Trio Mind Client Developer Guide for Control APIs

Jun. 18, 2014

Contents

1 About This Guide

This guide explains how to develop clients, typically user applications, that can interact with the Trio Mind, enabling a wide variety of operations on digital media content (including TV shows, movies, music, and images).

1.1 Background knowledge

General knowledge of client-server architecture will help you understand the material in this guide. Knowledge of basic object-oriented programming concepts is also assumed.

The client sends requests, and receives replies back, in the form of *dictionaries* specifying objects (and other structures) and operations with name-value pairs as defined by the Trio *schema*; the use of the Mind and those dictionaries is largely what this guide describes.

1.2 Conventions used in this guide

Where this guide shows syntax or examples, the following conventions apply:

- *Italics* denote a place-holder in the syntax or example. For instance, *object*Search refers to an operation whose name begins with an object type (denoted by the italic *object*) and ends with the word Search.
- Square brackets enclose an optional element in syntax. (The square brackets are not part of the syntax.) For example, `tivo:`*`type`*`[.namespace].id` indicates that a period followed by a namespace may optionally follow the type specifier (represented by the place-holder *`type`*), as in `tivo:ct.ts.000701`.

  Note: This convention does not apply in code examples; square brackets in examples must appear where shown (such as to encode an array in JSON).

- **Boldface** is used in some examples to emphasize the part of the example that's most relevant to the surrounding discussion.
- A backslash (\) is used as a line-continuation character to make long lines (such as curl commands) fit in a window or on a printed page. Here's an example:

```
http://pdk01.st:8085/mind/mind11 \
?type=customerInfoSearch \
&apikey=bmu5fyk68hpbdmprj6rdv94y \
&sig=df83a84de19d7c36dab1cf4578749225 \
&partnerCustomerId=0900000003
```

2 Trio concepts

Trio is a client-server architecture in which the Mind is the server providing TiVo services to clients. Clients â ” typically user applications â ” can use the Trio Mind to access TiVo services that enable a wide variety of operations on digital media content (including TV shows, movies, music, and images). For example, an application running on a consumer device such as a TiVo DVR or a personal computer could access TiVo services to search for, schedule, and record or download content.

2.1 The Trio architecture

The Trio architecture is based on three kinds of components: Mind, Body, and Face.

- The Trio *Mind* is a set of services that provide access to information, such as program guide data, and also enable control of certain consumer devices, such as TiVo DVRs (for example, to schedule a recording on a DVR).
- A *Body* is a consumer device whose behavior can be controlled by the Mind (at least partially). There are currently two types of Bodies: TiVo DVRs (whether in a TiVo TV or otherwise), and devices running software that makes them act like TiVo DVRs.
- A *Face* is an application that uses the Mind, typically providing an interactive user interface to a consumer via a TV screen, a web browser on a PC, or a mobile device. It can also be an application running on a web server, displaying its results in HTML. For example, another service can use the Mind to publish reviews of movies it offers. Faces use the services provided by the Mind to get data and to provide instructions to the Mind, and to Bodies via the Mind, based on the user's actions.

2.1.1 The Mind

The Trio Mind is responsible for:

- Managing user accounts and Bodies.
- Searching for content.
- Scheduling content downloads.
- Scheduling recordings.
- Exchanging information with Faces and Bodies.

The Mind also maintains databases that contain:

- Account data: user preferences, personal information, content bookmarks (place-holders in videos marking where to resume viewing later), access rights, and so on for each user account.
- Body-related data: user preferences, disk space budgets, subscriptions, and recordings for each Body.
- Content data: generic data about content, not specific to a particular Body â ” for example, current program guide data (including the schedules for broadcast TV).

2.1.2 The Body

The Body is responsible for:

- Exchanging information with the Mind about what to record from broadcast TV, what to download from broadband content sources, and the current status of recordings and other content stored on the device.
- Recording broadcast TV shows.
- Downloading broadband content.
- Managing storage space on the device.
- Playing back recorded programs.
- Playing streamed broadband content.
- Providing a platform that hosts the Face.

When a Body is running and connected to the network, it maintains a persistent connection to the Mind. This connection allows messages to be sent from the Mind to the Body without having to route packets through NATs and firewalls in the home. It also allows messages to be sent in both directions without the overhead of SSL negotiation. Client applications need to access the Body to interface with TiVo middleware. This access enables activities such as scheduling recordings, managing recordings, and navigating in the UI.

2.1.3 The Face

The Face is typically an application running on a consumer device; for example, one that runs inside a web browser and calls the Mind using JavaScript (more specifically, using the set of interrelated technologies known as Ajax, short for Asynchronous JavaScript and XML). The Face is responsible for the following (where mention of the Body applies only if the consumer device is a Body, as opposed to a device that cannot be controlled by the Mind):

- Presenting the user interface that reflects the state of the Mind and the Body
- Responding to the user's actions
- Giving instructions to the Mind, and to the Body via the Mind, based on the user's actions

2.2 Requirements for interacting with the Mind

To interact with the Trio Mind, you need, at a minimum, the following:

- Software that can send a Mind RPC request and receive the response.
- For authentication, an SSL certificate.

2.3 A simple example of Trio in action

This section describes how a web application might use the Mind to enable a user to find out when an episode of the TV series *Glee* is next showing on DIRECTV. Many details are omitted, not only about the Mind but also about web page design, implementation, and server configuration (including authentication). However, this example will help you understand how the components of the Trio architecture work together and also introduce some basic Trio terminology and concepts.

The scenario begins with a simple web page where the user enters the exact title of a TV series. When the user submits the page to the web server (by clicking Submit), the web application queries the Mind and uses the response to create a web page that lists the next showing of the series. Both the query made to the Mind and the response that the Mind returns take the form of a *dictionary* consisting of name-value pairs that specify *fields.*

In this example, the web application queries the Mind with the following request:

```
{
   "type": "offerSearch",
   "lineup": [
     {
        "lineupType": 84,
        "headend": "DITV501"
      }
   ],
   "title": "Glee",
   "orderBy": ["startTime"]
}
```

Note: Except where the context calls for showing a different encoding, examples in this guide generally show dictionaries in the JSON encoding.

In the terminology of the Mind:

- A TV series is represented by a *collection,* and each episode of the series is *content* within the collection. For example, the collection with the title `Glee` consists of content items that are episodes of that series (each with a subtitle identifying the particular episode).
- An *offer* is a specification of how to obtain content â ” in this case to record it from broadcast TV. In this example, the user will learn that the next showing of *Glee* is the episode "Laryngitis" on Wednesday, September 8, at 8 p.m. on channel 5.
- The preceding request dictionary specifies an *operation* to find an offer (the `offerSearch` operation, as indicated by the `type` field). Dictionaries also specify structures, the most basic of which are *objects*; `collection`, `content`, and `offer` are all object types.

The value of the `lineup` field in the request is a dictionary that specifies a `lineup`object (previously obtained by a `lineupSearch` request) corresponding to the DIRECTV lineup for the East Coast. The `orderBy` field in the request asks that the search results be sorted from earliest to latest start time. Only the first result will be returned in this case (because optional fields for specifying otherwise are not included in the request).

When the Mind receives this request, it searches its database of current program guide data, looking for the offer corresponding to the next showing of *Glee*. Its response might include an offer like the following (actually, a list of offers that consists of just one item in this case). Note that the fields in the response are in alphabetical order (placing the `type` field last in this example and many others); the Mind always orders responses this way, although it accepts names in any order in a JSON-encoded request.

```
{
   "isBottom": false,
   "isTop": true,
   "offer": [
     {
        "audioLanguage": ["en"],
        "channel": {
           "affiliate": "FOX affiliate",
           "callSign": "WNYWDT",
           "channelNumber": "5",
           "levelOfDetail": "low",
           "name": "WNYWDT (WNYW-DT)",
           "sourceType": "satellite",
           "stationId":
           "tivo:st.5587434",
           "type": "channel"
        },
        "collectionId": "tivo:cl.144781663",
        "collectionType": "series",
        "contentId": "tivo:ct.164212036",
        "contentType": "video",
        "duration": 3600,
        "episodeNum": [18],
        "episodic": true,
        "hdtv": false,
        "isAdult": false,
        "isEpisode": true,
        "levelOfDetail": "low",
        "noGiftCardPurchase": false,
        "offerId":
        "tivo:of.ctd.362.517.satellite.2010-08-29-20-00-00.3600",
        "repeat": true,
        "seasonNumber": 1,
        "startTime": "2010-09-08 20:00:00",
        "subtitle": "Laryngitis",
        "title": "Glee",
        "transportType": "stream",
        "type": "offer"
     }
   ],
   "type": "offerList"
}
```

Drawing on the information provided in the response dictionary, the web server then creates a web page displaying the result of the search, as follows:

```
Next showing of "Glee":
Episode 18: "Laryngitis"
Wednesday, September 8, 2010, 8 p.m., channel 5, WNYWDT (WNYW-DT)
(60 minutes)
```

3 Trio built-in data types

The following table presents the names and definitions of the atomic data types used by the Trio Schema. The definitions consist of regular expression patterns that each type is validated against.

Type name	Pattern
int	"((-?[0-9]+)|(0x[0-9A-Fa-f]+))"
boolean	"(true|false)"
bytestring	"[0-9a-zA-Z+/=]*"
float	"-?(([0-9]+)|(([0-9]+)[.]([0-9]*)?)|([.][0-9]+))([eE][+-]?[0-9]+)?"
date	"[0-9]{4}-[0-9]{2}-[0-9]{2}"
time	"([01][0-9]|2[0123]):[0-5][0-9]:[0-5][0-9]"
dateTime	"[0-9]{4}-[0-9]{2}-[0-9]{2}" + " " + "([01][0-9]|2[0123]):[0-5][0-9]:[0-5][0-9]"
channelNumber	"[0-9]+(-[0-9]+)?"
currency	"\\d*([.]\\d+)?"
idBase64	"(tivo:[a-z][a-z][.][-._a-zA-Z0-9]{1,33})|" + "(tsn:[0-9a-zA-Z]{1,15})"
idLongAndType	"tivo:[a-z]+[.][a-z]*[.]?{1,132}" **Note:** This represents any objectId referenced in the schema as `id:`*`objectType`*
idString	"tivo:[a-z][a-z][.].{1,132}"
localId	"[bm]-[a-zA-Z0-9\\-]{1,18}"
objectId28	"(tivo|tms|tvguide):[a-z][a-z][.][a-zA-Z0-9\\-.]{1,20}" + "|tsn:[0-9a-zA-Z]{1,15}"
objectId140	"(tivo|tms):[a-z][a-z][.].{1,132}" + "|tsn:[0-9a-zA-Z]{1,15}"
string	Any UTF-8 character(s).

4 Trio objects

Objects are the basic data structures involved in communicating with the Mind. They're typically referred to (by ID) in dictionaries requesting an operation or are included along with other structures in dictionaries you receive in response to requests.

The following table describes the most important Trio objects. Some object types extend other types, in that they have all the fields of another type (mostly with the same values as in the other type, though there may be some differences) plus some fields of their own. This extending does not create an is-a relationship; for instance, even though the `content` object type extends the `collection`type, a content object is not a collection. The relationships are set up this way to help Mind clients use the objects more interchangeably.

Object	Description
Account object	Data related to a TiVo user account. Each account has an associated profile indicating the personal data, preferences, content bookmarks, access rights, and so on for the account.
Partner object	Contact information and other data related to a TiVo partner.
Body object	Data related to the device on which a user is accessing TiVo services.
Collection object	A grouping for related digital media content, such as a TV series (all episodes or selected ones, such as those that form a given story arc), a music album, or a photo stream. Every content object is part of a collection, even if the collection has only one thing in it; for example, every movie has a collection object as well as a content object. Note: Currently, there is a separate collection for each language in which a TV series or a movie is available. The collection object doesnâ ™t include the content itself but rather is associated with each of the related content objects (through a collection ID field in the content object).
Content object	Data related to a piece of digital media such as a TV show (one episode, if part of a TV series), a movie, a song, or an image. This data may include, for example, a title, description, category (genre), rating, parental advisory, and credits (actors, director, and so on). A content object includes all the fields of a collection object. Note: The term *content* is used loosely in this guide, often referring to the digital media itself (and not just the data about it that's contained in the content object). The meaning should be clear from the context, but the term *content object* is used where important for clarity.
Offer object	A specification of how to obtain content, whether from broadcast TV, a cable systemâ ™s video on demand (VOD) service, a TiVo content delivery service (CDS), download sites like BitTorrent, or similar content providers. In addition to the channel and time or the URL for obtaining the content, an offer includes related information such as the content type, provider, cost, and access rights. For example, an offer for the content (episode) "Friend or Foe?" in the collection (TV series) Crossing Jordan might be: channel 64; July 23, 2006, 7 p.m.; on-air broadcast; NBC; free. Each content object has zero or more offers associated with it. An offer object includes all the fields of a content object.
Recording object	The object that contains (or will contain) actual bits of media that reside on a device as specified by an offer object. Recording objects can refer to offers that either have already been captured on disk (recorded or downloaded) or are scheduled for capture. A recording object includes all the fields of an offer object.

Category object	A classification of content, such as "Movies" (at the top level) or "Comedy" (at a lower level), that's part of a read-only hierarchy maintained by the Mind. The client can discover the category hierarchy through searching and then limit a content search to one or more specific categories.
Subscription object	An object representing that a given Body should get (record or download) certain content that the user is interested in. For example, a user might want to subscribe to all episodes of the TV series *The Simpsons*.
Mix object	A grouping of collections, content, and offers into a tree structure to present to the user. Users can subscribe to all or part of the tree. Mixes are the objects behind user-visible features such as celebrity playlists and a TiVo DVR's Guru Guide recommendations.

4.1 About objectIDs

Most types of object have a single object ID, which is a string stored in the `*object*Id` field (where `*object*` is the object type). This ID is assigned by the Mind and has the format described in the next section. Objects refer to each other by object ID, and object IDs can also be specified in other structures or in operations. IDs returned by one query can be used in subsequent queries.

For some object types, an additional ID may be assigned by a TiVo partner, for the partner's use only. For example, a collection may have a `partnerCollectionId` as well as a `collectionId`.

Within each object type, every `objectId` value is unique, though the context for uniqueness varies as follows:

- Some objects have IDs that are unique globally within the Mind. Examples of global IDs are `bodyId`, `contentId`, and `collectionId`.
- Other objects have IDs that are unique only within one Body. Examples of these *Body-relative*object IDs include `subscriptionId`, `recordingId`, and `channelId`. Body-relative IDs are always used in a context in which their Body is clear (and objects that have Body-relative IDs always include a `bodyId` field).

Partner-assigned IDs must be unique for that partner (that is, different from all other IDs assigned by the partner) within each object type.

Some object types do not have their own `objectId` field but instead are identified by a (unique) combination of other IDs. For example, the `partnerAccountInfo` object is identified by a `bodyId` and a `partnerId` (and also has a `partnerBodyId`, assigned by the partner).

For many objects, including collections and content representing TV series or movies, their object IDs always stay the same. Other objects are more transitory. Offers are the most common example of transitory objects. Offer objects for broadcast TV change daily as a new program guide becomes available and old offers slip into the past and become irrelevant.

You can compare two objects to see if they're the same by comparing their object IDs as strings. However, note that even if two objects have the same ID, the dictionaries that define them may include different sets of fields or have different values in some fields.

You should treat object IDs that are assigned by the Mind as opaque strings. Comparing them for anything other than an exact match will yield unpredictable results, and you should not write code that parses them.

4.2 objectID format

Each object ID assigned by the Mind is a string that begins with `tivo:` and has the following form:

```
tivo:type[.namespace].id
```

where:

- `type` is a two-letter type specifier â ” for example, `cl` for a collection object or`ct` for a content object. Most ID types are limited to 28 characters, but offer IDs may contain 100 characters.
- `namespace` is optional (as indicated by the square brackets, which are not part of the syntax). Some Mind queries, such as `contentSearch`, accept a namespace as a search parameter.
- `id` is the type-specific part of the object ID. Its format depends on the object type but always consists of only letters (`A`-`Z` and `a`-`z`), digits (`0`-`9`), periods (`.`), and hyphens (`-`).

Partner-assigned object IDs may consist of any printable ASCII characters (up to 100 characters total).

The following table lists type specifiers and examples of object IDs assigned by the Mind for some commonly used object types.

Object type	Type specifier	Example object ID
account	ta	tivo:ta.1OndFHQOmX5Q9LZFlqspx6ZRqhJqg48Cp Note: The account ID is an exception to the other object IDs in that, for security purposes, itâ ™s an encryption of an internal-only account ID. Furthermore, the schema and some older Trio documentation may, until updated, refer to the external account ID as an "account token" and use the name accountToken rather than accountId. This guide uses only "account ID" and accountId.
body	bd	tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-
candidate	cd	tivo:cd.23X532
category	ca	tivo:ca.413902
channel	ch	tivo:ch.8660782
collection	cl	tivo:cl.17787
content	ct	tivo:ct.ts.7368172
mind	md	The identifier for the mind itself. There may be multiple minds, such as "staging" and "production".
offer	of	tivo:of.ctd.727.2005-10-03-22-00-00.1800
partner	pt	tivo:pt.2147
recording	rc	tivo:rc.195349
schedulerRun	sr	
station	st	tivo:st.13274992
subscription	sb	tivo:sb.12
TiVo Service Number	tsn	tsn:7F60001901E2DF1 A TiVo Service Number (TSN) must contain exactly 15 hexadecimal characters [A-F, 0-9]. Letters must be upper-case.
tuner	tn	

5 Trio naming patterns

For many object types in the Mind, there are related operations and structures that follow predictable patterns in their names and semantics. For example, for the `collection` object type there's a `collectionStore` operation that creates or modifies collection objects, a `collectionSearch` operation that looks for existing collection objects, and a `collectionList` structure that contains a list of collection objects. Likewise, for the `content` object type there are `contentStore` and `contentSearch` operations and a `contentList` structure. The naming patterns are shown in the following table (where *object* is a place-holder for the object type). Not all object types support every one of these patterns, and for some types the usage might be restricted.

Type	Description
object	Each object type is represented by an *object* structure.
*object*Store	This operation creates or modifies an object of type *object* and stores the result in the Mind.
*object*Remove	This operation deletes an object of type *object*.
*object*Search	This operation searches the universe of objects for objects of type *object* (optionally filtering, ordering, and grouping them) and returns a subset of the results, called a *window*. When grouping is not specified, *object*Search returns an *object*List structure; otherwise, it returns an *object*GroupList structure.
*object*List	Object searches that do not organize their results into groups organize them into *object*List structures. This structure contains an ordered list of objects of type *object*, stored as multiple values of a field named *object*. For instance, an `offerList` structure contains 0 or more values for the field named `offer`.
*object*Group	Object searches that organize their results into groups organize them into *object*Group structures, each containing a count of the objects in the group and an example object from the group. For instance, an offer group contains an example offer that's in the group.
*object*GroupList	Object searches that organize their results into groups return *object*GroupList structures. An *object*GroupList structure contains an ordered list of *object*Group structures, stored as the multiple values of the field named *object*Group.

6 Trio built-in data types

The following table presents the names and definitions of the atomic data types used by the Trio Schema. The definitions consist of regular expression patterns that each type is validated against.

Type name	Pattern
int	"((-?[0-9]+)|(0x[0-9A-Fa-f]+))"
boolean	"(true|false)"
bytestring	"[0-9a-zA-Z+/=]*"
float	"-?(([0-9]+)|(([0-9]+)[.]([0-9]*)?)|([.][0-9]+))([eE][+-]?[0-9]+)?"
date	"[0-9]{4}-[0-9]{2}-[0-9]{2}"
time	"([01][0-9]|2[0123]):[0-5][0-9]:[0-5][0-9]"
dateTime	"[0-9]{4}-[0-9]{2}-[0-9]{2}" + " " + "([01][0-9]|2[0123]):[0-5][0-9]:[0-5][0-9]"
channelNumber	"[0-9]+(-[0-9]+)?"
currency	"\\d*([.]\\d+)?"
idBase64	"(tivo:[a-z][a-z][.][-._a-zA-Z0-9]{1,33})|" + "(tsn:[0-9a-zA-Z]{1,15})"
idLongAndType	"tivo:[a-z]+[.][a-z]*[.]?{1,132}" **Note:** This represents any objectId referenced in the schema as `id:`*`objectType`*
idString	"tivo:[a-z][a-z][.].{1,132}"
localId	"[bm]-[a-zA-Z0-9\\-]{1,18}"
objectId28	"(tivo|tms|tvguide):[a-z][a-z][.][a-zA-Z0-9\\-.]{1,20}" + "|tsn:[0-9a-zA-Z]{1,15}"
objectId140	"(tivo|tms):[a-z][a-z][.].{1,132}" + "|tsn:[0-9a-zA-Z]{1,15}"
string	Any UTF-8 character(s).

7 Trio dictionaries

A client communicates with the Mind via the TiVo-defined Mind RPC protocol. Remote procedure calls are made by the client to send a request to the Mind, and by the Mind to send back a response. Both the request and response are in the form of a *dictionary.*

Dictionaries consist of name-value pairs corresponding to *fields* that specify structures (including basic data structures called *objects*) and operations used in communicating with the Mind, as defined by the Trio *schema*. The value of a dictionary's `type` field indicates the type of structure or operation that the dictionary represents.

You make a request of the Mind by sending a dictionary that specifies an operation. Most operations take arguments, in which case the request dictionary will typically refer to one or more objects (and possibly other structures). The Mind usually responds with a dictionary that contains one or more objects. Some operations don't need to return any data; upon successful completion, they return an empty `success` dictionary. Usually an operation that fails will return an `error` dictionary, although in rare cases a 5*xx* (server error) HTTP status code may be returned instead.

7.1 Dictionary data structure

A dictionary is a set of names, each of which has an associated list of values. The names in a typical dictionary are unordered, although some encodings or language mappings may enforce a particular order.

A name must start with a letter, which can be followed by letters, underscores, and digits. Use camelCaps to indicate word breaks. Here are some legal names: "foo", "thisNameIsLonger", "tree_27".

Each name has an ordered list of one or more values associated with it, starting from an index of 0, with each value being either a Unicode string or a dictionary (that is, you can nest one dictionary within another). Other data types can be stored in dictionaries as strings; for example, an integer value is represented in decimal form by a string like `6358`, and a date is represented by a string like `2010-07-20`.

Note: There's no difference between a name being absent from a dictionary and the name being present with no values.

The Trio schema defines the types of structures and operations that dictionaries may contain, the fields that may be present in each type, and the type and number of values allowed for each field. Specifically, the schema defines:

- Structures (including objects) and operations involved in communicating with the Mind. The name of the structure or operation in the schema is the value of the `type` field in the dictionary.
- Enumerations (enums): lists of specific values that are acceptable for certain fields.
- Types for field values, including:
    - Fundamental (atomic) data types, such as `string`, `int`, `boolean`, and `dateTime`.
    - Unions, which specify a set (union) of types, any of which is acceptable for the value of the field.

   - `knownDict`, meaning that the field value is a dictionary of a defined (known) type that depends on the context in which the dictionary is being used (unlike most fields that contain a dictionary, which require a specific defined type or one of a union of defined types).
   - `anyDict`, meaning an arbitrary dictionary, whose contents are not defined by the schema. Fields within such a dictionary (which, unlike in other dictionaries, need not even include a `type` field) are not restricted in terms of the types of values or number of values they can hold.

The schema may provide more structure than allowed by the mapping of the dictionary type in a particular language. Using the mapping to the `Dict` class in the Java SDK as an example:

- Whereas `Dict` allows a single name to have both string and dictionary values, the schema requires all the values of a single name to be either strings or dictionaries.
- Whereas `Dict` allows a single name to have any number of values, the schema may restrict the number of values a particular field can hold.

For instance, the schema requires that the name `type` have exactly one value, which must be a string; this value declares which type of structure or operation (defined in the schema) the dictionary is representing.

7.2 JSON encoding

JSON (JavaScript Object Notation) is the recommended encoding to use for dictionaries. JSON is especially convenient and efficient for a web application that uses AJAX to communicate with the Mind, because JavaScript provides native support for JSON objects. The web site www.json.org has a clear and concise definition of JSON. Go there and take a look if you're not familiar with the format. For the full and complete specification, see RFC 4627.

A dictionary is represented as a JSON object in which the fields are specified by an unordered set of name-value pairs. The Mind orders names alphabetically (in ASCII collating sequence) within a JSON response dictionary but does not require alphabetical order in request dictionaries. Below are some conventions thatmake it easy to map back and forth between dictionaries and JSON objects without having to know the Mind schema: Fields that have no values do not appear in the JSON encoding.

If the schema defines a field name as having one value at most (`maxOccurs="1"`), the value is encoded as a single value in JSON. For example:

```
{
   "type": "contentStore",
   "title": "Jodi and her friends",
   "contentId": "tivo:ct.ts.1000701"
}
```

If the schema defines a field name as potentially having more than one value, the value or values of that field are represented in JSON as an array (with square brackets). In the following example, the `orderBy` field has only one value but can potentially have multiple values, so the single value is encoded as an array.

```
{
   "type": "offerSearch",
   "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
   "title": "The Sopranos",
   "orderBy": ["startTime"]
}
```

The next example illustrates a simple nested dictionary: the field named`lineup` has a single dictionary as its value (and can potentially have more than one value, hence the square brackets). Notice that the `type`value is not shown in the nested dictionary: although it's included in all such dictionaries that are returned by the Mind, it's required in request dictionaries only if the field can hold dictionaries of different types (otherwise the type is implied).

```
{
   "type": "collectionSearch",
   "categoryId": ["tivo:ca.413902","tivo:ca.413910"],
   "lineup": [
     {
        "lineupType": 84,
        "headend": "DITV501"
     }
   ],
   "orderBy": ["title"]
}
```

Because fields in an arbitrary dictionary (type `anyDict` in the schema) may have any number of values, this same square bracket syntax is used in representing the value or values of every field within such a dictionary.

The following table lists encodings for the different types of values that can be in a field:

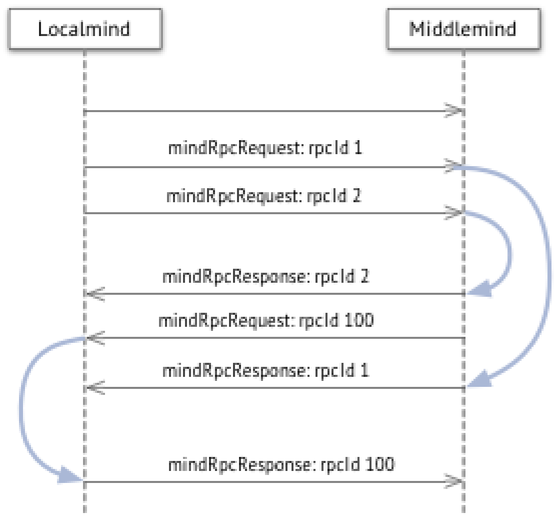
Type of Field	JSON Encoding
anyDict	JSON object, with every field holding a list. Because the maxOccurs of the fields is not known, every field is assumed to be multi-valued, so every field comes out as a list in JSON. Example: `{"size" : ["large"], "color" : ["red", "blue"]}`
anchorDict knownDict struct union	JSON object Example: `{"parentMixId":"tivo:mx.123", "type":"mixSource"}`
boolean	The json.org docs says there should not be quotes here, but Bravo, Iris, and Quattro/Tango accept either. Usually they are sent quoted. Example: `"true" or "false"`
int long	The number, without the quotes that would make it a string. Example: `8347`
enum	The string name of the enum item. Example: `"red"`
everything else	The string that appears in the dictionary.

The "type" field in the dictionary is encoded just like any other field. The ordering of fields in a JSON object is not defined. As an example, the following listings show a dictionary and an equivalent JSON object.

Dictionary	JSON object

`<recording>`
`  <category>tivo:ct.123</category>`
`  <category>tivo:ct.456</category>`
`  <credit>`
`    <first>James</first>`
`    <last>Gandolfini</last>`
`    <role>actor</role>`
`  </credit>`
`  <isEpisode>true</isEpisode>`
`  <size>5872</size>`
`  <title>The Sopranos</title>`
`</recording>`
	`{`
	`  "category": [`
	`    "tivo:ct.123",`
	`    "tivo:ct.456"`
	`  ],`
	`  "title": "The Sopranos",`
	`  "credit": {`
	`    "role": "actor",`
	`    "last": "Gandolfini",`
	`    "first": "James"`
	`  },`
	`  "isEpisode": "true",`
	`  "type": "recording",`
	`  "size": 5872`
	`}`

8 Mind RPC protocol



Sequencing of a theoretical Mind RPC session.

The Mind RPC protocol is used to make remote procedure calls to the Mind. A Mind RPC call is essentially an exchange of Trio dictionaries over a TCP connection, with a wrapper layer that matches requests and responses and allows multiple concurrent requests to be multiplexed over the same connection.

Version 2 of the Mind RPC protocol adds HTTP-style headers to each dictionary. These headers provide the following benefits:

- Requests can be routed without unmarshalling the contents.
- Internal information (such as authentication status and routing info) can be attached to a request without having to modify the body of the request.
- Logging-related information (such as application name) can easily be attached to a request.
- Application code can easily determine when a complete message has been received.

The Mind RPC protocol provides two key features that HTTP does not:

- It supports monitoring queries. After you've queried the Mind and received a response, the Mind can notify you later if there's a change to the response.
- When both sides of a connection have Mind services, calls can be made in both directions. Thus, in addition to receiving requests, the Mind can send requests to a client. This feature is useful for delivering asynchronous notifications and enabling remote access to a client.

8.1 Mind RPC Message Format

Each Mind RPC message consists of three parts, very similar in structure to an HTTP message:

1. A first line that describes the message.
2. A set of header lines.
3. The body of the request as a raw byte string.

8.1.1 First Line

The first line of a Mind RPC message provides data that describes the message. It contains:

- The string MRPC/2, which specifies the protocol and version.
- The size of the header in bytes. The header size does not include the first header line. It does include spaces and the one-byte CR/LF (carriage return/line feed) characters that delimit the header lines -- including the blank line between the header and the body -- although they are not shown in these examples.
- The size of the body in bytes. The body size does not include a CR/LF at the end.

Following is an example of a complete Mind RPC message. The first line begins with the string MRPC/2, followed by the number of bytes in the header (124 in this example: 117 characters + 7 CR/LFs). Mind RPC Version 2 does not support chunked transfer encoding (all messages must have a content-length).

The next value (18 in this example) specifies the number of bytes in the body of the message.

```
MRPC/2 124 18
Type: request
RpcId: 100/
SchemaVersion: 7
Content-Type: application/json
RequestType: infoGet
ResponseCount: single

{"type":"infoGet"}
```

8.1.2 Headers

A set of headers follows the first line of a Mind RPC message. The following headers are valid in all Mind RPC messages:

Header	Description
Type	One of: `request`, `response`, `requestUpdate`, or `cancel`. Required for all message types.
RpcId	The Mind RPC ID of the request: a 31-bit unsigned integer in decimal form. The ID is assigned by party creating a request, and must be unique for the given connection: the same ID must not be re-used before the connection is closed. All subsequent messages (`requestUpdate`, `response`, `cancel`) about the same request must use the same ID. Required for all message types.
SchemaVersion	The Mind schema version number for the body of the message. Required for request messages; not allowed in any other type of message.
Content-Type	One of the supported media types for Mind RPC data. For Mind RPC v2, the only supported value is `application/json`. Required for every message type except `cancel`.

As with HTTP messages, each Mind RPC header ends with a CR/LF character. There is an extra CR/LF after the last header, so there is a blank line between the headers and the message body. This line isn't really necessary (because the first line specifies the header length), but it was included for compatibility with existing HTTP header parsing libraries.

Note: Examples in this document do not show CR/LF characters.

Following is an example of a complete Mind RPC message. In addition to the Type and RpcId headers (required for all message types), this message includes the SchemaVersion header (required because the message type is request) and the optional Content-Type header.

```
MRPC/2 124 18
Type: request
RpcId: 100/
SchemaVersion: 7
Content-Type: application/json
RequestType: infoGet
ResponseCount: single

{"type":"infoGet"}
```

If the message type is `requestUpdate` or `cancel`, you only need to include the required headers. In fact, a `cancel` message does not have a body, so the SchemaVersion and Content-Type and headers are omitted. Here is an example of a cancel message:

```
MRPC/2 28 0
Type: cancel
RpcId: 100
```

8.1.3 Mind Custom Headers

There are many cases where callers want or need to pass custom headers to the Mind to enable certain functionality or logging. For example, the headers ApplicationName and ApplicationVersion provide information about the application making the request, while the ApplicationSession header identifies a set of interactions with a user.

The following table describes custom headers you can include in requests to the Mind.

Note: The TiVo service reserves the right to reject requests that do not include required headers.

Header	Description	Required?
ApplicationName • ~~X-Application~~ • ~~X-applicationName~~ • ~~X-ApplicationName~~	Unique name of the application in reverse domain notation. The name may be prefixed with the domain of the application. Example: `com.tivo.bravo`, where the application name is `bravo` and the domain is `com.tivo`. This value can be chosen arbitrarily by the application, but it must be unique to that application. The following application names are currently used: **com.tivo.atlas** SDUI for Series 4 retail boxes **com.tivo.bravo** ActionScript 2 implementation of HDUI version 1 for Series 4 retail boxes (Neutron, Lego) **com.tivo.bravo** ActionScript 2 implementation of HDUI version 1 for Virgin Media **com.tivo.encore** ActionScript 3 implementation of HDUI version 2 for Series 4 boxes (Neutron, Helium, Lego, Leo) This header is logged as part of the "operation log line" on the Mind application servers.	Yes
AccountId	TiVo AccountId associated with the request This header is used to ensure that the request only performs operations for data on the given account or data on bodies within the given account	No
AccountPartnerId	PartnerId of the MSO associated with the AccountId or PartnerAccountId in the request. This header is required when the request uses an accountTsn value for the bodyId.	Sometimes
ApplicationVersion • ~~X-applicationVersion~~ • ~~X-ApplicationVersion~~	Version of the application making the Mind call. Use this value to group results. It can be assigned arbitrarily by the application, and need not be unique between applications. This header is logged as part of the "operation log line" on the Mind application servers.	Yes

ApplicationSessionId • ~~X-applicationSessionId~~ • ~~X-ApplicationSessionId~~	ID of the session of the application making the Mind call. This value is a string that identifies a given user session (interaction with the user interface). It only needs to be unique between two sessions on a given client.	Yes
ApplicationFeatureArea • ~~X-applicationFeatureArea~~	Feature area that the request came from. This field can be used to measure performance by distinguishing calls to the same operation made from different parts of an application. You can use this value to group results. It can be assigned arbitrarily by the application, and need not be unique between applications.	Yes
BodyId	BodyId associated with the request. This overrides any bodyId present in the request payload.	No
HardwarePlatform	Descriptive name of the hardware platform that the caller is running. This value is used to group results, so it can be chosen arbitrarily by the application and need not be unique between applications. This field can be used to distinguish calls from various hardware platforms to determine cost of service.	Yes
HardwareIdentifier	Unique identifier of the hardware platform that the caller is running. This value is a unique string that identifies the hardware that is running the application making the call to the Mind. This field enables tracking of non-body-specific calls made from a given body for performance measurement.	Yes
HideAdult	Indicates whether adult content should be hidden for this request. Acceptable values are 1 (hide adult content) or 0. The value will override the bodyConfig.parentalControlsState of the TSN in the request. This is generally used by mobile or web-based applications to override the TSN setting based on the adult settings of the current login session.	No
PartnerAccountId	Partner's accountId that is linked to their TiVo Account. Together with AccountPartnerId, this is used to lookup the TiVo AccountId. This header is required when the request uses an accountTsn value for the bodyId.	Sometimes

MsoName	The name of the MSO, example RCN, Suddenlink, etc. The header could have three possible values -- MSO name,"retail", or null. Null is equivalent to unknown, a state expected on a new box.	Starting with Q2.8 on remotemind connection
X-customerId		
X-account	This header contains the accountId that is encrypted into the accountTokens generated by IT. It is not the same as the TiVo customerId, but in the future, we would like it to be.	No
X-clusterId	It tells to which cluster the request should be redirected to. By specifying X-clusterId=service, we are directing the request to service cluster DB. We should not force the request with bodyId going to service cluster by using header When both headers x-clusterId = service and route=serviceOnly are specified, then x-route will be ignored. When both headers x-clusterId = service and route=bodyOnly are specified, clusterId of the bodyClusterData will be considered as clusterId. and value of header x-clusterId will be ignored	Yes
X-partner		
X-requestId		
X-route	Tells the Mind to query the object in the same database that it was stored in, rather than sending the query to a replicated database that may be a few seconds behind in replication. This ensures that search requests performed after store requests return the latest data.	
X-isStaleDataOk		
X-authType		

Here's an example of a Mind RPC V2 query with custom headers.

```
MRPC/2 307 220
BodyId: tsn:7F60001901E2DF1
Content-Type: application/json
RequestType: candyBarPolicyGet
ResponseCount: single
RpcId: 768
SchemaVersion: 7
Type: request
X-RequestOrigin: internal
ApplicationName: Bravo
ApplicationSessionId: tsn:7F60001901E2DF1:33
ApplicationVersion: 1-9 2011.09.27-1121
```

```
{
    "bodyId": "tsn:7F60001901E2DF1",
    "includeCacheKeys": true,
    "screenIdentifier":
    {
        "collectionId":"tivo:cl.172902828",
        "screenName": "findActionContentScreen",
        "type":"hduiScreenIdentifier"
    },
    type":"candyBarPolicyGet"
}
```

8.1.4 Request Headers

A request message must include the following headers in addition to those required for all message types:

Header	Description
RequestType	Required. The type of the dictionary in the body of the request. Used to route the request to the right place and check permissions for the caller to make this type of request. Example: `subscriptionSearch`
ResponseCount	Required. One of: `none`, `single`, or `multiple`. This specifies the number of responses that the client is expecting. This header combines the `monitorFutureChanges` and `sendAndForget` flags from Mind RPC version 1.

Here is an example of a request message:

```
MRPC/2 124 18
Type: request
RpcId: 100
SchemaVersion: 7
RequestType: infoGet
ResponseCount: single
Content-Type: application/json

{"type":"infoGet"}
```

8.1.5 Response Headers

A response message may include optional headers to enable request metadata to be passed in both directions.

Header	Description
IsFinal	Optional. One of: `true` or `false`. Default: `true`. When this flag is `true`, there will be no more responses to the request. When `false`, more responses may come at some time in the future.

Here is an example of a response message. Note that the IsFinal header has the default value here, so it could be omitted:

```
MRPC/2 95 18
Type: response
RpcId: 100
SchemaVersion: 7
IsFinal: true
Content-Type: application/json

{"type":"success"}
```

8.1.6 Message Body

All Mind RPC message types other than `cancel` have a body. The message body comes after the headers. It begins with the open curly bracket character and ends with the close curly bracket character. In the following request, the message body is {"type":"infoGet"}.

```
MRPC/2 124 18
Type: request
RpcId: 100
SchemaVersion: 7
RequestType: infoGet
ResponseCount: single
Content-Type: application/json

{"type":"infoGet"}
```

For `cancel` messages, specify a body length of 0 in the first line, and leave the message body empty. Here is an example of a `cancel` message:

```
MRPC/2 28 0
Type: cancel
RpcId: 100
```

8.2 Semantics of Monitoring Queries

Each operation is allowed to define its own semantics of monitoring queries, but the general recommendation is to follow the standard outlined here.

The standard meaning of a monitoring query is, "Give me the answer, and give me updated answers as things change." The initial response to the query contains a full and complete answer. Later, if something changes, a new response is generated that also contains a full and complete answer.

Handling deltas in follow-on responses gets complicated, so the standard is to not do it.

As an example, consider a `recordingFolderItemSearch` operation that returns a list of all of the recordings on a DVR, used by a UI screen displaying those recordings. The UI would issue a `recordingFolderItemSearch` as a monitoring query and get the answer. If a recording gets deleted, the query service will create an updated answer for the query, the same answer it would return with a fresh new query, and return that response.

8.3 Request Updates

Request updates allow the original requestor to send more messages within the same request session. This facilitates full bidirectional communication for a given session. After a request is made, the sender may send `requestUpdate` messages with the same `rpcId` to update the handling of the original request.

9 Error handling

There are a few well-known ways in which operations requested of the Mind can fail: a bad argument (a field name not allowed by the schema), a duplicate object, or an internal error. Also, an application may be unable to contact the Mind at all, for any number of reasons. Language bindings may turn communication failures into exceptions, or into normal response dictionaries with an `error` dictionary as the sole response (just as if a "real" response had come from the Mind itself). In rare cases, it may not be possible for the Mind server to return an `error` dictionary; in that case, the server will return a standard 5*xx* (server error) HTTP status code.

Note: Incoming requests are rate-limited and may be throttled. If a client makes too many requests in a given time period, the Mind may reject the requests and return error messages.

An `error` dictionary contains the following required fields:

- `code`â ”The type of error (such as `badArgument`, `duplicateObject`, or `internalError`).
- `text`â ”A human-readable string describing what went wrong. (Do not make any assumptions about the format of this string.)

Here's an example `error` dictionary, returned after the caller sent the Mind a dictionary of type `sendMeAnError`, which is not a valid operation:

```
{
   "code": "badArgument",
   "text": "schemaViolation: type not defined: sendMeAnError",
   "type": "error"
}
```

If a Mind request returns a `serverBusy` error or a 5*xx* HTTP status code, the client should try again, using the following backoff algorithm to avoid making things worse for an already overloaded server. The algorithm is described in C, but it could be ported to other languages as needed.

```
#define SESSION_BACKOFF ( 15 * 60 )
#define SESSION_MINWIN ( 60 * 60 )
#define SESSION_MAXWIN ( 4 * 60 * 60 )
#define SESSION_WININCR ( 60 * 60 )

void CalculateNextRetryWait() {
   nRetrySlotM = (GetRandomInt() % nRetryCurrentWindowM);
   nRetryWaitM = nRetryNextAttemptM + nRetrySlotM;
   pRestartTimerM->SetDelay(Seconds(nRetryWaitM));
}

void ResetRetryValues() {
   nRetryNextAttemptM = nRetryBackoffTimeM;
   nRetryCurrentWindowM = nRetryMinWindowM;
   CalculateNextRetryWait();
}
```

```
void UpdateRetryValuesOnFailure(int nAttemptTime) {
   nRetryNextAttemptM = nRetryCurrentWindowM - nRetrySlotM;
   if (nAttemptTime < nRetryNextAttemptM) {
     nRetryNextAttemptM -= nAttemptTime;
   }
   if (nRetryCurrentWindowM < nRetryMaxWindowM) {
     nRetryCurrentWindowM += nRetryWindowIncrementM;
   }
   CalculateNextRetryWait();
}

void SetBootPhaseDelay() {
   int delay = RETRY_PHASE_BOOT_DELAY;
   pRestartTimerM->SetDelay(Seconds(delay));
}
```

10 Authenticating Requests

Not all client applications are created equal. Some applications will only be allowed to use a subset of the available Trio operations, and the subset may vary from application to application. An application must authenticate itself to gain access to the appropriate set of operations. To perform operations such as publishing program guide data, you must become a TiVo partner and use *certificate-based authentication*. TiVo issues each partner a partner ID, which is required for such operations, and an SSL certificate for authentication purposes. Being a partner also enables (and sometimes requires) you to link your customer accounts with TiVo customer accounts.

When you become a partner, TiVo sends you a passphrase-encrypted Personal Information Exchange file, which has the extension `.p12` and contains an SSL certificate and a key. A TiVo representative will give you the passphrase over the phone. You can use the file programmatically, import it into a web browser, or split it into a certificate and a key for command-line Mind queries. To split it into a certificate and a key, use the `openssl` command as follows:

```
openssl pkcs12 -in
```

You'll be prompted to supply the passphrase. The command will write the public certificate and private key to `stdout`; copy them to files with the extensions `.cert` and `.key`, respectively. You can then use `curl` from the command line to query the Mind, specifying these two files to authenticate the request. The following simple example uses `GET` with `curl`to request the operation `infoGet` (with no arguments); you can try it as a quick test of whether your certificate works.

```
curl -k --cert fileName.cert \
--key fileName.key \
-H 'Accept: application/json' \
'https://mind.tivo.com/mind/mind11?type=infoGet'
```

The response will be an `info` dictionary containing information about the Mind, similar to the following listing.

```
<?xml version="1.0" encoding="utf-8"?>
<info>
  <buildChange>b-server-011@608944</buildChange>
  <buildHost>corpspd05svr64</buildHost>
  <buildTime>2012-11-05 17:48:53</buildTime>
  <builder>build</builder>
  <clusterId>-1</clusterId>
  <mindVersion>7</mindVersion>
</info>
```

When accessing the Mind programmatically, send a Mind RPC call directly to the device. Your application will need to listen for the deviceâ ™s services (tivo_mindrpc) via Bonjour, and then connect to that device using port 1413.

Note: When connecting over HTTPS, it's strongly recommended that you implement SSL caching, which allows the client to make multiple SSL requests without the overhead of SSL negotiation for each request. In all other cases (including using a web browser or command-line interface to make Mind requests), you must authenticate each request.

11 Searching for Content

The *object*Search operation is used to find objects of type *object*. It searches the universe of objects, optionally filtering, ordering, and grouping them. Because the number of results found may be very large, *object*Search returns only a subset of them, called a *window*. This process can be represented conceptually by the following pipeline (where the optional parts are enclosed in square brackets):

all objects > [filter] > [order] > [[group] > [order]] > results list > window> response

Without grouping specified, *object*Search returns a list of objects (in an *object*List structure). With grouping specified, it returns a list of object groups (in an *object*GroupList structure), each of which contains only a count (of objects in the group) and an example object from the group. Also in the case of grouping, *object*Search will order the results both before and after grouping, to ensure that the requested ordering is observed for the example objects as well as for the groups as a whole.

Note: SQL users may find the filtering, ordering, and grouping to be familiar; they map to the `WHERE`, `ORDER BY`, and `GROUP BY` clauses in a `SELECT` statement. Simple, index-based windowing is usually available in SQL languages in the form of `LIMIT` or row clauses or as methods on the result set or cursor.

Search operations also let you specify what details you want the Mind to provide for each object in the response (either the exact fields to return or the general level of detail).

The example in the next section will give you an overview of how searching works.

11.1 Example search

The following example includes a Body ID in the search request, which instructs the Mind to use the channels available on that Body (as listed in the "Channels I Receive" list on the TiVo setup screens).

```
{
   "type": "offerSearch",
   "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
   "keyword": "friend",
   "orderBy": ["relevance"],
   "groupBy": ["collectionId"],
   "count": 2
}
```

When the Mind receives this request, it searches its database of current program guide data. Knowing which channels the Body can receive and record, it does the following:

1. Filters all offers to include only those that match the keyword `friend`.
2. Orders the matching offers by relevance to the search criteria.
3. Groups the offers by collection (for example, grouping together the offers for episodes of a particular TV series).

4. Orders the offer groups by relevance to the search criteria.
5. Returns a window consisting of the first two groups in the resulting list. For each group it returns, it includes an example offer from that group (with, by default, a low level of detail in terms of the number of fields returned). Because the ordering was done before as well as after the grouping, the group with the most relevant offer comes first in the list, with that offer provided as its example.

The Mind's response to the preceding query could be two offer groups such as the following:

```
{
   "isBottom": false,
   "isTop": true,
   "levelOfDetail": "low",

   "offerGroup": [
     {
        "count": 3,
        "example": {
           "channel": {
              "affiliate": "Satellite",
              "callSign": "FOODP",
              "channelId": "tivo:ch.8660782",
              "channelNumber": "35",
              "stationId": "tivo:st.13274992",
              "type": "channel"
           },
           "collectionId": "tivo:cl.8010225",
           "collectionType": "series",
           "contentId": "tivo:ct.18194348",
           "contentType": "video",
           "duration": 1800,
           "offerId": "tivo:of.ctd.13274992.2006-08-25-00-00-00.1800",
           "startTime": "2006-08-25 00:00:00"
           "subtitle": "Friend in Need",
           "title": "Barefoot Contessa",
           "type": "offer"
        },
        "levelOfDetail": "low",
        "type": "offerGroup"
     },

     {
        "count": 10,
        "example": {
           "channel": {
              "type": "channel",
              "affiliate": "A&amp;E",
              "callSign": "AETVP",
              "channelId": "tivo:ch.8660902",
              "channelNumber": "47",
              "stationId": "tivo:st.1433978"
           },
           "collectionId": "tivo:cl.6113467",
           "collectionType": "series",
           "contentId": "tivo:ct.35229067",
           "contentType": "video",
           "duration": 3600,
           "offerId": "tivo:of.ctd.1433978.2006-08-29-19-00-00.3600",
           "startTime": "2006-08-29 19:00:00"
           "subtitle": "Family Friend",
           "title": "Cold Case Files",
           "type": "offer"
        },
        "levelOfDetail": "low",
        "type": "offerGroup"
     }
   ],

   "type": "offerGroupList"
}
```

11.2 Filtering search results

Some of the fields available for filtering request an exact match with a field of the same name; for example, specifying a value in a `title` or `subtitle` field in a search request asks the Mind to find only objects that match that field exactly (except that the case of the characters in the strings being compared is ignored). Other filtering fields let you search for less exact matches (using keywords) or specify filtering based on time, category, and more. In general, adding additional filtering fields further refines the search (as `AND` does in SQL).

The following sections discuss some commonly used filters; however, not all types of search support each of these filters.

11.2.1 Filtering using keywords

Keyword filters specify strings for the Mind to compare to the values of certain string fields, limiting the search results to only those objects in which the field values match. The match need not be exact: the specified string may be a substring of the target field's value, and the case of the characters in the strings is ignored.

The `keyword` filter field requests comparing its value to those of the `title`, `subtitle`, and `description` fields. Other filters, such as `titleKeyword`, `subtitleKeyword`, `descriptionKeyword`, and `creditKeyword`, limit the comparison to a single field. For example, the following offer search finds a match only if the indicated keyword is in the value of the `title` field:

```
{
 "type": "offerSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titleKeyword": "friend",
 "orderBy": ["relevance"]
}
```

There are also `titlePrefix` and `subtitlePrefix` fields, requesting a match starting from the beginning of the title or subtitle.

11.2.2 Filtering by time

Some `objectSearch` operations let you filter based on time, with fields such as`minStartTime`, `maxStartTime`, `minEndTime`, and `maxEndTime`. For example:

```
{
 "type": "offerSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titleKeyword": "friend",
 "minStartTime": "2005-10-30.23:00",
 "orderBy": ["relevance"]
}
```

11.2.3 Filtering using categories

You can discover the Mind's category hierarchy with the `categorySearch` operation. The following query requests only the top-level categories. Every category has a text label, and this query requests that the results be ordered alphabetically by label (limiting the returned items to the first 25 results).

```
{
 "type": "categorySearch",
 "topLevelOnly": true,
 "orderBy": ["label"],
 "count": 25
}
```

A returned category ID can serve as a filter in other searches. The response to the preceding query revealed that among the top-level categories is one with ID `tivo:ca.413902` (Movies); the following category search requests only the subcategories of that category (also limiting the returned items to the first 25 results, although alternatively it could get the exact number of subcategories from the `nChildren` field of the parent category).

```
{
 "type": "categorySearch",
 "parentCategoryId": "tivo:ca.413902",
 "orderBy": ["label"],
 "count": 25
}
```

This next query asks the Mind to look only in category `tivo:ca.413902` for offers with a title that matches the keyword "friend":

```
{
 "type": "offerSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titleKeyword": "friend",
 "categoryId": ["tivo:ca.413902"],
 "orderBy": ["relevance"],
 "count": 10
}
```

11.3 Ordering search results

You can use the `orderBy` field to specify the order in which the Mind should return search results. The schema defines the `orderBy` values that are valid for each`objectSearch` type. Those for an offer search, for example, include `relevance`, `title`, and `collectionId`, among many others.

Although ordering is optional and a reasonable default ordering will be used if `orderBy` is omitted, specifying the order explicitly is recommended so that the client will know what results to expect. Note, however, that since the ordering of IDs assigned by the Mind is unpredictable, you should order by such an ID only when the exact order is not important (although the Mind will use the same ordering method each time, so the results will be stable in that sense).

The following offer search request uses `orderBy` to specify that results be returned in order of relevance to the search criteria.

```
{
   "type": "offerSearch",
   "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
   "keyword": "friend",
   "orderBy": ["relevance"],
   "count": 10
}
```

In this case the Mind would decide that an offer whose title starts with "friend" would come before an offer whose title contains "friend" later in the title, and both would come before an offer whose subtitle contains "friend".

Putting a hyphen (`-`) in front of an `orderBy` value reverses the order. Each of the following queries returns one offer (because the count isn't specified); the first query returns the first offer in the list of results, and the second query returns the last offer in the list (as specified by `-title`).

```
{
 "type": "offerSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titleKeyword": "Jump",
 "orderBy": ["title"]
}
{
 "type": "offerSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titleKeyword": "Jump",
 "orderBy": ["-title"]
}
```

11.4 Grouping search results

You can use the `groupBy` field to group the found objects instead of having them returned individually. The value of `groupBy` specifies a field name; all objects with the same value in that field will be grouped together. Without grouping specified, an *`object`*`Search` operation will return an *`object`*`List` dictionary, in which each object of type *`object`* in the response is stored as a value of the name *`object`*. With grouping specified, *`object`*`Search` will return an *`object`*`GroupList` dictionary, which contains an ordered list of *`object`*`Group` structures, each in turn containing a count of the objects in the group and only one example object from the group.

The basic structure of the returned `offerGroupList` dictionary is as follows:

```
{
 ...
 "offerGroup": [
  {
   "count": 3,
   "example": {
    ...
    "type": "offer"
   },
   ...
   "type": "offerGroup"
  },
  {
   "count": 10,
   "example": {
    ...
    "type": "offer"
   },
   ...
   "type": "offerGroup",
  }
 ],
 "type": "offerGroupList"
}
```

11.5 Specifying the search results window

There are three different ways of specifying the window of search results to return. The simplest method, which uses only the following fields, is effective for lists that won't change while being viewed.

- `count` â ” The desired number of items (objects or object groups) in the window, ranging from 1 (the default value) to a maximum of 50.
- `offset` â ” An offset from the beginning of the results list, indicating the first item to include in the window. Its default value is 0 (the first item in the results list).

For example, you would specify `count=2` to get the first two items in the results list, or `count=3` and `offset=2` to get the third through fifth items.

If you request more items than are available, the search returns only as many as are available. For instance, if the results list contains 15 items in all, a request for 10 items starting from an offset of 10 will return only 5 items (10 through 14). Also in that case, the Mind will set the `isBottom` field to `true` to indicate that the window includes the last item in the results list. If `isBottom` is `false`, you can query for more results in the list. Similarly, the `isTop` field is set to `true` if the window includes the first item in the results list (or if the results list is empty).

Some search operations include a `noLimit` field, which you can set to `true` to remove the restriction set by the `count` parameter and instead request all items in the results list.

11.5.1 Anchoring

An additional field, `anchor`, provides a way of accommodating lists (such as the To Do and Now Playing lists) that can change while being viewed. The offset from the beginning of such a list can change over time, leading to missed or repeated items when the user is scrolling or to an inability to maintain the position of the current visual highlight in the window when the list contents are updated after a change. Also, there are sometimes cases where you need to initially position the window in the middle of a very long list (such as the alphabetical listing of all program titles for the next two weeks); since in these cases you have no idea of the offset from the beginning of the list, you can't request the window that way.

The solution is to measure the offset relative not to the beginning of the list but to an *anchor* item, specified by a dictionary of the same type as the return type for the search query (for example, the anchor in an `offerSearch` request must be an offer). Typically, the anchor is one of the objects returned from a previous search and corresponds to the currently highlighted item in the list. If the anchor item itself goes away, the Mind will find a substitute anchor that best matches the one you specified â ” usually one nearby, but possibly a reasonable substitute elsewhere. This scheme enables you to scroll or request data about the list while being sure that you're always keeping the highlight in the same position (or close to it).

To summarize this method:

- The `count` field specifies the number of items in the window, ranging from 1 (the default value) to 50.
- The `anchor` field specifies an item in the list, and the value of the `offset` field (which can be negative) is an offset from the anchor item, indicating the first item to include in the window. The default offset is 0.

11.5.2 ID resolving

Even the anchoring method just described has disadvantages. For example, it can be an expensive operation for the Mind to find the anchor in a results list, sometimes requiring searching through the list from the beginning until the anchor item is found each time a query is made; in the worst case, this can slow down scrolling linearly (or worse). For this reason, search operations offer the alternative of doing ID resolving instead, making it much easier to do scrolling because you always know the total number of items in the list.

ID resolving involves two basic steps:

1. First you get essentially the entire results list but requesting only very minimal information for each object: specifically, only the value of its `objectIdAndType` field, which is an integer that uniquely represents a combination of the object's ID and type.
2. Then you request a typical (smaller) window based on a subset of `objectIdAndType` values: you essentially resolve each such value by getting the object that matches it along with whatever details (fields) you want for that object.

This way, you have a stable list that won't change out from under you, but without the overhead of getting details for the entire list; you get details only for those items included in the final window.

Note: The integer value of `objectIdAndType` is opaque; you should not attempt to decode it into its component parts.

Looking at this method more closely (and with an example), here's one way you might do it:

Perform a search query to get all the items in the list (up to some reasonable maximum count), setting the `format` field to `idSequence` to request that the results be returned not as objects themselves but as `objectIdAndType` values. For this purpose, the Mind allows a maximum value of 1000 for the `count` field (although 50 is considered a reasonable maximum in the following example).

```
{
 "type": "collectionSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titlePrefix": "friend",
 "orderBy": ["title"],
 "format": "idSequence",
 "count": 50
}
```

The response in this case consists of all the items found (a total of nine):

```
{
 "isBottom": true,
 "isTop": true,
 "objectIdAndType": [
   344155784192181,
   344155797280651,
   344155797441401,
   344155799786451,
   344147139557671,
   344155742201111,
   344147139510533,
   344147141491285,
   344147217883251
 ],
 "type": "idSequence"
}
```

When you need a window of items, perform another search that gets the appropriate details corresponding to the desired subset of previously returned `objectIdAndType` values. The following example requests (by default) a low level of detail for the four collections whose `objectIdAndType` values are specified:

```
{
 "type": "collectionSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "objectIdAndType": [
   344155799786451,
   344147139557671,
   344155742201111,
   344147139510533
 ]
}
```

Notice that a search filtered by `objectIdAndType` doesn't specify a count or an order; the Mind returns every matching object, in the order that the `objectIdAndType` values were listed in the query. The response to the preceding example query would be a `collectionList` structure containing the matching collections in the order listed.

Two additional fields, `resolveCount` and `resolveOffset`, enable you to streamline this process by combining both steps into a single query, as follows:

```
{
 "type": "collectionSearch",
 "bodyId": "tivo:bd.1tKTkb2jrgNRWqfDw_uULY6OyBlI-",
 "titlePrefix": "friend",
 "orderBy": ["title"],
 "format": "idSequence",
 "count": 50,
 "resolveCount": 4,
 "resolveOffset": 4
}
```

To summarize the fields involved in searching using ID resolving:

- The `format` field is set to `idSequence` to specify that only `objectIdAndType` values be returned initially (for the entire list).
- The `count` field specifies the number of items to return, initially a number large enough (up to 1000) to get the entire list, and, in a subsequent search query filtered by `objectIdAndType` values, only as many items as are desired in the window (ranging as usual from 1 to 50).
- The `offset` field specifies an offset from the beginning of the results list, initially 0 to get the list starting from its beginning, and, in a subsequent search query, only as many items as are desired in the window.
- The `resolveCount` and `resolveOffset` fields serve the same purpose as `count` and `offset` in a subsequent query but are specified in the same query as the one requesting the entire list. (There's also a `resolveLevelOfDetail` field, analogous to the `levelOfDetail` field; the `responseTemplate` field always applies, even when ID resolving is done.)

11.6 Specifying the search results details

You won't always want all details about all the objects found as the result of a search. The Mind's process of computing the details, writing them in responses, and sending them is expensive. If you ask for fewer fields to be filled in, you'll get a faster response from the Mind. Two alternatives for specifying the fields to return are available:

- You can specify the exact fields to return, in a `responseTemplate` structure.
- You can specify the general level of detail to return, in the `levelOfDetail` field.

In general, it's more efficient (for both the Mind and the client) to specify the exact set of fields, so using a response template is recommended unless you need to get all the fields in each object or are requesting a very small count.

Note that in any case, the Mind does not return optional fields for which no value has been set and for which there is no default value. Also, if you set the `format` field in a search operation to request that something other than the objects themselves be returned, the Mind ignores the `responseTemplate` and `levelOfDetail` fields.

11.6.1 Response template

Each search request can include one or more `responseTemplate` structures, each of which indicates the set of fields that should be returned for a given type of structure. This allows you to specify the desired fields for both the main returned objects and any substructures they might contain.

Each response template specifies the type of structure it applies to and the field or fields to include in that type of structure in the response. You don't have to specify required fields (such as `type`), which are returned for all structures by default; however, if you prefer to explicitly specify all the fields to return, you can change this default behavior by setting `includeRequiredFields` in the response template to `false`.

In the response template, you can specify a field by name alone in the `fieldName` field, or with additional information in the `fieldInfo` field â ” specifically, the maximum *arity* or number of values to return for the field. The response template in the following `collectionSearch` example illustrates both methods, requesting the field or fields to return in these structures:

- In `collectionList`, only `collection`.
- In `collection`, only `collectionId`, `title`, `movieYear`, and `credit`, and a maximum of two values for `credit`. Although every collection has a `collectionId`, that field isn't strictly required according to the schema, so you must specify it in the response template if you want it to be returned.
- In objects of type `credit`, only `first` and `last`.

```
{
  "type": "collectionSearch",
  "collectionType": ["movie"],
  "titlePrefix": "Shrek",
  "descriptionLanguage": "English",
  "orderBy": ["title"],
  "count": 3,
  "responseTemplate": [
    {
      "type": "responseTemplate",
      "fieldName": ["collection"],
      "typeName": "collectionList"
    },
    {
      "type": "responseTemplate",
      "fieldName": ["collectionId", "title", "movieYear"],
      "fieldInfo": [
        {
        "type": "responseTemplateFieldInfo",
        "fieldName": "credit",
        "maxArity": 2
        }
      ],
      "typeName": "collection"
    },
    {
      "type": "responseTemplate",
      "fieldName": ["first", "last"],
      "typeName": "credit"
    }
  ],
}
```

The dictionary returned in response to this request would look like this:

```
{
 "collection": [
  {
   "collectionId": "tivo:cl.3839776",
   "credit": [
    {
     "first": "Mike",
     "last": "Myers",
     "type": "credit"
    },
    {
     "first": "Eddie",
     "last": "Murphy",
     "type": "credit"
    }
   ],
   "movieYear": 2001,
   "title": "Shrek",
   "type": "collection"
  },
  {
   "collectionId": "tivo:cl.36556485",
   "credit": [
    {
     "first": "Mike",
     "last": "Myers",
     "type": "credit"
    },
    {
     "first": "Eddie",
     "last": "Murphy",
     "type": "credit"
    }
   ],
   "movieYear": 2004,
   "title": "Shrek 2",
   "type": "collection"
  },
  {
   "collectionId": "tivo:cl.94030578",
   "credit": [
    {
     "first": "Mike",
     "last": "Myers",
     "type": "credit"
    },
    {
     "first": "Eddie",
     "last": "Murphy",
     "type": "credit"
    }
   ],
   "movieYear": 2010,
   "title": "Shrek Forever After",
   "type": "collection"
  }
 ],
 "type": "collectionList"
}
```

11.6.2 Level of detail

The `levelOfDetail` field (or the `resolveLevelOfDetail` field if you're doing ID resolving) specifies whether the Mind should return a low, medium, or high number of fields. It's ignored if you specify a response template.

The values of this field and their meanings are as follows:

- `low` (the default value) provides minimal text information, usually IDs and text, for simple display.
- `medium` adds most non-chunky fields (where chunky refers to fields that have an unbounded number of values or a complex value).
- `high` adds the rest of the fields.

Note that all fields in the `low` level of detail are included in the `medium`level, and all fields in `medium` are included in `high`.

11.7 Using annotations

Annotation is a Trio mechanism whereby the results of a search can be "annotated" with data from follow-up queries automatically.

Some operations return annotations in `noteContainer` structures. You can also explicitly request annotations in a search dictionary.

To request an annotation, include a `note` attribute in the search dictionary. For example, the following listings show two `adSearch` queries; the query on the left doesn't include a `note` attribute, the query on the right does.

Without a note	With a note

`<adSearch>`
`   <bodyId>tsn:0001</bodyId>`
`   <count>10</count>`
`</adSearch>`
	`<adSearch>`
	`   <bodyId>tsn:0001</bodyId>`
	`   <count>10</count>`
	
	`<note>adInterestForAdId</note>`
	
	`</adSearch>`

The following shows sample query results. The extra data requested via the `note` attribute is highlighted. Notice that `adInterestForAdId` has a value for the second `ad`.

```
<adList>
  <ad>
    <adBrandId>tivo:ab.1000024</adBrandId>
    <adId>tivo:ad.1000034</adId>
    <description>Nike (long): second ad.</description>
    <imageUrl>http://www.nikead2.com/logo.jpg</imageUrl>
    <offerId>tivo:of.std.727.2005-05-01-02-00-00.1800</offerId>
    <shortDescription>Nike: second ad.</shortDescription>
    <title>Nike Air Jordan</title>
    <type>ad</type>
  </ad>
  <ad>
    <adBrandId>tivo:ab.1000024</adBrandId>
    <adId>tivo:ad.1000044</adId>

     <adInterestForAdId>
       <adId>tivo:ad.1000024</adId>
       <adInterestId>tivo:at.1000074</adInterestId>
       <bodyId>tsn:0001</bodyId>
       <type>adInterest</type>
     </adInterestForAdId>

<description>Second Nike (long): second ad.</description>
    <imageUrl>http://www.nikead2.com/logo.jpg</imageUrl>
    <offerId>tivo:of.std.727.2005-05-01-02-00-00.1801</offerId>
```

```
      <shortDescription>Second Nike: second ad.</shortDescription>
      <title>Second Ad Title</title>
      <type>ad</type>
    </ad>
  </adList>
```

The note `adInterestForAdId` is supported by `adSearch`. For each ad returned from this query, the note causes a follow-up query to be run. This query takes the `adId` for each returned `ad` and searches for an `adInterest` for that `adId` for the body in the query. It "annotates" each `ad` result with the `adInterest` (if any) that it finds.

That's basically all a client application needs to know. To perform a follow-up query for each item in your search results, find the appropriate note. Then, put that name in a `note` attribute on your search. Look for the results, if any, on each item in your result. Some notes include other notes so that you don't have to list them all individually.

12 Glossary

account

In general, identification by username and password, for authentication purposes. The account object in the Mind contains data related to a TiVo user account (including personal data, preferences, content bookmarks, access rights, and so on for the account).

anchor

A field that may be used in specifying the results window returned by a search operation. Best used for lists that can change while being viewed, it specifies an object to serve as a starting point for the window (along with an `offset` field specifying an offset from the anchor item).

API

Application programming interface; used in this guide to refer to any method of accessing the Mind, whether programmatically or directly from a web browser or command line.

API key

A name that uniquely identifies a Mind client to TiVo and that authorizes the client to use the Mind API to perform a subset of operations (excluding, for example, publishing program guide data). The TiVo developer website includes a page for generating an API key.

Body

A consumer device whose behavior can be controlled by the Mind (at least partially). There are currently two types of Bodies: TiVo DVRs, and personal computers running software that make them act like TiVo DVRs.

Body-relative

Unique within one Body. Some objects have Body-relative IDs as opposed to IDs that are globally unique (that is, global within the Mind).

category

A classification of content, such as "Movies" (at the top level) or "Comedy"(at a lower level), that's part of a read-only hierarchy maintained by the Mind. The client can discover the category hierarchy through searching and then limit a content search to one or more specific categories.

collection

A grouping for related digital media content, such as a TV series, a music album, or a photo stream. Every content object is part of a collection, even if the collection has only one thing in it; for example, every movie has a collection object as well as a content object.

content

Often used loosely in this guide to refer to digital media in general, but spelled out where necessary for clarity as *content object*, meaning an object containing data related to a piece of digital media such as a single TV show (episode), movie, song, or image.

dictionary
A set of name-value pairs corresponding to fields that specify structures (including objects) and operations used in communicating with the Mind, as defined by the Trio schema. The value of a dictionary's `type` field indicates the type of structure or operation that the dictionary represents.

Face
An application that uses the Mind, typically providing an interactive user interface to a consumer. Faces use the services provided by the Mind to get data and to provide instructions to the Mind, and to Bodies via the Mind, based on the user's actions.

field
An element of a structure or operation specified in a dictionary; designated by a name-value pair.

Mind
A set of services, made available by TiVo's servers, that provide access to information stored on the servers, such as program guide data, and also enable control of certain consumer devices, such as TiVo DVRs.

Mind RPC
A protocol defined by TiVo; used in communicating with the Trio Mind.

mix
An object that groups collections, content, and offers into a tree structure to present to the user. Users can subscribe to all or part of the tree.

object
A basic data structure involved in communicating with the Mind. Objects are typically referred to (by ID) in dictionaries requesting an operation or are included in dictionaries sent in response to requests.

offer
A specification of how to obtain content, whether from broadcast TV, a cable system's video on demand (VOD) service, a TiVo content delivery service (CDS), download sites like BitTorrent, or similar content providers. In addition to the channel and time or the URL for obtaining the content, an offer includes related information such as the content type, provider, cost, and access rights.

operation
An entity, specified in a dictionary as defined by the schema, that requests something of the Mind.

partner
A Mind client developer that receives a certificate and a partner ID from TiVo, enabling the client to perform operations such as publishing program guide data. Partners can also link their customer accounts with TiVo customer accounts.

recording
The object that contains (or will contain) actual bits of media that reside on a device as specified by an offer object. Recording objects can refer to offers that either have already been captured on disk (recorded or downloaded) or are scheduled for capture.

schema

The definition of what a dictionary sent or received in communicating with the Mind may contain: the types of structures and operations allowed, the fields within each type, and the type and number of values allowed for each field.

subscription
An object representing that a given Body should get (record or download) certain content that the user is interested in.

Trio
A client-server architecture in which the Mind is the server providing TiVo services to clients. The typical client is an application running on a consumer device (such as a TiVo DVR or a personal computer), but the client can also be the consumer device itself or even another server.

window
A subset of results returned by a search operation, specified by a count of items in the window and its offset from the beginning of the list or from an anchor item, or by ID resolving.