file"`	binary object (if database has `file` enabled)
`"tags"`	JSON array of string(s) (if database has `tag` enabled)

The `"expressions"` property allows field values to be specified by expression, rather than explicit value. Between the base object and `"expressions"` object, field may only have a single value provided, or an error will be thrown.

Multiple records may be encoded as a JSON array of JSON objects in this format.

DSV Format

Record data may be provided in a delimiter separated values format. In this case the record data itself is contained in a binary object.

Property	Value
`"type"`	`"dsv"`, `"csv"`, or `"tsv"`
`"file"`	binary object
`"delimiter"`	string (required for `"dsv"`)
`"quote"`	string (optional)

The `"csv"` and `"tsv"` types specify default delimiters of comma (`,`) and tab (`\t`), respectively.

Example

```
{
 "records": {
  "type": "dsv",
  "file" : "<object ID>",
  "delimit": ";"
 }
}
```

The format of the separated values file is largely based on the RFC 4180 standard. The specific requirements are:

- lines must end with `LF` (`\n`) or `CR` `LF` (`\r\n`)
- line breaks cannot be used in values
- the default quote character is `"` (double quotes)
- any field *may* be quoted by the quote character
- any field containing the delimiter must be quoted
- a quote character in a quoted value must be represented by two quote characters
- the first row must contain the names of each field
- blank lines with no data are ignored

Expression Syntax

XINA expressions translate to MySQL expressions, which are evaluated as a query is executed.

All expressions have a **standard form** as a JSON object, with a `type` property specifying the expression type, and additional properties as needed by that expression type.

Additionally, certain expression types may be represented using a **short form**, which is formatted as a JSON object with a single property prefixed with the `$` character.

Literals

Literal expressions represent a single, discrete value.

Null

The MySQL `NULL` value. May also be specified with the JSON `null` value.

Property	Value
`type`	`"null"`

Example (as object)

```
{ "type": "null" }
```

Example (as JSON literal)

```
null
```

Number

A numeric literal value. The value may be provided as a native JSON number, or encoded as a string. May also be provided directly as a JSON `number` value.

Property	Value
`type`	`"number"`
`value`	`number` or `string`

Example (as object)

```
{
 "type"  : "number",
 "value" : 123
}
```

Example (as JSON literal)

```
123
```

String

A string literal value. May also be provided directly as a JSON `string`.

Property	Value
`type`	`"string"`
`value`	`string`

Example (as object)

```
{
 "type"  : "string",
 "value" : "foo"
}
```

Example (as JSON literal)

```
"foo"
```

Datetime

A datetime literal value. Interpreted by the database as Unix time in milliseconds.

Property	Value
`type`	`"datetime"` or `"dt"`
`value`	`integer` or `string`

If the value provided is an `integer` it must be the number of milliseconds since the Unix epoch. If the value is a `string` it must be encoded according to the following syntax, taken from the ISO8601 standard:

```
date-opt-time     = date-element ['T' [time-element] [offset]]
date-element      = std-date-element | ord-date-element | week-date-element
std-date-element  = yyyy ['-' MM ['-' dd]]
ord-date-element  = yyyy ['-' DDD]
week-date-element = xxxx '-W' ww ['-' e]
time-element      = HH [minute-element] | [fraction]
minute-element    = ':' mm [second-element] | [fraction]
second-element    = ':' ss [fraction]
fraction          = ('.' | ',') digit [digit] [digit]
offset            = 'Z' | (('+' | '-') HH [':' mm [':' ss [('.' | ',') SSS]]])
```

If the offset is not provided the timezone will be assumed to be UTC.

Supports shorthand syntax with the `$dt` property.

Property	Value
`$dt`	`integer` or `string`

Local Datetime

A local datetime literal value.

Property	Value
`type`	`"localdatetime"` or `"ldt"`
`value`	`string`

The value must be encoded according to the same syntax as the datetime literal, except with the offset omitted.

Supports shorthand syntax with the `$ldt` property.

Property	Value
`$ldt`	`string`

Local Date

A local date literal value.

Property	Value
`type`	`"localdate"` or `"ld"`
`value`	`string`

The value must be encoded according to the same syntax as the `date-element` in the datetime literal.

Supports shorthand syntax with the `$ld` property.

Property	Value
`$ld`	`string`

Local Time

A local time literal value.

Property	Value
`type`	`"localtime"` or `"lt"`
`value`	`string`

The value must be encoded according to the same syntax as the `time-element` in the datetime literal.

Supports shorthand syntax with the `$lt` property.

Property	Value
`$lt`	`string`

Columns

Column expressions specify a column of a table. Although each column type has a separate full syntax, there is a shorthand syntax with the `$col` property, which infers the column type based on the content.

Property	Value
`$col`	`string` column

```
column          = system-column | database-column
system-column   = system-table-name '.' system-parameter-name
database-column = database-path ['@' database-table-name] '.' (parameter-name | attribute-name | field-name
| blob-attribute)
blob-attribute  = blob-name ':' blob-attribute-name
```

Examples

- `user.email` : `email` parameter of the `user` system table
- `a.b.record_id` : `record_id` attribute of the `record` table of database `b` in group `a`
- `a.b@trash.record_id` : `record_id` attribute of the `trash` table of database `b` in group `a`
- `a.b.c` : `c` field of the `record` table of database `b` in group `a`

System Parameter Column

Specifies a column of a system table.

Property	Value
`type`	`"column"`
`table`	`string`
`column`	`string`

Database Parameter Column

Specifies a parameter column of a database table.

Property	Value
`type`	`"column"`
`database`	database specifier
`table`	`string` table name
`column`	`string` parameter name

Database Attribute Column

Specifies an attribute column of a database table.

Property	Value
`type`	`"column"`
`database`	database specifier
`table`	`string` table name
`column`	`string` attribute name

Database Field Column

Specifies a field column of a database table.

Property	Value
`type`	`"column"`
`database`	database specifier
`table`	`string` table name
`column`	field specifier

Alias

Although the alias is not technically a column, it can refer directly by name to any column in the source, or to an alias of a result column.

Property	Value
`type`	`"alias"`
`value`	`string`

Supports shorthand syntax with the `$alias` property.

Property	Value
`$alias`	`string`

Evaluations

Evaluations are expressions evaluated by MySQL.

Between

Returns true if the expression `e` is between `min` and `max`.

Property	Value
`type`	`"between"`
`e`	expression
`min`	expression
`max`	expression

Supports shorthand syntax with the `$between` property. Takes a JSON array of exactly 3 expressions, in the order `e`, `min`, and `max`.

Property	Value
`$between`	array of three expressions

Binary

Binary operation, evaluated as `e1` `op` `e2`.

Property	Value
`type`	`"binary"`
`op`	`string`
`e1`	expression
`e2`	expression

Valid binary operators are as follows:

Operator	Description
`and`	logical AND
`or`	logical OR
`=`	equal
`!=`	not equal
`>`	greater
`>=`	greater or equal
`<`	less
`<=`	less or equal
`is`	test against `NULL`
`like`	simple pattern matching, see here
`regexp`	advanced pattern matching, see here
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus
`&`	bit-wise AND
`|`	bit-wise OR
`<<`	left shift
`>>`	right shift

Supports shorthand syntax with any operator by prefixing it with `$`. Takes a JSON array of two or more expressions. If more than two expressions are provided, behavior depends on the operator type. Logic and math operators perform each binary operation in order of expressions. For example:

- `{"$and": [true, true, false]}` = `(true and true) and false` = `false`
- `{"$/": [12, 3, 2, 2]}` = `((12 / 3) / 2) / 2` = `1`

Comparison operators perform a logical AND of the comparisons of the first expression to each additional expression.

- `{"$=": [0, 1, 2]}` = `(0 = 1) and (0 = 2)` = `false`

Case

Case logic expression. If the `base` is provided, returns the `then` expression of the first case in which `when` = `base`. Otherwise returns the first case in which `when` is `true`. If no case is satisfied returns `else` if it is provided, or `NULL` otherwise.

Property	Value
`type`	`"case"`
`base`	expression (optional)
`cases`	`array` of case options
`else`	expression (optional)

Case Option

Property	Value
`when`	expression
`then`	expression

Collate

Performs the MySQL `COLLATE` function.

Property	Value
`type`	`"collate"`
`e`	expression
`collation`	`string`

Count Rows

Performs the MySQL `COUNT(*)` function.

Property	Value
`type`	`"count_rows"`

Example

```
{ "type": "count_rows" }
```

Exists

Returns true if the enclosed `SELECT` returns at least one row.

Property	Value
`type`	`"exists"`
`select`	select

Supports shorthand syntax with the `$exists` property.

Property	Value
`$exists`	select

Function

Performs a MySQL function. The number of arguments varies depending on the function.

Property	Value
`type`	`"function"`
`function`	`string`
`args`	array of expressions

Available functions are:

Name	Args	Aggregate	Description
`AVG`	1	yes	arithmetic average
`AVG_DISTINCT`	1	yes	arithmetic average of distinct values of argument
`BIT_AND`	1	yes	bit-wise AND
`BIT_OR`	1	yes	bit-wise OR
`BIT_XOR`	1	yes	bit-wise XOR
`CEIL`	1	yes	returns the smallest integer value not less than the argument
`COUNT`	1	yes	returns the number of rows in the which the argument is not `NULL`
`COUNT_DISTINCT`	`n`	yes	returns the number of distinct value(s) of the arguments

Name	Args	Aggregate	Description
`FLOOR`	1	yes	returns the largest integer value not greater than the argument
`MAX`	1	yes	returns the maximum value of the argument
`MIN`	1	yes	returns the minimum value of the argument
`POW`	2	no	
`STDDEV_POP`	1	yes	returns the population standard deviation of the argument
`STDDEV_SAMP`	1	yes	returns the sample standard deviation of the argument
`SUM`	1	yes	returns the sum of the argument
`SUM_DISTINCT`	1	yes	returns the sum of the distinct values of the argument
`TRUNCATE`	2	no	
`VAR_POP`	1	yes	returns the population variance of the argument
`VAR_SAMP`	1	yes	returns the sample variance of the argument

Supports shorthand syntax by prefixing any function name with `$$`. For example, `{ "$$pow": [2, 3] }` evaluates to `8`.

In

Returns true if an expression is contained in a set of values. If an empty array is provided for `values`, will always return false.

Property	Value
`type`	`"in"`
`e`	expression
`values`	array of expressions

Supports shorthand syntax with the `$in` property. Takes an array of a single expression (`e`), followed by either an array of expression(s) (`values`) or a `SELECT` object.

Property	Value
`$in`	array of one expression, then either an array of expressions or a select

In Select

Returns true if `e` is in the result of the `select` query.

Property	Value
`type`	`"in_select"`
`e`	expression
`select`	select

Supports shorthand syntax with the `$in` property (see above).

Select Expression

Returns the value of the first column in the first row of the result set of the query.

Property	Value
`type`	`"select"`
`select`	select

Supports shorthand syntax with the `$select` property.

Property	Value
`$select`	select

Unary Expression

Unary operator expression, evaluated as `op` `e`.

Property	Value
`type`	`"unary"`
`op`	`string`
`e`	expression

Valid unary operators are:

Operator	Description
`not`	logical NOT
`-`	negate
`~`	bit invert

Supports shorthand syntax with any operator by prefixing it with `$`. Takes a single expression as a value.

Property	Value

`$` `op`	expression

Select Syntax

The SELECT syntax is essentially a JSON representation of the MySQL `SELECT` syntax. See the MySQL documentation for more detailed information.

SELECT

The SELECT syntax is contained in a single JSON object.

Property	Value	Notes
distinct	`boolean`, default `false`	If `true`, only returns unique values
columns	result columns	If empty, returns all columns available from source
from	source	Source being selected from
where	expression	Condition for rows, where expression returns `true`
group	`array` of expressions	Used to group rows for aggregation functions
having	expression	Like `where`, but can filter aggregation results
order	`array` of order terms	Used to sort the results
limit	expression	Limit the number of rows returned
offset	expression	Offset of the start of the rows

Result Columns

Specifies the column(s) to select.

All

Specifies all columns from the source. This is the same as the MySQL `SELECT *` syntax. This is the default if no value for the `columns` property is set.

Property	Value
`type`	`"all"`

Example

```
{ "type": "all" }
```

Array

Specifies column(s) as an array of result column objects. This is provided directly as a JSON array.

Example as JSON array:

```
[ ... ]
```

Result Column

Specifies an expression and optional alias. The alias can be referenced in the where clause with an alias expression.

Property	Value
`e`	expression
`alias`	`string` (optional)

Source

A source is a SQL table (or virtual table) from which a `SELECT` statement loads data.

Table Source

A source from any table.

Property	Value
`type`	`"table"`
`table`	`string` table
`alias`	`string` (optional)

The table syntax is the same as the table portion of the column expression syntax

```
table         = system-table-name | database-table
database-table  = database-path ['@' database-table-name]
```

May also be provided directly as a JSON string (without the `alias` property).

System Table Source

> Deprecated

A source from a system table.

Property	Value
`type`	`"table_system"` or `"ts"`
`table`	`string` table name
`alias`	`string` (optional)

Database Table Source

A source from a database table.

Property	Value
`type`	`"table_database"` or `"td"`
`database`	database specifier
`table`	`string` table name
`alias`	`string` (optional)

Join Source

A source derived from a SQL join of two sources.

Property	Value
`type`	`"join"`
`op`	`"join"`, `"left"`, `"left_outer"`, `"inner"`, or `"cross"`
`s1`	left join source
`s2`	right join source

Select Source

Source from the result of a select statement.

Property	Value
`type`	`"select"`
`select`	select

Order Term

Specifies an expression and optional order.

Property	Value
`e`	expression
`order`	`"asc"` or `"desc"` (optional, default `"asc"`

Definitions Syntax

> Under Construction

Group

Defines a XINA group.

Property	Value
`name`	`string`
`desc`	`string`

Database

Defines a XINA database. The `name` and `fields` values are required, and at least one field must be provided. If `label` is not provided it will be the same as `name`.

Property	Value
`name`	`string`
`label`	`string` (optional)
`format`	`string` (optional)
`path`	`string` (optional)
`desc`	`string` (optional)
`dynamic`	`boolean` (optional, default `false`)
`event`	`boolean` (optional, default `false`)
`file`	`boolean` (optional, default `false`)
`link`	`boolean` (optional, default `false`)
`lock`	`boolean` (optional, default `false`)
`log`	`boolean` (optional, default `false`)
`notify`	`boolean` (optional, default `false`)
`subscribe`	`boolean` (optional, default `false`)
`tag`	`boolean` (optional, default `false`)
`track`	`boolean` (optional, default `false`)

Property	Value
`trash`	`boolean` (optional, default `false`)
`wall`	`boolean` (optional, default `false`)
`objects`	`object` (optional)
`files`	`object` (optional)
`fields`	`array` of fields
`blobs`	`array` of blobs (optional)
`indexes`	`array` of `string` values (optional)
`databases`	`array` of databases (optional)

Field

Defines a XINA database field. The `name` and `type` are required. If `label` is not provided it will be the same as `name`.

```
{
 "name"    : <string>,
 "label"   : <string>,  (optional)
 "type"    : <string>,
 "format"  : <string>,  (optional)
 "meas"    : <string>,  (optional)
 "unit"    : <string>,  (optional)
 "desc"    : <string>,  (optional)
 "def"     : <string>,  (optional)
 "ref"     : <string>,  (optional)
 "key"     : <boolean>, (optional, default false)
 "nul"     : <boolean>, (optional, default false)
 "strict"  : <boolean>, (optional, default false)
 "lock"    : <boolean>, (optional, default false)
 "options" : [ <[[#Field Option|field option]]>, ... ] (optional)
}
```

Field Option

A value option for a field. Regardless of the field type the value here should be a string representation of the actual value.

```
{
 "value" : <string>,
```

```
 "desc"  : <string>  (optional)
}
```

Blob

Defines a XINA database blob. The `name` is required. If `label` is not provided it will be the same as `name`.

```
{
 "name"    : <string>,
 "label"   : <string>, (optional)
 "desc"    : <string>, (optional)
 "nul"     : <boolean> (optional, default false)
}
```

Index

Defines an index on one or more columns of a database record table.

Action Index

ACCESS

Acquire temporary access ID for websocket connection.

ALTER

Edit schema or user properties.

APPROVE

Approve user requests.

CANCEL

Cancel one or more asynchronous tasks.

CLEAN

Delete one or more asynchronous task records and any associated files.

CONCLUDE

Explicitly conclude an asynchronous task.

CONTINUE

Attempt to continue import of file(s) generated by an asynchronous task.

CREATE

Create group, database, team, or user entities.

DELETE

Permanently delete one or more database records.

DESTROY

Cancel and clean one or more asynchronous tasks.

DISPOSE

Permanently delete one or more database records.

DOWNLOAD

Acquire temporarly download links to one or more files.

DROP

Permanently delete a group, team, database, or cron.

ECHO

Echo a provided string (used for debugging).

EDIT

Edit a wall post.

FETCH

Load a variety of data types in JSON friendly formats.

FOLLOW

Assign one or more walls to be followed by a user.

GRANT

Grant one or more users permissions on one or more groups or databases.

IF

Perform an action conditionally based on the result of a query.

INSERT

Insert one or more records into a database.

INSERT SELECT

Insert records into a database from an arbitrary query.

INVOKE

Invoke a synchronous AWS Lambda function.

JOIN

Join one or more users, groups, or databases to one or more teams.

KEY

Create or delete an API key for a user.

KILL

Terminate one or more asynchronous tasks.

LEAVE

Remove one or more users, groups, or databases from one or more teams.

LINK

Create directional links from one or more records to one or more records.

LOAD

High performance record insertion using the MySQL LOAD DATA INFILE operation.

LOCK

Lock one or more records in a database.

MOVE

Rename an object in the general object store.

PAUSE

Pause execution of one or more asynchronous task sequences.

POST

Send a post to a wall.

PREFER

Apply one or more preferences for a user.

PULL

Permanently delete a post from a wall.

REJECT

Deny a user request.

REPLACE

Insert one or more records into a database, replacing existing records with matching keys.

REPLACE SELECT

Insert records into a database from an arbitrary SELECT query, replacing existing records with matching keys.

REQUEST

Request an action to be performed by a user with required permissions.

RESET

Resets a database to the initial state, permanently deleting all records.

RESTORE

Restores one or more trashed records to the active state.

RESUME

Resume execution of one or more asynchronous task sequence(s).

RETRACT

Retract one or more user requests.

REVOKE

Revoke one or more permissions on one or more groups or databases from one or more users.

RETRY

Restart a locked asynchronous task.

RUN

Run one or more asynchronous tasks.

SCHEMA

Returns the complete current group and database schema.

SELECT

Performs a SQL SELECT query.

SET

Inserts one or more records into a database, replacing **all** records already present in the database.

SIGN

Sign one or more records.

STORE

Add or remove data from the general object store.

STRUCT ALTER

Alter configuration of struct group or database.

STRUCT BUFFER EMPTY

Delete mnemonic buffer records by partition.

STRUCT BUFFER IMPORT

Import a single mnemonic buffer file.

STRUCT BUFFER UPDATE

Queue one or more updates to modify buffer file(s).

STRUCT CREATE

Create struct schema elements.

STRUCT EVENT

Insert, close, or update struct event record(s).

STRUCT EVENT IMPORT

Import struct event definition(s).

STRUCT MN ALIAS

Adds one or more aliases to a struct mnemonic.

STRUCT MN EDIT

Edits system-managed fields of a struct mnemonic.

STRUCT MN MERGE

Merges one or more struct mnemonics into a single mnemonic.

STRUCT MN RANGE IMPORT

Import the time range of one or more mnemonics.

STRUCT PIPELINE

Perform struct pipe processing, launching asyncronous task(s) as needed.

STRUCT UNMINE

Delete mined event and mnemonic data from a pipe, either by archive or partition.

SUBSCRIBE

Subscribes one or more users to notifications from one or more walls.

SYNC

Echo a provided string (used for debugging).

TAG

Applies one or more tags to one or more records in a single database.

TEAMS

Returns the complete team schema.

TRASH

Trashes one or more records in a database.

UNFOLLOW

Remove one or more walls from being followed by one or more users.

UNLINK

Permanently delete links between records.

UNLOCK

Unlock one or more records in a database.

UNSIGN

Remove signature from a record.

UNSUBSCRIBE

Remove subscriptions from one or more walls for one or more users.

UNTAG

Remove one or more tags from one or more records in a single database.

UPDATE

Update the fields, blobs, or file of one or more records in a single database.

UPDATE EXPRESSION

Update the fields, blobs, or file of one or more records in a single database, supporting field values defined as arbitrary expressions.

VERSION

Returns the current XINA version information.

VIEW

Marks tasks or notifications as viewed by a user.

WAIT

Waits a specified period of time and returns (used for debugging).

XDOWNLOAD

Generates a XINA Download utility file for set of file downloads.