Ichno Documentation

Ichno Solutions

Jul 22, 2020

GENERAL VIEW

Ichno is a solution, provided as a Sass model, to facilitate tracking changes and the history storage of object instances or database entries.

You only need to send the instance of the object, with its current state, and the changes will be automatically detected and will be available for later searches.

It will provide the following features:

Track the changes easily

Centralized history logic. Save your team time and computational resources with history feature implementation.

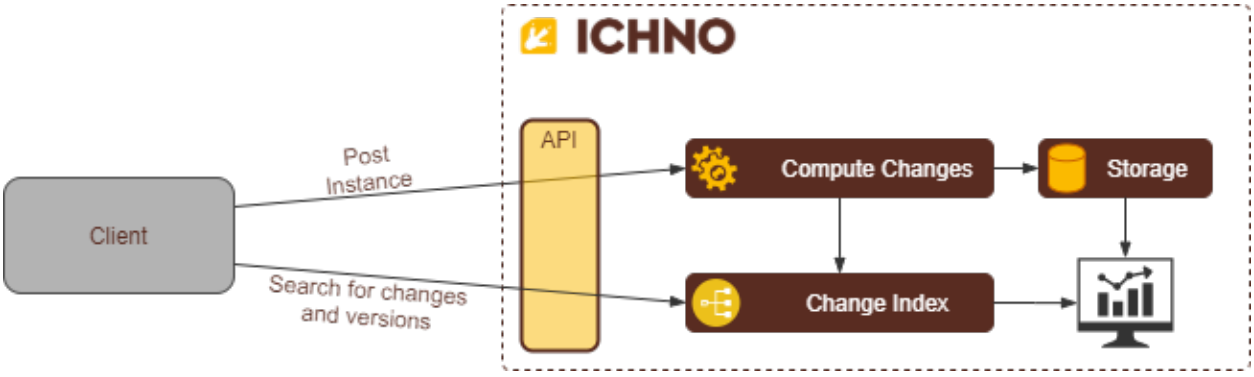
Lógica centralizada de histórico. Economize tempo da equipe técnica e de recursos computacionais com a implementação de funcionalidades de histórico.

Search for versions and changes

Through a simple interface, or an API to integrate, search for changes in instances of database records.

CHAPTER 1

General View



CHAPTER 2

Tips

2.1 Keys

2.1.1 Key on registering instances

Keys are used to identify a single instance of your objects. As you can register versions from many subsets of instances, you should send the name of instance subset as a part of key, to avoid keys conflicts. This subset can be the table or package name. For example, if you have a subset of users and another subset of products, the correct way to send this entities, through API, is:

User 1

```
{
  keys: [
    { subset: 'user' },
    { id: 1 }
  ],
  object: {
    name: 'User Name',
    ...
  }
}
```

User 2

```
{
  keys: [
    { subset: 'user' },
    { id: 2 }
  ],
  object: {
    name: 'User Name',
    ...
  }
}
```

Product 1	Product 2
{ keys: [{ subset: 'product' }, { id: 1 }], object: { name: 'Product Name', ... } }	{ keys: [{ subset: 'product' }, { id: 2 }], object: { name: 'Product Name', ... } }

2.1.2 Key on querying instances

You can query by keys using only a subset of keys registered with your instances. In the same example used above, if you need ti query all changes made in the 'user' subset, you can use the following json to get that:

```
{
  keys: [
    { subset: 'user' }
  ]
}
```

2.2 Labels

Labels are useful to add some data that you want to query for. For example, if you want to register the user who made those changes, you can use labels, allowing you to search for all change made by a user.

CHAPTER 3

Open API Specification

Ichno provides an API to be used to integrate easily with. The swagger definition can be accessed . Remember that you need an API Key to communicate, even using the Swagger Interface.

CHAPTER 4

API Key

To post and query versions, you need to send your api-key through the header, in the `X-API-KEY` entry. You can see how to manage your keys at Api Keys.

CHAPTER 5

Posting Versions

Two endpoints to register instance versions is provided:

1. **Asynchronous Post** `POST /api/v1/version/async`
Using this endpoint, the api will response almost instantly, and the version will be processed asynchronously, and will be ready to be queried in few seconds.

2. **Synchronous Post** `POST /api/v1/version`
Using this endpoint, the version will be processed synchronously with a bigger response time. Use this endpoint if you need this versions ready to be queried right after the register post response.

5.1 Post Struct

More than only your object instance, you can send other fields to store aditional information about the posted version?

Field	Type	Required	Description
id	`UUID`	**Yes**	The identifier for this version. This id must be generated by client. It allows the client application to refer to the posted version, in a asynchronous scenario, even if is not processed by Ichno yeat.
object	`object`	**Yes**	The instance to be register. Send the current state of this instance, after all changes applied. Ichno will compare with the version registered previously and compute all changes.
metadata	`object`	No	Additional data to this version. Use this field to add some additional data to your version. The values in this field **cannot** be queried, but can be retrieved when you need.
keys	`[string]: string | number | boolean` (Key Value List)	**Yes**	Unique identifier of the instance in the clients system. Send the id's of this instance. It is a list to allow sending objects with composite keys. If you are registering versions from diferent instance types, you must send a type identifier as a key too, to avoid conflicts between too instances, with diferent types and same identifier.
labels	`[string]: string | number | boolean` (Key Value List)	**No**	Version label. Send labels to this version. Labels are additional data for this version but, diferent from metadate, these labels can be used as a filter. You can use this, for example, to send the user id who is changing the instance, enabling you to filter all changes made by a specific user.

CHAPTER 6

Quering Changes

After have your instance versions registered, it is possible to query changes using the following parameters:

Field	Type	Required	Description
keys	`[string]: string | number | boolean` (Key Value List)	**No**	Unique identifier of the instance in the clients system. Use the same values used to register the instance. If you are using composite keys, its possible to send all keys or a subset of these keys.
startDate	`date` (yyyy-MM-ddThh:mm:ss)	**No**	Start date for date range.
endDate	`date` (yyyy-MM-ddThh:mm:ss)	**No**	End date for date range.
labels	`[string]: string | number | boolean` (Key Value List)	**No**	Change labels.
properties	`object`	**No**	Filter changes by properties
start	`integer`	**No**	Number of register to skip on this query. Useful to pagination.
length	`integer`	**No**	Number of register that must be returned. Useful to pagination.

6.1 Query by properties

Ichno also allow you to query by properties, allowing you to query changes made on specific property. Properties query parameters are send through an object in the property `properties` on change query endpoint:

Field	Type	Required	Description

path	`[string | number | boolean]` (array)	**No**	Property path.
			For example, if you are registering version for the following object:
			`{ name: 'John Hanson', address: { street: 'The Great, Ave', number: 153 } }`
			You can query the name changes using the following array as parameter:
			`['name']`
			Or this one to filter by street name changes:
			`['address', 'street']`

newValue	`string | number | boolean`	**No**	Filter properties by the new value.
oldValue	`string | number | boolean`	**No**	Filter properties by the old value.

CHAPTER 7

Aggregations

Aggregations provide aggregated data based on a search query. It is based on simple building blocks called aggregations, that can be composed in order to build complex summaries of the data.

An aggregation can be seen as a unit-of-work that builds analytic information over a set of versions. The context of the execution defines what this document set is (e.g. a top-level aggregation executes within the context of the executed query/filters of the search request).

7.1 Structuring Aggregations

The following snippet captures the basic structure of aggregations:

```
{
    "aggregations" : {
        "<aggregation_name>" : {
            "aggregationTarget": "<aggregation_target>",
            "aggregationType: "<aggregation_type>"
            [,"parameters" : {
                ["<parameter_1>" : { ... } ]*
            } ]?
            [,"aggregations" : { [<sub_aggregation>]+ } ]?
        }
        [,"<aggregation_name_2>" : { ... } ]*
    }
}
```

7.2 Aggregations Types

There are many different types of aggregations, each with its own purpose and output.

7.2.1 Date Histogram by Version

The date histogram by version shows the number of versions posted in a specific date value within you changes dataset.

Parameters

Name	Type	Required	Description
format	`string`	**No**	Se avaiable formats in *Date Format/Pattern*
interval	`string`	**No**	Interval period for aggregation. Avaiable values are: - Second - Minute - Hour - Day - Week - Month - Quarter - Year

Example

Request

```
{
  "aggregations": {
    "versions_datehistogram": {
      "aggregationType": "VersionDateHistogram",
      "parameters": {
        "format": "yyyy",
        "interval": "Year"
      }
    }
  }
}
```

Response

```
{
  "aggregations": {
    "versions_datehistogram": {
      "key": null,
      "count": 307065,
      "results": [
        {
          "key": "2019",
          "count": 155095,
          "results": null,
```

(continues on next page)

(continued from previous page)

```
          "aggregations": null
        },
        {
          "key": "2020",
          "count": 151970,
          "results": null,
          "aggregations": null
        }
      ],
      "aggregations": null
    }
  }
}
```

7.2.2 Property Name

Aggregates by properties.

Parameters

Name	Type	Required	Description
jsonPath	`string`	**No**	Filter the properties that must be included in aggregation. Read about Json Paths in *JSON Path*.
depth	`integer`	**No**	If the object contains objects as properties, you can define how many levels should be included in the aggregation.
size	`integer`	**No**	Number of aggregation to be returned

Example

Request

```
{
  "length": 0,
  "aggregations": {
    "properties_names": {
      "aggregationType": "Property",
      "parameters": {
        "depth": 0,
        "jsonPath": "$['Documents']['[^('\\])]*']",
        "size": 5
      }
    }
  }
}
```

Response

```
{
  "total": 315025,
  "data": [],
  "aggregations": {
    "properties_names": {
      "count": 115814,
      "results": [
        {
          "key": "$['Documents']['edc17b99-d956-4300-8b0e-8aa60f9cdb94']",
          "count": 23818
        },
        {
          "key": "$['Documents']['8dfc2491-dab0-4d36-8f1a-bf96d4002e91']",
          "count": 23568
        },
        {
          "key": "$['Documents']['b93acc29-f93c-4968-903c-6fa961497969']",
          "count": 23545
        },
        {
          "key": "$['Documents']['ba86b58b-1899-4289-b191-b638586eddf9']",
          "count": 22459
        },
        {
          "key": "$['Documents']['7b526f64-413e-4dca-8120-22ab98b33ab8']",
          "count": 22424
        }
      ]
    }
  }
}
```


7.2.3 Label Values

Aggregates label values.

Parameters

Name	Type	Required	Description
name	`string`	**No**	Label name. Will include only values with this label name.
size	`integer`	**No**	Number of aggregation to be returned

Example

Request


```
{
  "length": 0,
  "aggregations": {
    "labels_aggregations": {
      "aggregationType": "LabelValue",
```


(continues on next page)

(continued from previous page)

```
      "parameters": {
        "name": "userName",
        "size": 5
      }
    }
  }
}
```

Response

```
{
  "total": 315025,
  "data": [],
  "aggregations": {
    "labels_aggregations": {
      "count": 2920,
      "results": [
        {
          "key": "Adriano Queiroz",
          "count": 810
        },
        {
          "key": "Mayara Caldeira",
          "count": 743
        },
        {
          "key": "Priscila Batista",
          "count": 603
        },
        {
          "key": "Maria Eduarda",
          "count": 397
        },
        {
          "key": "Bryan Santos",
          "count": 367
        }
      ]
    }
  }
}
```

CHAPTER 8

Discover View

With Discover View you can filter versions, view the changes, the posted json instance and navigate through a change chain.

8.1 Changes Tree

In changes tree view you can navigate through all posted versions and their labels and metadata.

8.2 Json Version

With json view, you can see the json of the instance posted to register the version.

8.3 Chain View

In chain view, you can navigate through versions from the same instance and see the changes made on properties.

8.4 Quering Versions

Using filters you can filter by keys, labels and properties values. Use multiple filters to look for the exact instance that you want.

CHAPTER 9

User Permissions

You can grant permission to other users to access Ichno. You only need to provide a valid e-mail, and Ichno will invite him to sign up and access your dashboard. User administration is accessed through Settings menu.

CHAPTER 10

API Keys

API Keys allow other systems to integrate with Ichno, through a API, easily. You can manage API Keys through Settings menu.

CHAPTER 11

Date Format/Pattern

Note: this information was copied from

All ASCII letters are reserved as format pattern letters, which are defined as follows:

Stymbol	Meaning	Presentation	Examples
G	era	text	AD; Anno Domini; A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul; July; J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday; T
e/c	localized day-of-week	number/text	2; 02; Tue; Tuesday; T
F	week-of-month	number	3
a	am-pm-of-day	text	PM
h	clock-hour-of-am-pm (1-12)	number	12

Continued on next page

Table 1 – continued from previous page

Stymbol	Meaning	Presentation	Examples
K	hour-of-am-pm (0-11)	number	0
k	clock-hour-of-am-pm (1-24)	number	0
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30
s	second-of-minute	number	55
S	fraction-of-second	fraction	978
A	milli-of-day	number	1234
n	nano-of-second	number	987654321
N	nano-of-day	number	1234000000
V	time-zone ID	zone-id	America/Los_Angeles; Z; -08:30
z	time-zone name	zone-name	Pacific Standard Time; PST
O	localized zone-offset	offset-O	GMT+8; GMT+08:00; UTC-08:00;
X	zone-offset Z for zero	offset-X	Z; -08; -0830; -08:30; -083015; -08:30:15;
x	zone-offset	offset-x	+0000; -08; -0830; -08:30; -083015; -08:30:15;
Z	zone-offset	offset-Z	+0000; -0800; -08:00;
p	pad next	pad modifier	1
‘	escape for text	delimiter	‘’
single quote	literal	‘	[
optional section start	]	optional section end	#
reserved for future use	{	reserved for future use	}

The count of pattern letters determines the format.

11.1 Text

The text style is determined based on the number of pattern letters used. Less than 4 pattern letters will use the short form. Exactly 4 pattern letters will use the full form. Exactly 5 pattern letters will use the narrow form. Pattern letters L, c, and q specify the stand-alone form of the text styles.

11.2 Number

If the count of letters is one, then the value is output using the minimum number of digits and without padding. Otherwise, the count of digits is used as the width of the output field, with the value zero-padded as necessary. The following pattern letters have constraints on the count of letters. Only one letter of c and F can be specified. Up to two letters of d, H, h, K, k, m, and s can be specified. Up to three letters of D can be specified.

11.3 Number/Text

If the count of pattern letters is 3 or greater, use the Text rules above. Otherwise use the Number rules above.

11.4 Fraction

Outputs the nano-of-second field as a fraction-of-second. The nano-of-second value has nine digits, thus the count of pattern letters is from 1 to 9. If it is less than 9, then the nano-of-second value is truncated, with only the most significant digits being output.

11.5 Year

The count of letters determines the minimum field width below which padding is used. If the count of letters is two, then a reduced two digit form is used. For printing, this outputs the rightmost two digits. For parsing, this will parse using the base value of 2000, resulting in a year within the range 2000 to 2099 inclusive. If the count of letters is less than four (but not two), then the sign is only output for negative years as per SignStyle.NORMAL. Otherwise, the sign is output if the pad width is exceeded, as per SignStyle.EXCEEDS_PAD.

11.6 ZoneId

This outputs the time-zone ID, such as Europe/Paris. If the count of letters is two, then the time-zone ID is output. Any other count of letters throws IllegalArgumentException.

11.7 Zone names

This outputs the display name of the time-zone ID. If the count of letters is one, two or three, then the short name is output. If the count of letters is four, then the full name is output. Five or more letters throws IllegalArgumentException.

11.8 Offset X and x

This formats the offset based on the number of pattern letters. One letter outputs just the hour, such as +01, unless the minute is non-zero in which case the minute is also output, such as +0130. Two letters outputs the hour and minute, without a colon, such as +0130. Three letters outputs the hour and minute, with a colon, such as +01:30. Four letters outputs the hour and minute and optional second, without a colon, such as +013015. Five letters outputs the hour and minute and optional second, with a colon, such as +01:30:15. Six or more letters throws IllegalArgumentException. Pattern letter X (upper case) will output Z when the offset to be output would be zero, whereas pattern letter x (lower case) will output +00, +0000, or +00:00.

11.9 Offset O

This formats the localized offset based on the number of pattern letters. One letter outputs the short form of the localized offset, which is localized offset text, such as GMT, with hour without leading zero, optional 2-digit minute

and second if non-zero, and colon, for example GMT+8. Four letters outputs the full form, which is localized offset text, such as GMT, with 2-digit hour and minute field, optional second field if non-zero, and colon, for example GMT+08:00. Any other count of letters throws IllegalArgumentException.

11.10 Offset Z

This formats the offset based on the number of pattern letters. One, two or three letters outputs the hour and minute, without a colon, such as +0130. The output will be +0000 when the offset is zero. Four letters outputs the full form of localized offset, equivalent to four letters of Offset-O. The output will be the corresponding localized offset text if the offset is zero. Five letters outputs the hour, minute, with optional second if non-zero, with colon. It outputs Z if the offset is zero. Six or more letters throws IllegalArgumentException.

11.11 Optional section

The optional section markers work exactly like calling DateTimeFormatterBuilder.optionalStart() and DateTimeFormatterBuilder.optionalEnd().

11.12 Pad modifier

Modifies the pattern that immediately follows to be padded with spaces. The pad width is determined by the number of pattern letters. This is the same as calling DateTimeFormatterBuilder.padNext(int).

For example, ppH outputs the hour-of-day padded on the left with spaces to a width of 2.

Any unrecognized letter is an error. Any non-letter character, other than [,], {, }, # and the single quote will be output directly. Despite this, it is recommended to use single quotes around all characters that you want to output directly to ensure that future changes do not break your application.

CHAPTER 12

JSON Path

Json paths is used in Ichno to identify an property uniquelly. It's important to distinguish that the concept applied in Ichno is different from other libraries avaiable, which use json paths as a language to filter properties and objects inside an json object.

12.1 Examples

Given the json.

```
{
  "store": {
      "book": [
          {
              "category": "reference",
              "author": "Nigel Rees",
              "title": "Sayings of the Century",
              "price": 8.95
          },
          {
              "category": "fiction",
              "author": "Evelyn Waugh",
              "title": "Sword of Honour",
              "price": 12.99
          }
      ],
      "bicycle": {
          "color": "red",
          "price": 19.95
      }
  },
  "expensive": 10
}
```

Json Path	Property Value
`$['expensive']`	10
`$['store']['book'][0]['author']`	“Nigel Rees”
`$['store']['book'][1]['price']`	12.99
`$['store']['book']['bicycle']['bicycle'`	“red”

12.2 JSON Path Regex

In some queries and aggregations, is possible to use regex in properties names to expand the possibilities of filter.

Given the following sample:

```
{
  "store": {
      "book": [
          {
              "category": "reference",
              "sub-category": "reference",
              "author": "Nigel Rees",
              "title": "Sayings of the Century",
              "price": 8.95
          },
          {
              "category": "fiction",
              "sub-category": "fiction",
              "author": "Evelyn Waugh",
              "title": "Sword of Honour",
              "price": 12.99
          }
      ]
  },
  "expensive": 10
}
```

If you use the regex `$['store']['book'][.*]['.*category']` your query will include all values of ‘category’ or ‘sub-category’ book properties in results.

For more information about regular expressions, see *Regular expression syntax*.

CHAPTER 13

Regular expression syntax

Note: this information was based on

A is a way to match patterns in data using placeholder characters, called operators.

Ichno uses 's regular expression engine to parse these queries.

13.1 Reserved characters

Lucene's regular expression engine supports all Unicode characters. However, the following characters are reserved as operators::

```
. ? + * | { } [ ] ( ) " \
```

Depending on the optional operators enabled, the following characters may also be reserved::

```
# @ & < >  ~
```

To use one of these characters literally, escape it with a preceding backslash or surround it with double quotes. For example::

```
\@                  # renders as a literal '@'
\\                  # renders as a literal '\'
"john@smith.com"    # renders as 'john@smith.com'
```

13.2 Standard operators

Lucene's regular expression engine does not use the library, but it does support the following standard operators.

. Matches any character. For example:

ab. # matches ‘aba’, ‘abb’, ‘abz’, etc.

? Repeat the preceding character zero or one times. Often used to make the preceding character optional. For example:

abc? # matches ‘ab’ and ‘abc’

+ Repeat the preceding character one or more times. For example:

ab+ # matches ‘ab’, ‘abb’, ‘abbb’, etc.

***** Repeat the preceding character zero or more times. For example:

ab* # matches ‘a’, ‘ab’, ‘abb’, ‘abbb’, etc.

{} Minimum and maximum number of times the preceding character can repeat. For example:

a{2} # matches ‘aa’ a{2,4} # matches ‘aa’, ‘aaa’, and ‘aaaa’ a{2,} # matches ‘a’ repeated two or more times

| OR operator. The match will succeed if the longest pattern on either the left side OR the right side matches. For example:

abc|xyz # matches ‘abc’ and ‘xyz’

(...) Forms a group. You can use a group to treat part of the expression as a single character. For example:

abc(def)? # matches ‘abc’ and ‘abcdef’ but not ‘abcd’

[...] Match one of the characters in the brackets. For example:

[abc] # matches ‘a’, ‘b’, ‘c’

Inside the brackets, - indicates a range unless - is the first character or escaped. For example:

[a-c] # matches ‘a’, ‘b’, or ‘c’ [-abc] # ‘-‘ is first character. Matches ‘-‘, ‘a’, ‘b’, or ‘c’ [abc-] # Escapes ‘-‘. Matches ‘a’, ‘b’, ‘c’, or ‘-‘

A ^ before a character in the brackets negates the character or range. For example:

[^abc] # matches any character except ‘a’, ‘b’, or ‘c’ [^a-c] # matches any character except ‘a’, ‘b’, or ‘c’ [^-abc] # matches any character except ‘-‘, ‘a’, ‘b’, or ‘c’ [^abc-] # matches any character except ‘a’, ‘b’, ‘c’, or ‘-‘

13.3 Unsupported operators

Lucene’s regular expression engine does not support anchor operators, such as ^ (beginning of line) or $ (end of line). To match a term, the regular expression must match the entire string.