

Advanced Energy System Design (AESD): Technical Manual for the Records API

Nicholas Brunhart-Lupo Brian Bush Kenny Gruchalla
Michael Rossol
National Renewable Energy Laboratory

18 July 2017

Contents

1	Abstract	1
2	Overview	2
3	Use Cases	4
3.1	Static Data	4
3.2	Dynamic Data	8
3.3	Simulations	10
3.4	Bookmarks	11
3.5	Filtering	13
4	Records API, Version 4	14
4.1	Message Groups	14
4.2	General conventions	16
4.3	Messages	17
4.4	Scalar Value Types	27
5	Appendices	28
5.1	Protocol Buffers for Records API Version 4	28
6	References	33

1 Abstract

The Records API (application program interface) for Advanced Energy System Design (AESD) enables software that serves multidimensional record-oriented data to interoperate with software than uses such data. In the context of the

Records API, multidimensional data records are simply tuples of real numbers, integers, and character strings, where each data value is tagged by a variable name, according to a pre-defined schema, and each record is assigned a unique integer identifier. Conceptually, these records are isomorphic to rows in a relational database, JSON objects, or key-value maps. Records servers might supply static datasets, sensor measurements that periodically update as new telemetry become available, or the results of simulations as the simulations generate new output. Records client software might display or analyze the data, but in the case of simulations the client request the creation of new ensembles for specified input parameters. It is also possible to chain records clients and servers together so that a client consuming data from a server might transform that data and serve it to further clients.

This minimalist API avoids imposing burdensome metadata, structural, or implementation requirements on developers by relying on open-source technologies that is readily available for common programming languages. In particular, the API has been designed to place the smallest possible burden on services that provide data. This document defines the message format for the Records API, a transport mechanism for communicating the data, and semantics for interpreting it. The message format is specified as Google Protocol Buffers (Google Developers 2017b) and the transport mechanism uses WebSockets (Internet Engineering Task Force 2017). We discuss three major use cases for serving and consuming records data: (i) static data, (ii) dynamically augmented data, (iii) on-demand simulations, (iv) with filters, and (v) with bookmarks. Separate implementations of the API exist in C++, Haskell, JavaScript, Python, and R.

2 Overview

Client-server communication in the Records API simply consists of clients sending **Request** messages to the server and servers asynchronously sending **Response** messages to the client. The request and response messages hold the specifics of the request or response and the responses are correlated with the requests, but it is important to note that multiple responses may occur for a single request, as when record data is chunked into multiple response, or that an error response may be sent at any time. The nested messages within **Request** and **Response** may in turn contain nested fields and messages providing further details. The table below shows the correspondence between requests and responses, while the figure following that shows the containment relationships between message types.

Table 1: Correlation between requests and responses.

Request Field	Response Field
<code>models_metadata</code>	<code>models</code> or <code>error</code>

Request Field	Response Field
records_data	data or error
bookmark_meta	bookmarks or error
save_bookmark	bookmarks or error
cancel	no response or error
work	data or error

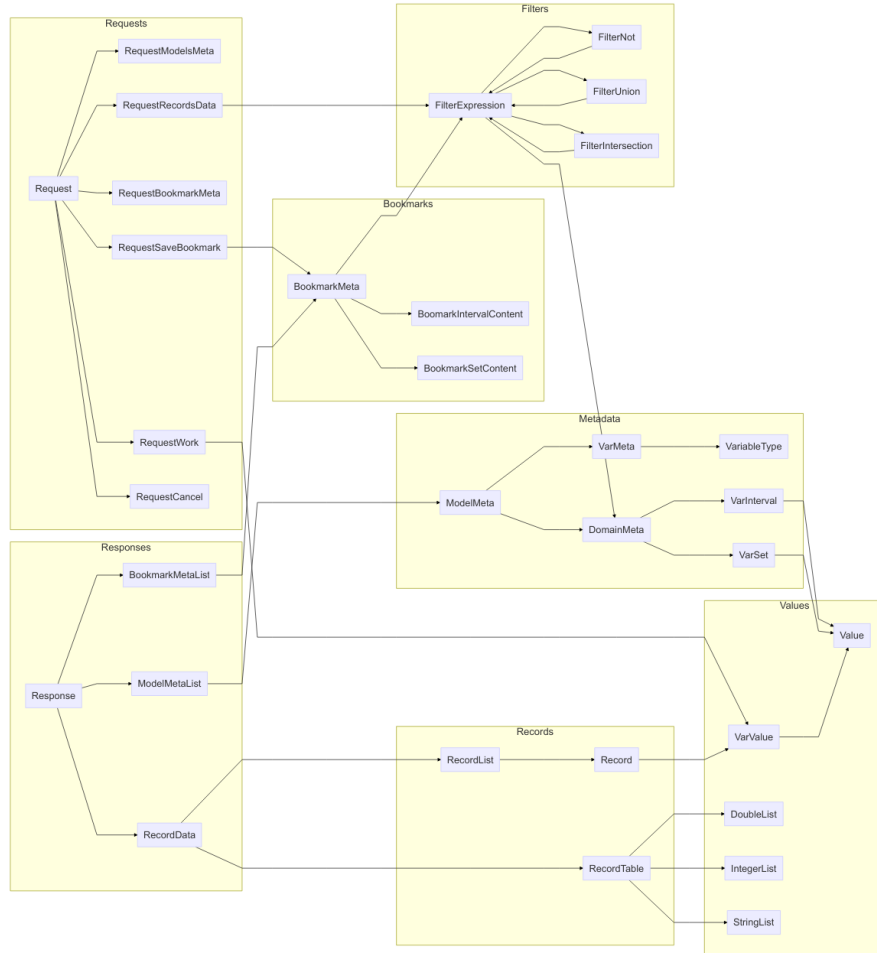


Figure 1: Containment relationships between protocol buffer messages in the AESD Records API.

Metadata messages describe “models”, which are just sources of data, and the variables they contain. Data record messages hold the data itself. Data records

are simply tuples of real numbers, integers, and character strings, where each data value is tagged by a variable name, according to a pre-defined schema, and each record is assigned a unique integer identifier. Conceptually, these records are isomorphic to rows in a relational database, JSON objects, or key-value maps. For efficiency and compactness, **RecordData** may be provided in list format or tabular format, with the latter format only obtained when the contents of the table all have the same data type. The data records may be provided *in toto* or filtered using filter messages so that only certain fields or records are returned. The API contains a small embedded language for filtering via set and value operations. Sets of records may be bookmarked for sharing or later retrieval by (i) enumerating their unique record identifiers, (ii) defining a range of unique record identifiers, or (iii) specifying a filtering criterion.

Servers that perform computations or simulations can receive input parameters via a **RequestWork** message that contains those input parameters. After the server has completed its computations, it sends the results as **RecordData** messages.

In general the response to a request for data records comes in *chunks* numbered in sequence, where each chunk has an identifier, **chunk_id**, and specifies the identifier of the next chunk, **next_chunk_id**. Thus, the chunks form a linked list. The sending of additional chunks can be cancelled using a **RequestCancel** message. If the **subscribe** flag is set when making a request, then the server will respond indefinitely with additional data as the data becomes available, until the subscription is cancelled.

3 Use Cases

In this section we outline some standard use cases for the Records API. UML Sequence Diagrams (Fowler 2017) illustrate the flow of messages and the messages themselves are printed in the text format output by the Google **protoc** tool (Google Developers 2017a).

3.1 Static Data

The retrieval of static data records forms the simplest use case for the Records API. A user chooses a particular data source (a “model” in the parlance of the Records API) and then the data is retrieved and displayed. The visualization client software communicates with a Records server, which in turn accesses the static data. The figure below illustrates the process.

A **Request** without **model_id** specified requests the server to list all models:

```
version: 4
id: 1
```

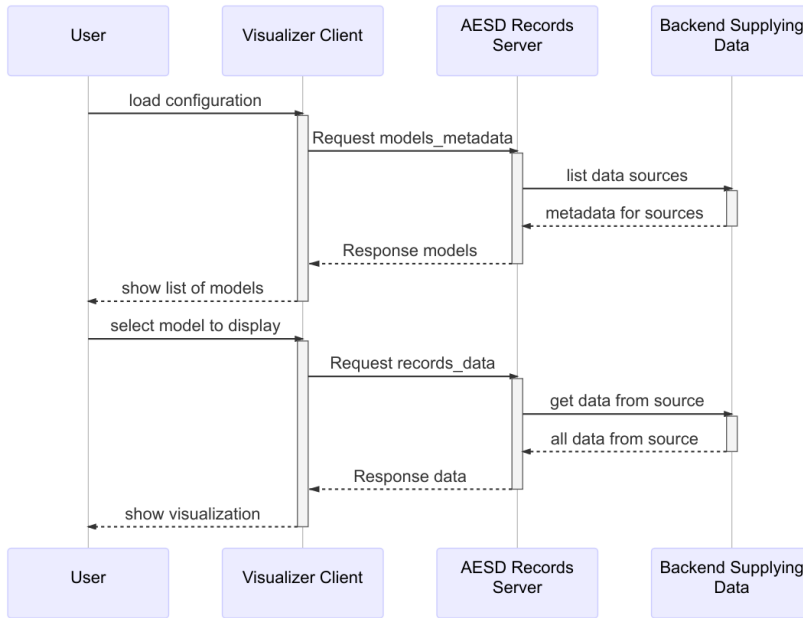


Figure 2: Visualizing data from a static source using the AESD Records API.

```
models_metadata {
}
```

The Response from the server provides metadata for all of the models:

```
version: 4
id: 1
models {
  models {
    model_id: "example-model-1"
    model_name: "Example Model #1"
    model_uri: "http://esda.nrel.gov/examples/model-1"
    variables {
      var_id: 0
      var_name: "Example Real Variable"
      type: REAL
    }
  }
  variables {
    var_id: 1
    var_name: "Example Integer Variable"
    type: INTEGER
  }
  variables {
```

```

        var_id: 2
        var_name: "Example String Variable"
        type: STRING
    }
models {
    model_id: "example-model-2"
    model_name: "Example Model #2"
    model_uri: "http://esda.nrel.gov/examples/model-2"
    variables {
        var_id: 0
        var_name: "POSIX Epoch"
        type: INTEGER
    }
    variables {
        var_id: 1
        var_name: "Measurement"
        type: REAL
    }
}
models {
    model_id: "example-simulation-3"
    model_name: "Example Simulation #3"
    model_uri: "http://esda.nrel.gov/examples/simulation-3"
    variables {
        var_id: 0
        var_name: "Input"
        type: REAL
    }
    variables {
        var_id: 1
        var_name: "Time"
        type: REAL
    }
    variables {
        var_id: 2
        var_name: "Value"
        type: REAL
    }
    inputs {
        var_id: 0
        interval {
            first_value: 0
            second_value: 100
        }
    }
}
}

```

```
}
```

Note that the response above is tagged with the same `id` as the request: this allows the client to correlate responses with the particular requests it makes. Next the user might request three records from the first model:

```
version: 4
id: 2
records_data {
  model_id: "example-model-1"
  max_records: 3
}
```

The record data might be returned as two chunks, where the first chunk is

```
version: 4
id: 2
chunk_id: 1
next_chunk_id: 2
data {
  list {
    records {
      record_id: 10
      variables {
        var_id: 0
        value: 10.5
      }
      variables {
        var_id: 1
        value: -5
      }
      variables {
        var_id: 2
        value: "first"
      }
    }
  }
  records {
    record_id: 20
    variables {
      var_id: 0
      value: 99.2
    }
    variables {
      var_id: 1
      value: 108
    }
    variables {
      var_id: 2
```

```

        value: "second"
      }
    }
  }
}

```

and the last chunk is:

```

version: 4
id: 2
chunk_id: 2
next_chunk_id: 0
data {
  list {
    records {
      record_id: 30
      variables {
        var_id: 0
        value: -15.7
      }
      variables {
        var_id: 1
        value: 30
      }
      variables {
        var_id: 2
        value: "third"
      }
    }
  }
}
}

```

3.2 Dynamic Data

As shown in the following figure retrieving data from a dynamic source proceeds quite similarly to retrieving data from a static source. The only essential difference is that the server repeatedly sends additional responses containing new data, until a request to cancel is sent:

When requesting dynamic data, it is advisable to set the `subscribe` flag in the request for data:

```

version: 4
id: 2
subscribe: true
records_data {
  model_id: "example-model-2"
}

```

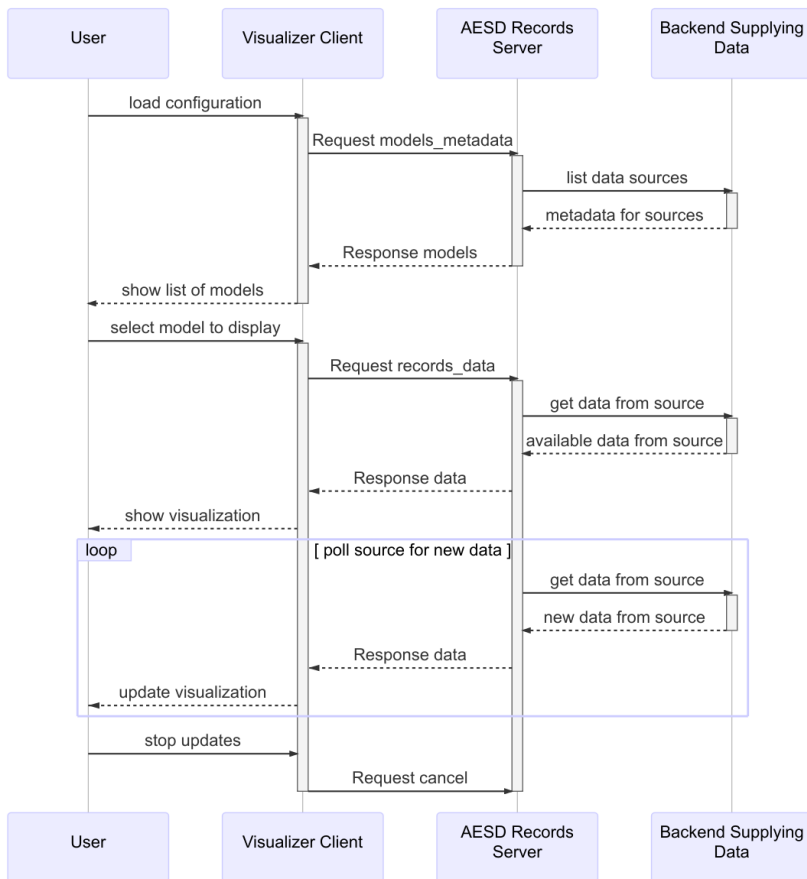



Figure 3: Visualizing data from a dynamic source using the AESD Records API.

```
}

```

The **RequestCancel** message is the **cancel** field **Request** and must include the **id** of the request to be cancelled:

```
version: 4
cancel {
  id: 2
}
```

3.3 Simulations

The model **Example Simulation #3** in the Static Data use case is a simulation model, as evidenced by the presence of the **inputs** field in its metadata. The following figure shows a typical interaction with a simulation-based model via the Records API.

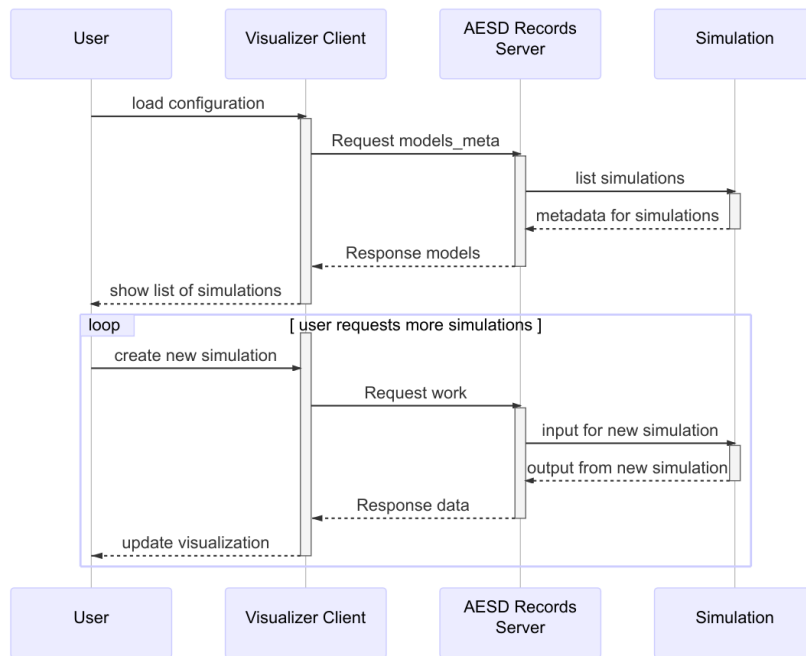


Figure 4: Steering and visualizing simulation results using the AESD Records API.

The **RequestWork** message, which is contained in the **work** field of a **Request**, specifies the input for a simulation to be run:

```
version: 4
```

```

id: 3
work {
  model_id: "example-simulation-3"
  inputs {
    var_id: 0
    value: 50
  }
}

```

The response to this message will be data for the result of the simulation.

3.4 Bookmarks

Once data from a model is loaded, it may be bookmarked. One simply supplies a description of the data to be bookmarked. Bookmarks can be listed and loaded, as shown in the following figure.

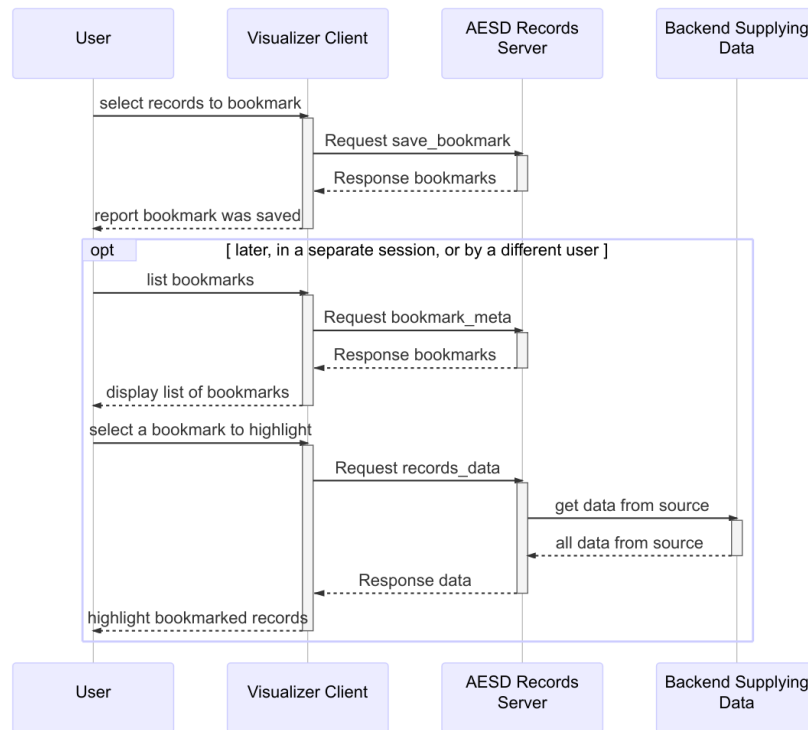


Figure 5: Creating and retrieving a bookmark and its associated data.

To create a bookmark for a specific list of records, simply supply their record identifiers as part of a **BookmarkMeta** message in the **save_bookmark** field of

Request:

```
version: 4
id: 4
save_bookmark {
  model_id: "example-model-1"
  new_bookmark {
    bookmark_name: "Sample Bookmark"
    set {
      record_ids: 10
      record_ids: 30
    }
  }
}
```

The response will be the same bookmark, but with the `bookmark_id` field added:

```
version: 4
id: 4
bookmarks {
  bookmark metas {
    bookmark_id: "bookmark-1"
    bookmark_name: "Sample Bookmark"
    set {
      record_ids: 10
      record_ids: 30
    }
  }
}
```

The user or another user can retrieve the records corresponding to the bookmark:

```
version: 4
id: 5
records_data {
  model_id: "example-model-1"
  bookmark_id: "bookmark-1"
}
```

This will return precisely the bookmarked records:

```
version: 4
id: 5
data {
  list {
    records {
      record_id: 10
      variables {
        var_id: 0
      }
    }
  }
}
```

```

        value: 10.5
    }
    variables {
        var_id: 1
        value: -5
    }
    variables {
        var_id: 2
        value: "first"
    }
}
records {
    record_id: 30
    variables {
        var_id: 0
        value: -15.7
    }
    variables {
        var_id: 1
        value: 30
    }
    variables {
        var_id: 2
        value: "third"
    }
}
}
}

```

3.5 Filtering

Filtering records can be used to select particular records for retrieval, via the `RequestRecordsData` message, or in defining bookmarks, via the `BookmarkMeta` message. The filtering of records is accomplished through expressions, `FilterExpression`, combining values for variables, `DomainMeta`, and the set operators not, union, and intersection, encoded in the messages `FilterNot`, `FilterUnion`, and `FilterIntersection`, respectively. For example, the expression $x \leq 20$ would be expressed as the following `FilterExpression`

```

filter_domain {
    interval {
        var_id: 0
        last_value: 20
    }
}

```

provided that x has `var_id = 0`. The expression $(10 \leq x \leq 20) \cup (y \notin \{4, 7\})$ would be expressed as

```
filter_union {
  filter_expressions {
    filter_domain {
      var_id: 0
      first_value: 10
      last_value: 20
    }
    filter_not {
      filter_expression {
        filter_domain {
          var_id: 1
          set {
            elements: 4
            elements: 7
          }
        }
      }
    }
  }
}
```

provided that x has `var_id = 0` and y has `var_id = 1`.

4 Records API, Version 4

The AESD Records API consists of Google Protobuf 3 (Google Developers 2017b) messages used for requesting and providing data and metadata for record-oriented information. This section contains the complete specification for version 4 of the Records API. Clients send **Request** messages and servers send **Response** messages, typically transported via WebSockets (Internet Engineering Task Force 2017).

4.1 Message Groups

The message types in the Records API are organized into thematic groups below.

4.1.1 Requests and Responses

Request messages are sent from client to server and **Response** messages are sent from server to client. Request messages contain a specific type of request and response messages contain a corresponding specific type of response.

- Request
- RequestModelsMeta
- RequestRecordsData
- RequestWork
- RequestBookmarkMeta
- RequestSaveBookmark
- RequestCancel
- Response

4.1.2 Metadata

Metadata messages describe data sources (“models”) and variables.

- ModelMeta
- ModelMetaList
- DomainMeta
- VarMeta
- VariableType
- VarSet
- VarInterval

4.1.3 Data Records

Data is represented as either lists of records or tables of them.

- Record
- VarValue
- Value
- RecordData
- RecordList
- RecordTable

4.1.4 Filtering

Records can be filtered by logical operations on conditions on values of variables in the records.

- FilterExpression
- FilterNot
- FilterIntersection
- FilterUnion
- DomainMeta

4.1.5 Bookmarks

Bookmarks record particular sets or records or conditions on record data.

- BookmarkMeta
- BookmarkMetaList
- BookmarkIntervalContent
- BookmarkSetContent

4.1.6 Miscellaneous

The following messages wrap data types for the content of records.

- DoubleList
- IntegerList
- StringList
- OptionalInt32
- OptionalUInt32
- OptionalString

4.2 General conventions

All fields are technically optional in ProtoBuf 3, but some fields may be required in each message type in order for the message to be semantically valid. In the following specifications for the messages, fields are annotated as *semantically required* or *semantically optional*. Also, the specification notes when field in the protobuf **oneof** construct are required or mutually exclusive.

Furthermore, one cannot determine whether an optional value has been set or not if it is just a value, as opposed to a message. That is not true for fields that are messages, where the absence of the field truly indicates that the value is absent, not just a default or unset value. The message **OptionalString**, for example, is used in this API to indicate whether a character string value is truly present. Thus **RequestModelsMeta** has a **model_id** field that indicates whether the request is for all models, when the field has not been set, or for a specific one, when the field has been set.

Throughout this specification, the following types are used for identifiers: * **var_id** is int32 * **model_id** is string * **record_id** is int64

This specification conforms to Protocol Buffers version 3.

4.3 Messages

4.3.1 BookmarkIntervalContent

A range of record identifiers can specify the content of a bookmark. Bookmark interval content provides a convenient means to bookmark a contiguous selection of records in a model.

Both fields in this message are optional:

- If neither field is present, then the bookmark interval designates all records in the model.
- If only `first_record` is present, then the bookmark interval designates all records starting from that record identifier.
- If only `last_record` is present, then the bookmark interval designates all records ending at that record identifier. For a dynamic model, such a bookmark interval includes all “future” records.
- If both fields are present, then the bookmark interval designates all records between the two identifiers, inclusive.

Field	Type	Label	Description
<code>first_record</code>	int64	optional	[semantically optional] The identifier for the first record in the interval.
<code>last_record</code>	int64	optional	[semantically optional] The identifier for the last record in the interval.

4.3.2 BookmarkMeta

A bookmark is metadata defining a subset of records in a model.

There are three alternatives to specifying a bookmark:

1. Interval content specifies a range of records in the bookmark.
2. Set content specifies a list of records in the bookmark.
3. A filter expression defines a set of logical conditions for determining whether a record is in the bookmark.

Exactly one of `interval`, `set`, or `filter` must be specified in this message.

Field	Type	Label	Description
<code>bookmark_id</code>	string	optional	[semantically optional] When creating a new bookmark, the identifier of the bookmark.
<code>bookmark_name</code>	string	optional	[semantically required] A name for the bookmark.
<code>interval</code>	BookmarkIntervalContent	optional	The range of records in the bookmark.
<code>set</code>	BookmarkSetContent	optional	The list of records in the bookmark.
<code>filter</code>	FilterExpression	optional	Logical conditions for defining which records are in the bookmark.

4.3.3 BookmarkMetaList

Bookmarks may be grouped into lists (sets).

Field	Type	Label	Description
bookmark metas	BookmarkMeta	repeated	[semantically optional] The bookmarks in the list.

4.3.4 BookmarkSetContent

A list (set) of record identifiers can specify the contents of a bookmark. Bookmark set content provides a convenient means to bookmark a specific selection of non-continuous records in a model.

Field	Type	Label	Description
record_ids	int64	repeated	[semantically optional] The list of record identifiers in the set.

4.3.5 DomainMeta

The domain (set of valid values) for a variable.

There are two alternatives to specifying a domain:

1. An interval specifies a range of values in the domain.
2. A set specifies a list of values in the domain.

*Exactly one of **interval** or **set** must be specified in the message.*

Field	Type	Label	Description
var_id	int32	optional	[semantically required]
interval	VarInterval	optional	The interval of values in the domain.
set	VarSet	optional	The list of values in the domain.

4.3.6 DoubleList

A list of real numbers.

Field	Type	Label	Description
values	double	repeated	[semantically required] The real numbers.

4.3.7 FilterExpression

A filtering expression is a composition of logical conditions on a record. It can be used to filter records. There are four alternatives to specifying a filter expression:

1. The logical negation of another filtering expression.
2. The set union of multiple filtering expressions.
3. The set intersection of multiple filtering expressions.
4. Particular values of variables in a record.

Exactly one of `filter_not`, `filter_union`, `filter_intersection`, or `filter_domain` must be specified in this message.

Field	Type	Label	Description
<code>filter_not</code>	FilterNot	optional	Logical negation of an expression.
<code>filter_union</code>	FilterUnion	optional	Set union of expressions.
<code>filter_intersection</code>	FilterIntersection	optional	Set intersection of expressions.
<code>filter_domain</code>	DomainMeta	optional	Particular values of variables.

4.3.8 FilterIntersection

Set intersection of filtering expressions. A record satisfies this expression if it satisfies all of `filter_expressions`.

Field	Type	Label	Description
<code>filter_expressions</code>	FilterExpression	repeated	[semantically required] The expressions to be intersected.

4.3.9 FilterNot

Logically negate a filtering expression. A record satisfies this expression if it does not satisfy `filter_expression`.

Field	Type	Label	Description
<code>filter_expression</code>	FilterExpression	optional	[semantically required] The expression to be negated.

4.3.10 FilterUnion

Set union of filtering expressions. A record satisfies this expression if it satisfies any of `filter_expressions`.

Field	Type	Label	Description
filter_expressions	FilterExpression	repeated	[semantically required] The expressions to be unioned.

4.3.11 IntegerList

A list of integers.

Field	Type	Label	Description
values	sint64	repeated	[semantically required] The integers.

4.3.12 ModelMeta

Metadata for a model.

Field	Type	Label	Description
model_id	string	optional	[semantically required] The unique identifier for the model <i>on the fly</i> .
model_name	string	optional	[semantically required] A name for the model, useful for display the model.
model_uri	string	optional	[semantically required] The unique URI for the model. Additional metadata for the model.
variables	VarMeta	repeated	[semantically required] Metadata for the variables.
inputs	DomainMeta	repeated	[semantically optional] Metadata for input values to the model, if a model has inputs.

4.3.13 ModelMetaList

A list of metadata for models.

Field	Type	Label	Description
models	ModelMeta	repeated	[semantically optional] The metadata for the models.

4.3.14 OptionalInt32

Wrapper for an optional signed integer.

Field	Type	Label	Description
value	int32	optional	[semantically required] The signed integer value.

4.3.15 OptionalString

Wrapper for an optional string.

Field	Type	Label	Description
value	string	optional	[semantically required] The character string value.

4.3.16 OptionalUInt32

Wrapper for an optional unsigned integer.

Field	Type	Label	Description
value	uint32	optional	[semantically required] The unsigned integer value.

4.3.17 Record

A record is a list of variables and their associated values.

Field	Type	Label	Description
record_id	int64	optional	[semantically required] A unique identifier for the record.
variables	VarValue	repeated	[semantically optional] The values for variables in the record.

4.3.18 RecordData

A collection of records.

There are two alternatives to specifying record data:

1. A list specifies a heterogeneously typed list.
2. A table specifies a homogeneously typed table.

*Exactly one of **list** or **table** must be present in the message.*

Field	Type	Label	Description
list	RecordList	optional	A heterogeneously typed list of records.
table	RecordTable	optional	A homogeneously typed table of records.

4.3.19 RecordList

A list of records. The list is heterogeneous in the sense that each variable may have a different type.

Field	Type	Label	Description
records	Record	repeated	[semantically optional] The list of records.

4.3.20 RecordTable

A homogeneously typed table of records, where each variable has each type, with a row for each record and a column for each variable.

This message represents the following table:

Record Identifier	var_id[0]	var_id[1]	. . .	var_id[N]
rec_id[0]	list[0][0]	list[0][1]	. . .	list[0][N]
rec_id[1]	list[1][0]	list[1][1]	. . .	list[1][N]
.
rec_id[M]	list[M][0]	list[M][1]	. . .	list[M][N]

The underlying list is a **single** array, addressable using the following row-major index formula $\text{list}[\text{row}][\text{var}] = \text{array}[\text{var} + \text{NY} * \text{row}]$ where $\text{NX} = \text{length of } \text{rec_ids}$ and $\text{NY} = \text{length of } \text{var_ids}$.

*Exactly one of **reals**, **integers**, or **strings** must be specified in the message.*

Field	Type	Label	Description
var_ids	int32	repeated	[semantically required] The identifiers of the variables (columns) in the table.
rec_ids	int64	repeated	[semantically required] The identifiers of the records (rows) in the table.
reals	DoubleList	optional	The real numbers comprising the values of the variables, in row-major order.
integers	IntegerList	optional	The integers comprising the values of the variables, in row-major order.
strings	StringList	optional	The character strings comprising the values of the variables, in row-major order.

4.3.21 Request

A request. There are six types of requests:

Request	Response
Metadata for model(s)	ModelMetaList
Data records	RecordData

Request	Response
Metadata for bookmark(s)	BookmarkMetaList
Saving a bookmark	BookmarkMetaList
Canceling a previous request	n/a
New work, such as a simulation	RecordData

*Exactly one of `models_metadata`, `records_data`, `bookmark_meta`, `save_bookmark`, `cancel`, or `work` must be specified in the message.

Field	Type	Label	Description
version	uint32	optional	[semantically required] The version number for the model.
id	OptionalUInt32	optional	[semantically optional, but recommended] An identifier for the model.
subscribe	bool	optional	[semantically optional] Whether to continue receiving updates.
models_metadata	RequestModelsMeta	optional	Request metadata for model(s).
records_data	RequestRecordsData	optional	Request data records.
bookmark_meta	RequestBookmarkMeta	optional	Request metadata for bookmark(s).
save_bookmark	RequestSaveBookmark	optional	Request save a new bookmark or update an existing one.
cancel	RequestCancel	optional	Request cancel a previous request).
work	RequestWork	optional	Request request work (e.g., simulation results).

4.3.22 RequestBookmarkMeta

A request for one or more bookmarks for a model.

The response to this request is `BookmarkMetaList`.

Field	Type	Label	Description
model_id	string	optional	[semantically required] Which model for which to list bookmarks.
bookmark_id	OptionalString	optional	[semantically optional] If empty, list all bookmarks for the model.

4.3.23 RequestCancel

Cancel a previous request.

Field	Type	Label	Description
id	OptionalUInt32	optional	[semantically required] Which request to cancel.

4.3.24 RequestModelsMeta

A request for metadata about model(s).

The response to this request is ModelMetaList.

Field	Type	Label	Description
model_id	OptionalString	optional	[semantically optional] If absent, the request is for metadata for all models.

4.3.25 RequestRecordsData

Request record data for a model.

There are three alternatives to requesting record data.

1. Request all records.
2. Request records in a bookmark.
3. Filter records according to a criterion.

The response to this request is RecordData.

No more than one of `bookmark_id` or `expression` may be present in the message.

Field	Type	Label	Description
model_id	string	optional	[semantically required] The identifier for the model.
max_records	uint64	optional	[semantically optional] If specified, this is the maximum number of records to return.
var_ids	int32	repeated	[semantically optional] Which variables to include in the response.
bookmark_id	string	optional	[semantically optional] Only respond with records in a specified bookmark.
expression	FilterExpression	optional	[semantically optional] Only respond with records matching a specified expression.

4.3.26 RequestSaveBookmark

A request to create or update a bookmark.

The response to this request is BookmarkMetaList.

Field	Type	Label	Description
model_id	string	optional	[semantically required] Which model for which to save the bookmark.
new_bookmark	BookmarkMeta	optional	[semantically optional] If empty, create a new bookmark. (In v1.0, this field was required.)

4.3.27 RequestWork

Request that the server compute new records based on input values.

The response to this request is RecordData.

Field	Type	Label	Description
model_id	string	optional	[semantically required] The identifier for the model.
inputs	VarValue	repeated	[semantically optional] Which input variables to set to which values.

4.3.28 Response

A response to a request.

Note that a server may send multiple responses to a single request, expressed as a linked list of chunks. It is strongly recommended that servers chunk by **record_id** so that each record is kept intact. A chunk may be empty.

Field	Type	Label	Description
version	uint32	optional	[semantically required] The version number for the API. <i>TODO</i>
id	OptionalUInt32	optional	[semantically optional] A response without an identifier is
chunk_id	int32	optional	[semantically optional, but recommended] The identifier for
next_chunk_id	int32	optional	[semantically optional] The identifier of the next chunk, or
error	string	optional	An error message.
models	ModelMetaList	optional	A list of model metadata.
data	RecordData	optional	A list of record data.
bookmarks	BookmarkMetaList	optional	A list of bookmark metadata.

4.3.29 StringList

A list of character strings.

Field	Type	Label	Description
values	string	repeated	[semantically required] The character strings.

4.3.30 Value

Value that may be a real number, an integer, or a character string

*Exactly one of **real_value**, **integer_value**, or **string_value** must be specified in this message.*

Field	Type	Label	Description
real_value	double	optional	The real number.
integer_value	int64	optional	The integer.

Field	Type	Label	Description
string_value	string	optional	The character string.

4.3.31 VarInterval

A range of values of a variable.

Both fields in this message are optional:

- If neither field is present, then the interval designates all values in the domain.
- If only `first_value` is present, then the interval designates all values starting from that value.
- If only `last_value` is present, then the bookmark interval designates all values ending at that value.
- If both fields are present, then the interval designates all values between the two values, inclusive.

Field	Type	Label	Description
first_value	Value	optional	[semantically optional] The first value in the interval.
last_value	Value	optional	[semantically optional] The last value in the interval.

4.3.32 VarMeta

Metadata for a variable.

Field	Type	Label	Description
var_id	int32	optional	[semantically required] A integer identifying the variable.
var_name	string	optional	[semantically required] The name of the variable.
units	string	optional	[semantically optional] The name of the unit of measure for values of
si	sint32	repeated	[semantically optional] The unit of measure expressed as a list of the
scale	double	optional	[semantically optional] An overall scale relative to the fundamental S
type	VariableType	optional	[semantically optional] The data type for values of the variable. The

4.3.33 VarSet

A set of values for a variable.

Field	Type	Label	Description
elements	Value	repeated	[semantically optional] The list of values in the set.

4.3.34 VarValue

The value of a variable.

Field	Type	Label	Description
var_id	int32	optional	[semantically required] The identifier for the variable.
value	Value	optional	[semantically required] The value of the variable.

4.3.35 VariableType

The data type for a value.

Name	Number	Description
REAL	0	A real number.
INTEGER	1	An integer.
STRING	2	A character string.

4.4 Scalar Value Types

.proto Type	Notes
double	
float	
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely
uint32	Uses variable-length encoding.
uint64	Uses variable-length encoding.
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28} .
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56} .
sfixed32	Always four bytes.
sfixed64	Always eight bytes.
bool	
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.
bytes	May contain any arbitrary sequence of bytes.

5 Appendices

5.1 Protocol Buffers for Records API Version 4

```
syntax = "proto3";
package AesdRecords;

option optimize_for = LITE_RUNTIME;

message OptionalInt32 {
    int32 value = 1; /// [semantically required]
}

message OptionalUInt32 {
    uint32 value = 1; /// [semantically required]
}

message OptionalString {
    string value = 1; /// [semantically required]
}

message Value {
    oneof value /// [semantically required]
    {
        double real_value = 1;
        int64 integer_value = 2;
        string string_value = 3;
    }
}

message DoubleList {
    repeated double values = 1; /// [semantically required]
}

message IntegerList {
    repeated sint64 values = 1; /// [semantically required]
}

message StringList {
    repeated string values = 1; /// [semantically required]
}

message BookmarkIntervalContent {
    int64 first_record = 1; /// [semantically optional]
    int64 last_record = 2; /// [semantically optional]
```

```

}

message BookmarkSetContent {
    repeated int64 record_ids = 1; /// [semantically optional]
}

message BookmarkMeta {
    string bookmark_id = 1; /// [semantically optional]
    string bookmark_name = 2; /// [semantically required]
    oneof content /// [semantically required]
    {
        BookmarkIntervalContent interval = 3;
        BookmarkSetContent set = 4;
        FilterExpression filter = 5;
    }
}

message BookmarkMetaList {
    repeated BookmarkMeta bookmark_metas = 1; /// [semantically optional]
}

message RequestBookmarkMeta {
    string model_id = 1; /// [semantically required]
    OptionalString bookmark_id = 2; /// [semantically optional]
}

message RequestSaveBookmark {
    string model_id = 1; /// [semantically required]
    BookmarkMeta new_bookmark = 2; /// [semantically optional]
}

message FilterExpression {
    oneof expression /// [semantically required]
    {
        FilterNot filter_not = 1;
        FilterUnion filter_union = 2;
        FilterIntersection filter_intersection = 3;
        DomainMeta filter_domain = 4;
    }
}

message FilterNot {
    FilterExpression filter_expression = 1; /// [semantically required]
}

message FilterUnion {

```

```

    repeated FilterExpression filter_expressions = 1; /// [semantically required]
}

message FilterIntersection {
    repeated FilterExpression filter_expressions = 1; /// [semantically required]
}

enum VariableType
{
    REAL          = 0;
    INTEGER       = 1;
    STRING        = 2;
}

message VarMeta {
    int32          var_id   = 1; /// [semantically required]
    string         var_name = 2; /// [semantically required]
    string         units    = 3; /// [semantically optional]
    repeated sint32 si      = 4; /// [semantically optional]
    double         scale    = 5; /// [semantically optional]
    VariableType   type     = 6; /// [semantically optional]
}

message ModelMeta {
    string         model_id   = 1; /// [semantically required]
    string         model_name = 2; /// [semantically required]
    string         model_uri  = 3; /// [semantically required]
    repeated VarMeta variables = 4; /// [semantically required]
    repeated DomainMeta inputs = 5; /// [semantically optional]
}

message ModelMetaList {
    repeated ModelMeta models = 1; /// [semantically optional]
}

message RequestModelsMeta {
    OptionalString model_id = 1; /// [semantically optional]
}

message VarInterval {
    Value first_value = 1; /// [semantically optional]
    Value last_value  = 2; /// [semantically optional]
}

message VarSet {
    repeated Value elements = 1; /// [semantically optional]
}

```

```

}

message DomainMeta {
    int32      var_id    = 1; /// [semantically required]
    oneof      domain    /// [semantically required]
    {
        VarInterval interval = 2;
        VarSet      set      = 3;
    }
}

message RequestWork {
    string model_id      = 1; /// [semantically required]
    repeated VarValue inputs = 2; /// [semantically optional]
}

message VarValue {
    int32 var_id = 1; /// [semantically required]
    Value value  = 2; /// [semantically required]
}

message Record {
    int64 record_id      = 1; /// [semantically required]
    repeated VarValue variables = 2; /// [semantically optional]
}

message RecordList {
    repeated Record records = 1; /// [semantically optional]
}

message RecordTable {
    repeated int32 var_ids    = 1; /// [semantically required]
    repeated int64 rec_ids    = 2; /// [semantically required]
    oneof          list      /// [semantically required]
    {
        DoubleList reals      = 3;
        IntegerList integers = 4;
        StringList strings    = 5;
    }
}

message RecordData {
    oneof          style      /// [semantically required]
    {
        RecordList list  = 1;
        RecordTable table = 2;
    }
}

```

```

    }
}

message RequestRecordsData {
    string          model_id    = 1; /// [semantically required]
    uint64          max_records = 2; /// [semantically optional]
    repeated int32  var_ids     = 3; /// [semantically optional]
    oneof           filter      /// [semantically optional]
    {
        string          bookmark_id = 4; /// [semantically optional]
        FilterExpression expression = 5; /// [semantically optional]
    }
}

message Response {
    uint32          version      = 1; /// [semantically required]
    OptionalUInt32  id           = 2; /// [semantically optional]
    int32           chunk_id     = 3; /// [semantically optional, but recommended]
    int32           next_chunk_id = 4; /// [semantically optional]
    oneof           type         /// [semantically optional]
    {
        string          error      = 5;
        ModelMetaList   models     = 6;
        RecordData      data       = 7;
        BookmarkMetaList bookmarks = 8;
    }
}

message RequestCancel {
    OptionalUInt32 id = 1; /// [semantically required]
}

message Request {
    uint32          version      = 1; /// [semantically required]
    OptionalUInt32  id           = 2; /// [semantically optional, but recommended]
    bool            subscribe    = 3; /// [semantically optional]
    oneof           type         /// [semantically required]
    {
        RequestModelsMeta models_metadata = 4;
        RequestRecordsData records_data   = 5;
        RequestBookmarkMeta bookmark_meta = 6;
        RequestSaveBookmark save_bookmark = 7;
        RequestCancel      cancel         = 8;
        RequestWork        work           = 9;
    }
}

```


6 References

Fowler, Martin. 2017. “UML Distilled.” Accessed April 11. <http://my.safaribooksonline.com/book/software-engineering-and-development/uml/0321193687/sequence-diagrams/ch04>.

Google Developers. 2017a. “Protocol Buffers - Google’s Data Interchange Format.” Accessed April 11. <https://github.com/google/protobuf/blob/master/README.md>.

———. 2017b. “Protocol Buffers | Google Developers.” Accessed April 11. <https://developers.google.com/protocol-buffers/>.

Internet Engineering Task Force. 2017. “RFC 6455 - the WebSocket Protocol.” Accessed April 11. <https://tools.ietf.org/html/rfc6455>.