MQL Reference Guide

David Flanagan

MQL Reference Guide

David Flanagan

Published 2009-01-13

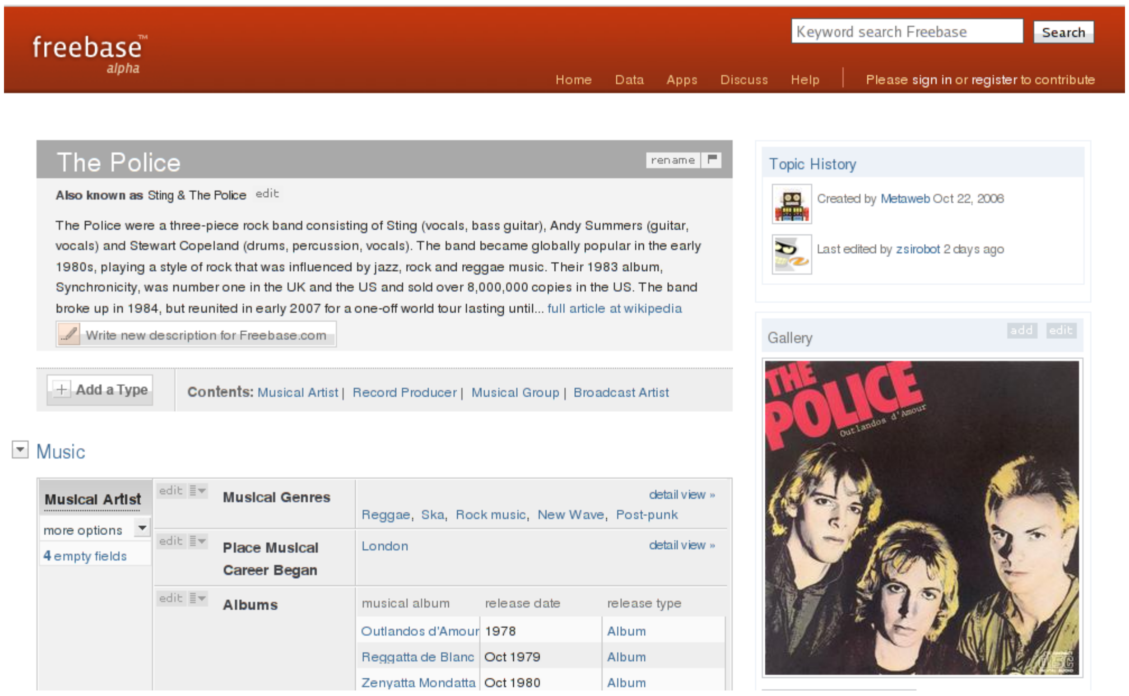
Copyright © 2008 Metaweb Technologies, Inc.

Table of Contents

List of Examples

1 Introduction

Freebase is a vast, free, open online database of structured knowledge, powered and maintained by Metaweb Technologies (*metaweb.com*). If you visit the *freebase.com* website, you'll find information about such diverse topics as popular music, movies, geography and genetics. Figure 1.1 is a sample page from this site: [1] Users can access and contribute to Freebase at *http://www.freebase.com*, or by using the API explained in this manual.



The Police

Also known as Sting & The Police

The Police were a three-piece rock band consisting of Sting (vocals, bass guitar), Andy Summers (guitar, vocals) and Stewart Copeland (drums, percussion, vocals). The band became globally popular in the early 1980s, playing a style of rock that was influenced by jazz, rock and reggae music. Their 1983 album, Synchronicity, was number one in the UK and the US and sold over 8,000,000 copies in the US. The band broke up in 1984, but reunited in early 2007 for a one-off world tour lasting until... full article at wikipedia

Write new description for Freebase.com

Contents: Musical Artist | Record Producer | Musical Group | Broadcast Artist

Music

Musical Artist — more options — 4 empty fields

Musical Genres: Reggae, Ska, Rock music, New Wave, Post-punk

Place Musical Career Began: London

Albums

musical album	release date	release type
Outlandos d'Amour	1978	Album
Reggatta de Blanc	Oct 1979	Album
Zenyatta Mondatta	Oct 1980	Album

Topic History

Created by Metaweb Oct 22, 2006

Last edited by zsirobot 2 days ago

Gallery

Figure 1.1. Browsing knowledge at freebase.com

This manual teaches you how to write applications that interact with Freebase. It assumes that you already know the "what" and "why" of Freebase, and that you have visited the Freebase Help Center[2], viewed the Freebase tutorial[3] and read the Freebase FAQ[4].

[1] The screenshot is of the URL *http://freebase.com/view/en/the_police* At the time of this writing (September, 2008) the Freebase user interface is still in flux, and you may find that this page looks different when you visit it.
[2] http://www.freebase.com/help/
[3] http://www.freebase.com/help/tutorial/index.html
[4] http://www.freebase.com/help/faq

Metaweb and Freebase

Metaweb (the company) has developed Metaweb (the database and API). Freebase (the open global structured knowledge base) is a high-profile public instantiation of the Metaweb technology. This manual documents general Metaweb services and APIs, and relies on Freebase for example data and applications. The services and APIs are applicable beyond Freebase, however, and you'll find that this manual uses the name "Metaweb" far more than it does the name "Freebase".

1.1. The Metaweb Query API

Metaweb offers a powerful API for making programmatic queries. This allows you to incorporate knowledge from the Freebase database into your own applications and websites. Let's take this API for a spin. Type the following URL into your web browser's location bar. (Type it all on a single line: it is broken across two lines here to fit on the page.)

```
http://api.freebase.com/api/service/mqlread?query=
{"query":{"type":"/music/artist","name":"The Police","album":[]}}
```

There are a lot of braces, quote marks, colons and commas in that URL, but remember that this is a programmatic API: the query is supposed to be generated by a computer, not pecked out by human fingers! The query is easier to understand if it is formatted like this:

```
{
  "query": {
    "type":"/music/artist",
    "name":"The Police",
    "album":[]
  }
}
```

Translated into English, this Metaweb query says:

> Find an object in the database whose type is "/music/artist" and whose name is "The Police". Then return its set of albums.

If you got all the punctuation correct, the Metaweb server running at *api.freebase.com* will return a plain-text response that looks something like this:

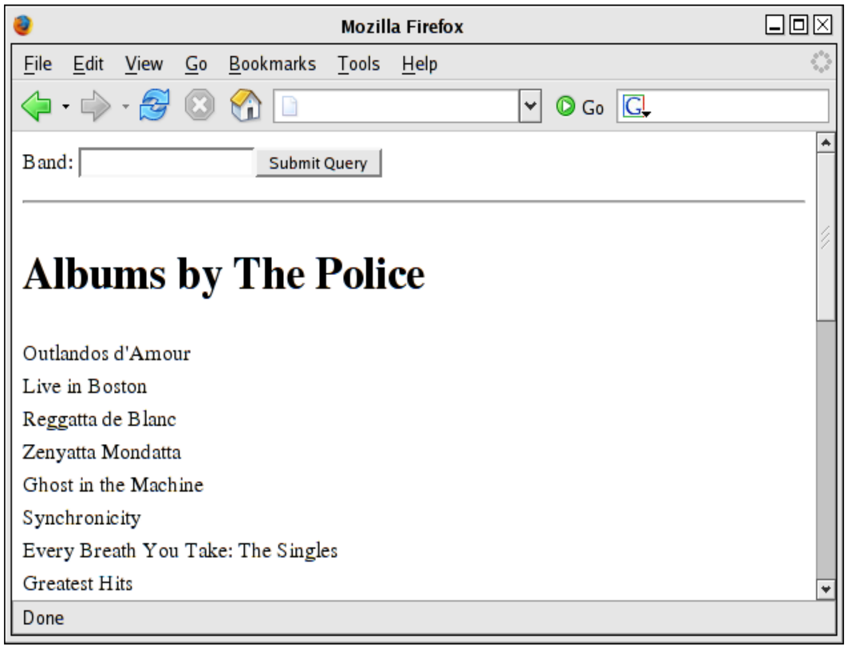
```
{
  "status": "200 OK",
  "code": "/api/status/ok",
  "transaction_id":"cache;cache01.p01.sjc1:8101;2008-09-18T17:56:28Z;0029",
  "result": {
    "type": "/music/artist",
    "name": "The Police",
    "album": [
      "Outlandos d'Amour",
      "Reggatta de Blanc",
      "Zenyatta Mondatta",
      "Ghost in the Machine",
      "Synchronicity"
    ]
  }
}
```

The response has the same braces and quotes that the query did: they provide the structure that makes this response easy to parse (for a computer). This response contains status codes and then a property "result". This `result` property of this response has the same structure as the `query` property of the query did, except that the empty parts have been filled in. Our query included the text:

```
"album":[]
```

In the response, those empty square brackets have been filled in with a long list of album names. (For brevity, many of albums have been omitted in the result shown above.)

Making queries from your web browser's location bar is interesting, but it becomes much more interesting if we make the queries under programmatic control. Imagine a script running on your own web server that sends queries to Freebase and formats the results as HTML: this is a Freebase-enabled web application. It might look like Figure 1.2.



Albums by The Police

Outlandos d'Amour
Live in Boston
Reggatta de Blanc
Zenyatta Mondatta
Ghost in the Machine
Synchronicity
Every Breath You Take: The Singles
Greatest Hits

Figure 1.2. A Metaweb-enabled web application

1.2. About this Manual

The goal of this manual is to explain everything you need to know to create Metaweb-enabled web applications like the one pictured in Figure 1.2. We assume that you are a programmer who has some experience with languages like PHP, Python, Perl and JavaScript, and that you understand the basics of the HTTP protocol.

The chapters that follow are:

Chapter 2: *Metaweb Architecture*
: This chapter explains the Metaweb architecture, including a discussion of Metaweb objects, values, types, domains, namespaces and access control.

Chapter 3: *The Metaweb Query Language*
: This chapter explains the Metaweb Query Language (MQL) that is used to express Metaweb queries. The syntax is quite a bit more powerful and complex than what was shown in this introduction.

Chapter 4: *Metaweb Read Services*
: This chapter explains how to read data from Metaweb servers. It demonstrates (with examples in a variety of scripting languages) how to submit MQL queries to the *mqlread* service, and how to interpret the response. It also demonstrates how to use the *trans* service to download content (such as HTML articles or binary image data) from Metaweb.

  This chapter includes the source code for the application that created Figure 1.2.

Chapter 5: *The MQL Write Grammar*
: This chapter explains MQL syntax for writing to the Metaweb database. The MQL write grammar resembles MQL for read queries, but is different in a number of important ways.

Chapter 6: *Metaweb Write Services*
: This chapter explains how to write to Metaweb servers. It demonstrates (with Python code) the *mqlwrite* service for sending write queries to Metaweb, and the *upload* service for uploading textual or binary content. In addition, it shows how to use the *login* service, which is a necessary prerequisite for *mqlwrite* and *upload*.

2 Metaweb Architecture

The database that underlies Metaweb is fundamentally different than the relational databases that you may be familiar with. Relational databases store data in the form of tables, but the Metaweb database stores data as a graph of nodes and relationships between those nodes. Relational databases use the SQL query language and accept queries and return results using a specialized network protocol. Metaweb uses the MQL query language and communicates via standard HTTP requests and responses. The bulk of this manual is devoted to explaining MQL and demonstrating its use with the *mqlread* and *mqlwrite* web services. But because Metaweb's underlying database technology is new and different, it will be helpful if you understand the fundamental architecture of Metaweb first. This chapter explains the graph-based representation of Metaweb data, then shows how the graph of nodes and relationships can be viewed as a collection of objects. It also covers a number of other important architectural details, explaining properties, types, domains, names, ids, namespaces and access control. If you find this chapter difficult, feel free to skip ahead and skim Chapter 3 and Chapter 4 to get an overview of MQL and the *mqlread* service. With that context you can return to this chapter to solidify your understanding of the Metaweb architecture.

In addition to the Metaweb graph database, Metaweb servers also implement a content store. The content store is responsible for storing chunks of binary data (in SQL these chunks are called BLOBs) and is tightly integrated with the graph database. Each chunk in the content store has a corresponding node in the graph, and metadata about the content (such as its MIME type) is stored as relationships in the graph. Interestingly, the SHA2 hashcode of a chunk is used as the key into the content store, which makes it possible to test whether the store contains a specific chunk without uploading that chunk, and also prevents duplication of entries in the content store. This chapter does not cover the content store; we'll learn how to download data from the content store in Chapter 4 and how to upload data to it in Chapter 6.

2.1. Nodes and Relationships

When viewed at the lowest level, the Metaweb graph is a set of nodes and a set of links or relationships between those nodes. Each node has a unique identifier (so it can be named and referred to) and a record of when and by whom it was created. Other than the id, timestamp and creator, however, the nodes in the graph hold no information themselves. All the interesting data in the database is stored in the form of relationships between nodes (or between nodes and primitive values).

Graphs can be represented visually using circles to represent nodes and arrows between the circles to represent relationships. In this section, however, we will model the relationships in the Metaweb object store as a tuples of four pieces of data [1] and we will represent sets of these relationships in tabular form, where each row of a table specifies one tuple.

The table below, for example, represents information about The Police, their album Zenyatta Mondatta, and the song Driven to Tears on that album. The **From** column identifies the node that is the subject of the relationship. This node could also be called the "source" node or the "left" node. The **Property** column specifies the kind of relationship being described. (Notice that this complicates the visual representation of a Metaweb graph: each arrow in the diagram must be tagged with a property to specify what kind of relationship it represents.) The **To** and **Value** columns specify the object of the relationship. **To** (which could also be called "target", "destination", or "right" specifies another node, and **Value** specifies a primitive value such as a string of text, a number or a date. One or both of these columns may have a value, depending on the kind of relationship (the **Property** column) being described.

From	**Property**	**To**	**Value**
/en/the_police	/type/object/name	/lang/en	The Police
/en/zenyatta_mondatta	/type/object/name	/lang/en	Zenyatta Mondatta
/guid/1234	/type/object/name	/lang/en	Driven to Tears
/en/zenyatta_mondatta	/music/album/artist	/en/the_police	
/guid/1234	/music/track/album	/en/zenyatta_mondatta	
/guid/1234	/music/track/length		200.266

The first three rows of this table describe a `/type/object/name` relationship, which defines a human-readable name for a node. Thus, the node we've identified as `/en/the_police` has the name "The Police", and the node `/guid/1234` has the name "Driven to Tears". Notice that these rows have a value in the **To** column as well as the **Value** column, and that the **To** column refers to the node identified as `/lang/en`. That node represents the English language. Human readable text in Metaweb is always tagged with the language in which it is written. Thus any relationship, such as `/type/object/name`, that expects a human-readable text value will refer to a language node in the **To** column and will have a string of text in the **Value** column.

The fourth row in the table specifies that `/en/zenyatta_mondatta` has the relationship `/music/album/artist` with `/en/the_police`. In English, we might say: "Zenyatta Mondatta is by The Police". The fifth row is similar. It specifies a `/music/track/album` relationship between two nodes. It says: "Driven to Tears appears on Zenyatta Mondatta". Finally, the sixth row specifies the `/music/track/length` of Driven to Tears. Note that this relationship is not a link between two nodes, but merely specifies a number in the **Value** column.

[1] There are actually six elements of each tuple: relationships, like nodes, have a timestamp indicating when they were added to the database and also a reference to the user that defined the relationship. We'll learn more about the timestamp and creator of a relationship in §3.7.

2.2. Properties

The **Property** column of our relationship tables have specified relationships with names like `/type/object/name` and `/music/track/album`, and you may have wondered about the fact that these property identifiers look so much like node identifiers that appear in the **From** and **To** columns. One of the key features of Metaweb is that it does not pre-define the kinds of relationships that can exist between nodes. New properties, representing new kinds of relationships, can be defined at any time and by any user. What this means, of course, is that properties are themselves nodes in the Metaweb graph.

Since properties are nodes, they can appear in the **From** column of a table of tuples, and can have relationships themselves. For example:

From	**Property**	**To**	**Value**
/type/object/name	/type/object/name	/lang/en	Name
/music/album/artist	/type/object/name	/lang/en	Artist
/music/track/album	/type/object/name	/lang/en	Appears On
/music/album/artist	/type/property/unique		true
/music/track/album	/type/property/unique		false

The first row in this table is interestingly self-referential. The `/type/object/name` property defines its own name as "Name". The second and third rows define names for the `/music/album/artist` and `/music/track/album` properties, and it is worth noting that the human-readable names of Metaweb nodes are not always the same as the last component of the node identifier.

2.2.1. Properties and Uniqueness

The fourth and fifth rows of the table above specify that `/music/album/artist` is a *unique* property and that `/music/track/album` is not. Because `/music/album/artist` has a `/type/property/unique` property with value `true`, Metaweb will not allow any node to have more than one `/music/album/artist` property. The `/type/property/unique` property is another unique property: no node can have more than one `/type/property/unique` relationship.

On the other hand, `/music/track/album` has `/type/property/unique` set to `false` so Metaweb allows a node to have any number of `/music/track/album` properties (think of songs that are released as singles, then on LPs, and then again on compilation albums). Non-unique properties are common in Metaweb databases, and the group of nodes that are linked to a given node by the same property can be thought of as an unordered set of values.

By default, properties are not unique: that is, a property node that does not have a `/type/property/unique` property is treated as if it had a `/type/property/unique` property with the value `false`.

2.2.2. Properties and Direction

The Metaweb graph is a *directed* graph: each of the relationships has a direction from the **From** node to the **To** node. On a circles-and-lines representation of the nodes and relationships, each of the lines has an arrowhead to indicate the direction of the relationship. Despite the directionality of the links between nodes, Metaweb can traverse those links both forward and backward when searching the database to find results that match a query.

For example, consider the relationship between albums and the tracks that appear on those albums. We saw above this is represented in the Metaweb graph as a directed link *from* the track node *to* the album node. If we ask "what album(s) does Driven to Tears appear on?", the Metaweb database engine searches for tuples that have Driven to Tears in the **From** column and the `/music/track/album` property in the **Property** column, and then reports the value of the **To** column of any matches it finds.

But it is perfectly possible to write a MQL query that asks "what tracks appear on Zenyatta Mondatta?" (we'll see queries like this many times in Chapter 3 and Chapter 4). In this case, the database engine searches for tuples that have Zenyatta Mondatta in the **To** column and `/music/track/album` in the **Property** column and then reports the value of the **From** column for any matches it finds.

Because Metaweb is so good at traversing links in either direction, we can usually consider those links to be bi-directional. In fact, properties can be defined to represent links in either direction. The `/music/track/album` property models the track-to-album link in its forward direction, and it is known as a "master property". But there is also a "reverse property" `/music/album/track` that represents the same relationship, traversed in the opposite direction. In typical music-related queries, this reverse property is more commonly used – it is the one that allows us to ask about the set of tracks that appear on an album.

A master property and its reverse are known as "reciprocal properties", and the distinction between master and reverse does not usually matter in your queries. We'll see more about reciprocal properties in §3.4.4, §3.4.5.4 and §5.9. For now, we'll just note that the relationship between a property node and its reciprocal property node is defined in the Metaweb graph by the `/type/property/reverse_property` property [2]:

From	Property	To	Value
/music/track/album	/type/property/reverse_property	/music/album/track	

2.2.3. Properties and Meaning

Some properties are a rudimentary part of the Metaweb architecture and have a well-defined meaning that is enforced by the Metaweb implementation. `/type/object/name`, for example, defines a human-readable name for a node, and Metaweb enforces an important restriction: nodes can have multiple names, but only one name per language. (We'll learn more about the names of nodes later in this chapter). `/type/property/unique` is another property that is part of Metaweb's architecture: this property defines something fundamental about the behavior of another property,

[2] And that the reverse of /type/property/reverse_property is /type/property/master_property!

and Metaweb's behavior depends its value. In general, if a property id begins with `/type`, that is a good hint that it has architectural significance.

The vast majority of properties are not like this, however. The `/music/album/artist` property, for example, is not part of the Metaweb architecture at all – it was defined on *freebase.com* to model knowledge about music. The Metaweb implementation knows nothing about this property, and its interpretation is ultimately up to the person who is setting or querying its value (or who is writing the application that sets or queries its value). The id `/music/album/artist` gives us a hint about how to interpret the property and that hint is reinforced by the fact that the property has "Artist" as its `/type/object/name`. We can even obtain explicit instructions about the interpretation of `/music/album/artist` by inspecting its `/freebase/documented_object/tip` property, whose value is "albums recorded primarily by this artist (direct credit or under a pseudonym, but not as part of a band)".

Another way to say this is that the *meaning* of `/type/property/unique` is defined by the behavior of the Metaweb implementation, but that the meaning of `/music/album/artist` exists only in the minds of its users. As far as Metaweb is concerned, `/music/album/artist` is just another node in the graph.

2.3. Names, Identifiers and Namespaces

In an example above, we used the identifier `/en/the_police` to refer to a node that has the `/type/object/name` "The Police". This section explains the important differences between names and identifiers.

The `/type/object/name` property defines a human-readable name for a node. The value of the property includes both the text and the language of the name. As we saw above:

From	**Property**	**To**	**Value**
/en/the_police	/type/object/name	/lang/en	The Police

Nodes can have more than one name, but Metaweb enforces an important constraint: a node can have only one name in any given language. (Use `/common/topic/alias` to define any number of nicknames, in any language, for a node.) As an example, the following table shows hypothetical English, French and Spanish names (and aliases) for the `/lang/en` node:

From	**Property**	**To**	**Value**
/lang/en	/type/object/name	/lang/en	English
/lang/en	/type/object/name	/lang/fr	Anglais
/lang/en	/type/object/name	/lang/es	Ingles
/lang/en	/common/topic/alias	/lang/en	American English
/lang/en	/common/topic/alias	/lang/en	British English
/lang/en	/common/topic/alias	/lang/en	Canadian English

Names are not unique and not expected to be: it is common for multiple nodes to have the same name. The nodes with ids `/lang/en`, `/en/english_people`, and `/authority/gnis/57724` represent a language, a nationality, and a town in Arkansas and all have the name "English".

When you ask Metaweb for the name of a node without specifying a language, it returns only the node's name in your preferred language, fostering the convenient illusion that the node has only a single name. Human-readable names exist for the convenience of the human users of Metaweb, and although they are treated specially by the Metaweb architecture, they are not really fundamental to that architecture.

Identifiers, on the other hand, are fundamental. Identifiers consist of three parts: the node that is identified, a namespace and a key within the namespace. The identifier `/en/the_police` identifies a node in the graph that represents the band The Police. It has namespace `/en` and key "the_police". The identifier `/type/object/name` identifies a property node with namespace `/type/object` and key "name". `/type/object` is itself an identifier, with namespace `/type` and key "object". We can also turn this around and say that the namespace `/lang` and key "en" combine to define the identifier `/lang/en`. We'll return to the notion of an identifier as a node, a namespace, and a key in §2.5.9.

A key can be considered a "local name" within a namespace, and the key plus namespace pair can be called a "fully-qualified name" to distinguish it from a "human-readable name". Usually, however, the pair is simply called an identifier or id. As you have seen, Metaweb identifiers are typically written in "flat form" as strings that use a slash character to separate the namespace from key. This means that identifiers look something like Unix filenames or URLs. Identifiers are intended for use by developers and are often used in code, which is why they appear in code font in this manual.

The important thing about identifiers is that they are unique. Metaweb never allows a namespace to contain duplicate keys, which means that two nodes will never have the same identifier. A node can be associated with more than one namespace/key pair, but any given namespace and key can only be used once, and can only refer to a single node. Note that identifiers are not immutable: nodes can be given new identifiers, and identifiers can be altered so that they refer to new nodes.

Identifiers like `/en/the_police` are not human-readable names, but are comprehensible to technically-savvy English-speakers, like the readers of this manual. Many nodes in a Metaweb graph (and most nodes on *freebase.com*) have identifiers of this kind. Those that don't can be referred to using identifiers in the namespace `/guid`. The key that follows `/guid` is a string of hexadecimal digits that serves as a globally-unique identifier (or "guid") for a node. The song "Driven to Tears" from the album Zenyatta Mondatta, for example has the identifier `/guid/9202a8c04000641f800000000129a87a` (we called it `/guid/1234` for brevity earlier in the chapter).

Every node in a Metaweb graph has a numeric guid. The guid uniquely identifies the node and is immutable – the guid of a node can never change. The guid is the fundamental identity of a Metaweb node, and at the implementation level, the tuples that define the Metaweb graph refer to nodes (in the **From**, **Property**, and **To** columns) by their guids, not by the identifiers that we've shown in our tables.

Since the guid is the fundamental identity of a node, the identifier of a node is not fundamental. In fact, identifiers are defined by the `/type/object/key` property, much as names are defined by the `/type/object/name` property. Here, for example, are tuples that define the identifiers `/en/the_police` and `/en/zenyatta_mondatta`.

From	Property	To	Value
/en/the_police	/type/object/key	/en	the_police
/en/zenyatta_mondatta	/type/object/key	/en	zenyatta_mondatta
/en	/type/object/key	/	en

The `/type/object/key` property expects values in both the **To** and **Value** columns. The **To** column is a reference to the namespace of the identifier being defined, and the **Value** column holds the key within the namespace. Namespaces are themselves nodes, and they have identifiers. The third row of the table above shows that the identifier of the `/en` namespace is defined by the "en" key within the special root namespace `/`. For clarity, these example tuples use ids in each of the columns, which means that they define and use an identifier at the same time. A more accurate representation of the underlying graph would use guids in place of these ids.

A proper understanding of names, ids and namespaces is critical to understand Metaweb, and we'll review and explore them in more depth in §3.3 and §5.11.

2.4. Objects and Types

Until now we've been describing the Metaweb database as a graph of nodes and relationships, and have been carefully avoiding the word "object". But that word has appeared many times in property ids like `/type/object/key` and `/freebase/documented_object/tip`. While it is important to understand that at a low-level Metaweb databases consist of tuples that define relationships between nodes, it is usually helpful to view those nodes and relationships at a higher level through an object-oriented filter.

According to this object-oriented view, the nodes in the graph define objects, and the relationships in the graph define properties of those objects. Instead of thinking about the relationship between The Police, Zenyatta Mondatta and Driven to Tears in terms of tuples, we might think of them using pseudo-code [3] like this:

```
{
  id: "/en/the_police",
  name: "The Police",
  /music/artist/album: {
    id: "/en/zenyatta_mondatta",
    name: "Zenyatta Mondatta",
    /music/album/track: {
      name: "Driven to Tears",
      /music/track/length: 200.266
    }
    /music/album/track: {
```

[3] The resemblance to JavaScript object syntax is intentional. The reason will become clear in Chapter 3.

```
      name: "Canary in a Coalmine",
      /music/track/length: 146.506
    }
  }
}
```

In this view of the data, we see an object with id `/en/the_police` and name "The Police". (We're using properties `id` and `name` as shorthand for the "universal" `/type/object/id` and `/type/object/name` properties that are shared by all objects. We'll have more to say about these universal properties below.) That object has an album named "Zenyatta Mondatta", and that album has two tracks (others are omitted here for simplicity) named "Driven to Tears" and "Canary in a Coalmine". Each of those track objects has a length property that specifies its length in seconds. Note that this particular view of the data relies on the properties `/music/artist/album` and `/music/album/track`. These are the reverse of the `/music/album/artist` and `/music/track/album` properties that we saw in the nodes-and-relationships view of the data.

In order to complete the object-oriented view of Metaweb nodes, we have to introduce the notion of *types*. A Metaweb type is a collection of related properties, and a Metaweb object can be an instance of one or more types. `/en/the_police` is an instance of the type `/music/artist`, `/en/zenyatta_mondatta` is an instance of `/music/album`, and the object that represents the track Driven to Tears is an instance of `/music/track`. Like properties, types are represented as nodes in the graph. Every property node is an instance of `/type/property`, and every type node is an instance of `/type/type` (which means that `/type/type` is an instance of itself!) Property objects use the type of which they are a part as the namespace in which their id is defined. So the properties of `/music/track`, such as `/music/track/album` and `/music/track/length`, have ids that begin with `/music/track`.

Types and properties are related via the (unique) `/type/property/schema` property, which defines the relationship between a property and the type of which it is a part. The reverse property `/type/type/properties` represents the (non-unique) relationship between a type and all of its properties.

There are two other, more important, properties that involve types. The first is `/type/object/type`. Like `/type/object/name` and `/type/object/key`, this is a universal property that is used on practically all object in the database. It defines the types (it is non-unique) that an object belongs to. If we ask the Metaweb engine at freebase.com about the types of the `/en/the_police` object, for example, we find that in addition to `/music/artist`, that object is also an instance of `/common/topic`, `/music/producer`, and `/music/musical_group`.

The `/type/object/type` property is important for a couple of reasons. First, the type of an object tells us what properties it is likely to have. For example, if we know that an object is a `/music/artist`, we know it makes sense to ask about the `/music/artist/album` property of the object. Second, the type of an object is a useful disambiguator. If we ask Metaweb for objects named "English", we will likely find many. We can substantially narrow our search by asking for for objects named "English" that are also instances of `/type/lang`.

In addition to `/type/object/type`, there is one other very important type-related property. The (unique) `/type/property/expected_type` property of any property specifies the type of the value of that property. The expected type of the `/music/artist/album` property, for example, is `/music/album`, and the expected type of `/music/album/track` is `/music/track`.

The addition of types to our object-oriented view of the Metaweb graph allows us to simplify our pseudo-code representation of it. Compare this object representation with the one at the beginning of this section:

```
{
  id: "/en/the_police",
  type: "/music/artist",
  name: "The Police",
  album: {
    id: "/en/zenyatta_mondatta",
    name: "Zenyatta Mondatta",
    track: {
      name: "Driven to Tears",
      length: 200.266
    }
    track: {
      name: "Canary in a Coalmine",
      length: 146.506
    }
  }
}
```

We've added a `type` property to the outermost object, specifying that it is a `/music/artist`. This allows us to use the simple property name `album` instead of `/music/artist/album`. Furthermore, since we know that expected type of `/music/artist/album` is `/music/album`, we're now just using the property name `track` instead of `/music/album/track`. For the same reason, we've shortened `/music/track/length` to `length`.

Both the `/type/object/type` and `/type/property/expected_type` properties are a very useful part of the object-oriented view of Metaweb, but they are not fundamental to the nodes-and-relationships view. A node in the graph can have a relationship described by a property `p` even if that node does not have a `/type/object/type` relationship with the type that defines `p`. That is, an object can use a property without "declaring" itself to be a member of the type that defines the property. Properties like `/type/object/name`, for example, are commonly used on objects, but `/type/object/type` is never set to `/type/object`. Similarly, it is perfectly possible (and not uncommon) to define a `/common/topic/alias` property on an object without setting `/type/object/type` to `/common/topic`.

Also, the expected type of a property is only the *expected* type. The open and fluid nature of the Metaweb graph means that Metaweb cannot guarantee that the values of a property will always be members of the expected type.

2.4.1. Universal Properties

`/type/object` is a special type that serves just to define a namespace for a set of special properties that can be used with any object. The properties with ids in the `/type/object` namespace are commonly used in queries on any object and are typically written in MQL queries with unqualified names – we speak of the `name` property and the `key` property, for example, instead of `/type/object/name` and `/type/object/key`. Because of the universality of this type, objects should not have their type set to `/type/object`. Similarly, properties should not have their expected type set to `/type/object`.

The following are the universal properties defined by `/type/object`. The most important ones have already been introduced, but they are all listed here for completeness:

`name`	This property defines human-readable names for the object, suitable for display to the end users. Each name is a `/type/text` value which holds a string and defines the human language in which it is written. The `name` property is special in two ways: • An object may have more than one name, but may only have one name per language. That is, it can have only one English name, only one French name, and so on. • When querying Metaweb, you may treat the `name` property as if it was a single `/type/text` value rather than a set of values. Metaweb will automatically return the object's name (if it has one) in your preferred language. Because of this special feature, the `name` property has `/type/property/unique` set to `true`.
`key`	This property defines identifiers for the object. These identifiers are intended for use by developers and scripts and are not typically displayed to end users. Each `key` property specifies a namespace object and a name within the namespace. Metaweb guarantees that no two objects will ever have the same identifier.
`id`	The `id` property is used to uniquely identify an object using an identifier defined by one of its keys. Identifiers are written as strings with slash characters between namespaces and names. "/type/object" is an `id` value, as are "/en/the_police" and "/user/docs/music/note". If you query the `id` of an object that has more than one key, it is unspecified which one is returned. If you query the id of an object with no keys, the value returned is a synthetic id formed by removing the hash character from the object's `guid` and prefixing it with "/guid/". This property is read-only, but you can define new ids for or alter existing ids of an object with the `key` property.
`type`	This property defines the types associated with the object. The object can be viewed as an instance of any of these types. Each type is itself a Metaweb object, of `/type/type`.
`timestamp`	This unique read-only property specifies when the object was created.
`creator`	This unique read-only property specifies which Metaweb user created the object. It has an expected type of `/type/user`.
`permission`	This unique read-only property is a link to a `/type/permission` object. A permission object specifies which Metaweb usergroups are allowed to alter the object. See §2.7 for more on users, usergroups and permissions.

`guid` Every object in a Metaweb database has a *globally unique identifier* or *guid*. The `guid` property specifies the unique identifier for an object. A guid is a long string of hexadecimal digits following the hash character, and might look like this: `#9202a8c04000641f800000000006df1b`. No two objects will ever have the same value of the `guid` property. This property is read-only, and its use is discouraged: you should usually use the `id` property instead.

2.4.2. Types and Domains

The Metaweb type system does not include an inheritance mechanism. The `/type/object` type is not the supertype of other types; it is simply a set of very general properties that are useful on any object. Although Metaweb types do not form an inheritance hierarchy, they can be categorized as illustrated in Figure 2.1.

```
                      +--/type/id
                      |
                      +--/type/int
                      |
                      +--/type/float
                      |
                      +--/type/boolean
                      |
      +--Value Types--+--/type/text
      |               |
      |               +--/type/rawstring
      |               |                     +--/restaurant domain
      |               +--/type/uri          |
      |               |                     +--/location domain
      |               +--/type/datetime     |
      |               |                     +--/film domain   +-/music/track
Types-+               +--/type/key          |                 |
      |                                     +--/music domain--+-/music/album
      |                                     |                 |
      |               +--Freebase Types-----+--/book domain   +-/music/artist
      |               |                     |
      |               |                     +--etc.
      |               |
      +--Object Types-+--Core Types (/type domain)
                      |
                      +--Common Types (/common domain)
                      |
                      +--User-defined types-+--/user/joe/default_domain
                                            |
                                            +--/user/joe/music
```

Figure 2.1. Categories of Metaweb types

Metaweb defines a small set of *value types* that represent primitives such as numbers, strings, dates and booleans. These value types are described in §2.5. All other types are *object types*. Types are organized into *domains*, which are simply collections of related types. Like properties and types, domains are represented by Metaweb objects and these domain objects serve as the namespaces for the types they include.

Core types that are fundamental to Metaweb are in the `/type` domain. This domain includes the value types plus fundamental object types such as `/type/type` and `/type/property`. Other commonly useful (but not so fundamental) types are part of the `/common` domain. §2.6 describes the object types in the `/type` domain plus the most important types in the `/common` domain. It also discusses a few other domains that contain commonly used types.

In addition to these core and common domains, *freebase.com* defines many other domains, such as `/film`, `/finance` `/government` and `/chemistry` for general knowledge representation. You can browse these domains at *http://www.freebase.com/site/data*, and we'll continue to make heavy use of types from the `/music` domain in Chapter 3 and Chapter 4.

Finally every Metaweb user has a domain in which they can define their own types. (Object types only, however: users cannot define new value types.) If your Metaweb username is "joe", then you have a domain `/user/joe/default_domain`. The *freebase.com* client also allows you to define additional domains in the `/user/joe` namespace. Chapter 5, for example, will ask you to create a personal domain named `/user/joe/music`.

2.5. Value Types

Like many programming languages, Metaweb draws a distinction between objects and primitive values such as numbers, dates and strings. When we view the Metaweb graph as a set of tuples, we see that some tuples have a reference to another node in the **To** column and some have a primitive value in the **Value** column instead. If a property has an expected type that is a value type, then a tuple involving that property will have a value in the **Value** column. On the other hand, if a property has an expected type that is an object type then a tuple involving that property will have no value in the **Value** column.

Metaweb defines nine value types. Like all Metaweb types, value types are identified by type objects such as `/type/int` (for the value type that represents integer values). The sub-sections that follow explain each of the value types in detail. We begin, however, with a short discussion of value types and properties.

2.5.1. Properties of Value Types

In Chapter 3, we'll learn that there are two ways to ask for the value of a property in MQL. Think of the `/music/track/length` property: it represents the duration of a track has an expected type of `/type/float`, which is a value type representing a floating-point number. If we use the first MQL query technique to ask for the length of a particular song, we simply get a single number back. If we use the other technique, MQL will pretend that the value is a simple object with two properties:

`value`	this property holds the primitive value: a number in this case.
`type`	this property specifies the type of the primitive: `/type/float` in this case.

When queried in this way, all value types appear to have these two properties. Keep in mind, however, that these are not true properties: MQL simply allows value types to behave as if they have `value` and `type` properties. These properties are represented by the `/type/value/value` and `/type/value/type` objects. `/type/value` is nominally a type, but is never used as one: like `/type/object` it exists only to group a set of related properties. `/type/object` defines the universal properties of object types, and `/type/value` defines the universal properties (only two of them) of value types.

If a property has an expected type of `/type/text` or `/type/key`, then any tuple involving that property will have values in both the **To** and **Value** column. `/type/text` and `/type/key` are considered value types, but are really something of a hybrid between object types and value types. In addition to the synthetic `value` and `type` properties, each of these types has a real property as well: `/type/text/lang` specifies the human language of a string of text and `/type/key/namespace` specifies the namespace of an id. Further details are in §2.5.5 and §2.5.9.

2.5.2. /type/int

Values of this type are signed integers. Metaweb uses a 64-bit representation internally, which means that the range of valid values of `/type/int` is from -9223372036854775808 to 9223372036854775807. An integer literal is simply an optional minus sign followed by a sequence of decimal digits. Metaweb does not support octal or hexadecimal notation for integers, nor does it allow the use of exponential notation for expressing integers.

2.5.3. /type/float

Values of this type are signed numbers that may include an integer part, a fractional part, and an order of magnitude (a power of ten by which the integer and fractional parts are multiplied.) Metaweb uses the 64-bit IEEE-754 floating point representation which supports magnitudes between 10^{-324} and 10^{308}. C and Java programmers may recognize this as the `double` datatype. Metaweb does not support the special values Infinity and NaN, however.

A literal of `/type/float` consists of an optional minus sign, and optional integer part, and optional decimal point and fractional part and an optional exponent. The integer and fractional parts are simply strings of decimal digits. The exponent begins with the letter e or E, followed by an optional minus sign, and one to three digits. The following are all valid `/type/float` literals:

```
1.0       # integer and fractional part
1         # integer part alone
.0        # fractional part alone
-1        # minus sign allowed as first character
1E-5      # exponent: 1 × 10^-5 or 0.00001
5.98e24   # weight of earth in kg: 5.98 × 10^24
```

There are an infinite number of real numbers, and a 64-bit representation can only describe a finite subset of them. Any number with 12 or fewer significant digits can be stored and retrieved exactly with no loss of precision. Numbers with more than 12 significant digits may have those digits truncated when they are stored in Metaweb.

2.5.4. /type/boolean

There are only two values for this type; they represent the boolean truth values `true` and `false`. Note that Metaweb sometimes uses the absence of a value (`null`) in place of `false`.

2.5.5. /type/text

A value of `/type/text` is a string of text plus a reference to a `/type/lang` object that specifies the human language of that text. The `/type/object/name` property is the most frequently used property of this type.

`/type/text` is unusual. Its `value` property specifies the text itself, but it also has a `lang` property that specifies the language in which the text is written. The `lang` property refers to an object of type `/type/lang`. The `/lang` namespace holds many instances of this type, such as `/lang/en` for English.

The text of a `/type/text` value must be a string of Unicode characters, encoded using the UTF-8 encoding. The encoded string must not occupy more than 4096 bytes. Longer chunks of text (or binary data) can be stored in Metaweb content store.

2.5.6. /type/rawstring

A value of `/type/rawstring` is a string of bytes with no associated language specification. The length of the string must not exceed 4096 bytes.

Use `/type/rawstring` instead of `/type/text` for small amounts of binary data and for textual strings that are not intended to be human readable.

2.5.7. /type/uri

A value of `/type/uri` represents a URI (Uniform Resource Identifier: see RFC 3986). The `value` property holds the URI text, which should consist entirely of ASCII characters. Any non-ASCII characters, and any characters that are not allowed in URIs should be URI-encoded using hexadecimal escapes of the form `%XX` to represent arbitrary bytes.

2.5.8. /type/datetime

An instance of `/type/datetime` represents an instant in time. That instant may be as long as a year or as short as a fraction of a second. The `value` property is a string representation of a date and time formatted according to a subset [4] of the ISO 8601 standard.

A `/type/datetime` value that represents the first millisecond of the 21st century looks like this:

```
2001-01-01T00:00:00.001Z
```

[4] `/type/datetime` only supports dates specified using month and day of month. It does not support the ISO 8601 day-of-year, week-of-year and day-of-week representations.

Notice the following points about this format:

- Longer intervals of time (years, months, etc.) are specified before shorter intervals (minutes, seconds, etc.).
- Years must be specified with a full four digits, even when the leading digits are zeros. Negative years are allowed, but years with more than four digits are not allowed.
- Months and days must always be specified with two digits, starting with 01, even when the first digit is a 0.
- The components of a date are separated from each other with hyphens.
- A date is separated from the time that follows with a capital letter T.
- Times are specified using a 24-hour clock. Midnight is hour 00, not hour 24. Hours and minutes must be specified with two digits, even when the first digit is 0.
- Seconds must be specified with two digits, but may also include a decimal point and a fractional second. Metaweb allows up to 9 digits after the decimal point.
- The hours, minutes, and seconds components of a time specification are separated from each other with colons.
- A time may be followed by a timezone specification. The capital letter Z is special: it specifies that the time is in Universal Time, or UTC (formerly known as GMT). Local timezones that are later than UTC (east of the Greenwich meridian) are expressed as a positive offset of hours and minutes such as `+05:30` for India. Local times earlier than UTC are expressed with a negative offset such as `-08:00` for US Pacific time. If no timezone is specified, then then the `/type/datetime` value is assumed to be a local time in an unknown timezone. Specifying a timezone of `+00:00` is the same as specifying `Z`. Specifying `-00:00` is the same as omitting the timezone altogether.
- All characters used in the `/type/datetime` representation are from the ASCII character set, so date and time values can be treated as strings of 8-bit ASCII characters.

A `/type/datetime` value can represent time at various granularities, and any of the date or time fields on the right-hand side can be omitted to produce a value with a larger granularity. For example, the seconds field can be omitted to specify a day, hour, and minute. Or all the time fields and the day-of-month field can be omitted to specify just a year and a month. Also, the date fields can be omitted to specify a time that is independent of date. A timezone may not be appended to a date alone: there must be at least an hour field specified before a timezone.

Here are some example `/type/datetime` values that demonstrate the allowed formats:

```
2001                   # The year 2001
2001-01                # January 2001
2001-01-01             # January 1st 2001
2001-01-01T01Z         # 1 hour past midnight (UTC), January 1st 2001
2000-12-31T23:59Z      # 1 minute before midnight (UTC) December 31st, 2000
2000-12-31T23:59:59Z   # 1 second before midnight (UTC) December 31st, 2000
```

```
2000-12-31T23:59:59.9Z # .1 second before midnight (UTC) December 31st, 2000
00:00:00Z              # Midnight, UTC
12:15                  # Quarter past noon, local time
17-05:00               # Happy hour, Boston (US Eastern Standard Time)
```

2.5.9. /type/key

Values of `/type/key` define object ids. `/type/key` is the expected type of the `/type/object/key` property and the `/type/namespace/keys` property, and is not intended for use by any other properties.

The `value` property of a `/type/key` value is the local, or unqualified part of an identifier. It must be a string of ASCII characters, and may include letters, numbers, underscores, hyphens and dollar signs. A key may not begin or end with a hyphen or underscore. The dollar sign is special: it must be followed by four hexadecimal digits (using letters A through F, in uppercase), and is used when it is necessary to map Unicode characters into ASCII so that they can be represented in a key. To represent an extended Unicode character (that does not fit in four hexadecimal digits), encode that character in UTF-16 using a surrogate pair, and then express the surrogate pair using two dollar-sign escapes. Each component of an id used for domains, types and properties are further restricted; they may not include two underscores in a row and they may not start with a digit.

Like `/type/text`, `/type/key` has a third property. The `/type/key/namespace` property refers to an object, but the interpretation of that object depends on whether the key is the value of a `/type/object/key` property or a `/type/namespace/keys` property. The `/type/object/key` property defines an id for the object (let's call it `o`) that is the subject of that property. The value of the `key` property of `o` is a `/type/key` value which we'll call `k`. The `namespace` property of `k` is a namespace object `n`. In this case, the id of `n` plus the `value` property of `k` define an id for the original object `o`.

The `/type/namespace/keys` property, on the other hand, defines an identifier within the namespace `n` that is the subject of that property. The value of the `keys` property is a `/type/key` value `k` as before, and the `namespace` property of `k` refers to an object `o`. Here, the id of `n` plus the `value` of `k` define an id for `o`. These two different uses of `/type/key` values are somewhat confusing, but we'll see clarifying examples in §3.3.3 and §5.11.

2.5.10. /type/id

The final value type, `/type/id`, represents an object identifier in "flat form", as a string, rather than as a key in a namespace. `/type/id` exists only to serve as the expected type of the `/type/object/id` and `/type/object/guid` properties. It can not be used by other properties.

2.6. Object Types

This section covers the core object types in the `/type` domain, and also introduces commonly-used types from the `/common` domain and elsewhere. You do not need to understand these types in detail in order to make productive use of Metaweb. Still, knowing what these basic types are is a helpful orientation to the system. You can learn more about these (or any) Metaweb types at *freebase.com*. Learn about a type and its properties by appending the type id to the URL *http://freebase.com/type/schema*. To read more about the type `/type/unit`, for example, visit: *http://freebase.com/type/schema/type/unit* .

2.6.1. Core Types

The `/type` domain defines the value types listed in §2.5 and the other core types that Metaweb depends on. These core types can be loosely divided into categories. The first category is types that define the Metaweb type system. These types have already been introduced, but are summarized here for completeness:

`/type/object`
: This type exists simply to group the properties, such as `/type/object/name` and `/type/object/type` shared by all objects. `/type/object` is *not* a supertype of other types, and objects are never actually typed with `/type/object`.

`/type/type`
: This type describes a type, which means that it is the only type that is an instance of itself. The `properties` property defines the set of properties of the type. The `instance` property defines the set of instances of the type (it is the reverse property of `/type/object/type`). The `domain` property links to the domain that defines the type. The `expected_by` property is the reverse of `/type/property/expected_type` : it is is the set of properties whose values are of this type.

`/type/property`
: This type defines a property. The properties of a property object include `expected_type` which specifies the type of the value of the property and `unique` which specifies whether the property is restricted to a single value. The `schema` property refers to the type object of which the property is a part: it is the reciprocal of `/type/type/properties`. The `unit` property specifies a `/type/unit` value associated with the property: it is useful for properties whose expected type is `/type/int` or `/type/float`. If a property is a master property, then the `reverse_property` property refers to its reciprocal, if one is defined. And if a property is a reverse property, then the `master_property` property refers to the reciprocal. The `enumeration` property of a property has to do with identifiers and namespaces and is explained in §3.3.5. The `requires_permission` property is related to access control and is explained in §5.12.3.

`/type/domain`
: A domain represents a set of related types, and also serves as a namespace for those types. The `types` property (the reciprocal of `/type/type/domain`) specifies the members of the domain. For access control purposes, each domain has an associated usergroup that "owns" the domain.

`/type/namespace`
: This type represents a namespace, and its `keys` property specifies a set of named objects that exist within the namespace. `/type/namespace/keys` is the reciprocal of `/type/object/key`. See §2.5.9 for more about keys and namespaces. Namespaces are more fundamental to the Metaweb architecture than types are, and any object can be used as a namespace, even if it is not typed as a `/type/namespace` object.

Another category of core type are those involved in access control (see §2.7 for more on this topic):

`/type/user`
: Each registered Metaweb user is represented by an object of `/type/user`. User objects have ids in the `/user` namespace. If your username is `joe_developer`, then your `/type/user` object is `/user/joe_developer`. The `usergroup` property of a `/type/user` specifies the usergroups of which the user is a member.

`/type/usergroup`
: This type represents a set of users. The `member` property is the set of users that belong to the group, and the `permitted` property is the set of permissions that are granted to the group.

`/type/permission`
: This type is the key to Metaweb access control. The `permits` property is the set of usergroups that have been granted this permission. And the `controls` property is the (possibly very large) set of object controlled by this permission object. The default permission object is `/boot/all_permission` which allows access by any user.

The following types represent content, and metadata about that content, in the Metaweb content store:

`/type/content`
: Large chunks of content, such as HTML documents and graphical images are not stored in regular Metaweb nodes. Instead, these large binary objects (sometimes called *blobs*) are kept in a separate store. A `/type/content` object is the bridge between the Metaweb object store and the Metaweb content store. A `/type/content` object represents an entry in the content store, and the id of the `/type/content` object is used as an index for retrieving the content.

  In addition to providing access to the content store, `/type/content` defines important properties. The `media_type` property specifies the MIME type of the content. For textual content, the `text_encoding` and `language` properties specify the encoding and language of the text. The `length` property specifies the size (in bytes) of the content. The `source` property refers to zero or more `/type/content_import` objects that specify the source of the content. (If the same content is uploaded multiple times, it may have multiple sources.)

  Chapter 4 shows how to download content from Metaweb, and Chapter 6 demonstrates how to upload content.

`/type/content_import`
: This type describes the source of imported content. Its properties include the URI or filename from which the content was obtained, the user who imported the content, and a timestamp that specifies when the content was imported.

`/type/media_type`
: Instances of this type represent a MIME media type such as "text/html" or "image/png". Instances are given fully-qualified names within the `/media_type` namespace, and can be specified with ids like `/media_type/text/html` or `/media_type/image/png`.

`/type/text_encoding`
: Instances of this type represent standard text encodings, such as ASCII and Unicode UTF-8. Instances are given fully-qualified names within the `/media_type/text_encoding` namespace, and can be specified with ids such as `/media_type/text_encoding/ascii`.

Finally, the `/type` domain also includes a few miscellaneous types:

`/type/lang`
: This type represents a human language. Instances, such as `/lang/en` which represents English, exist in the `/lang` namespace. The `iso639` property specifies the two-letter code (such as "en") of the language. `/type/text` values have an associated language, as do `/type/content` objects that represent text.

`/type/unit`
: This type defines a measurement unit, and is the expected type of `/type/property/unit`. A number of instances, such as `/en/meter`, `/en/kilogram`, and `/en/second`, are defined. This type has no properties.

`/type/enumeration`
: This type is used as the expected type of any property that defines an enumeration. See §3.3.5 for details.

`/type/link`
: This special type allows us to view the links in the Metaweb graph as objects. It is used in advanced MQL queries as explained in §3.7.

`/type/reflect`
: This is not a true type, but a collection of special properties used in reflective queries. See §3.7.3.

2.6.2. Common Types

The types in the `/common` domain are not a core part of the Metaweb infrastructure, but, as the name implies, they are commonly useful, and some of them are quite important for the *freebase.com* client. The five most commonly-used `/common` types are:

`/common/topic`
: Metaweb objects that are intended for display to end users through *freebase.com* are called "topics". Such objects typically have some appropriate domain-specific type, such as `/music/artist` or `/food/restaurant`, but are also typically instances of the type `/common/topic`. This type defines properties that allow documents, images, webpages, and nicknames to be associated with the topic. This type is so common that properties like `/common/topic/image` and `/common/topic/alias` are sometimes used without actually adding `/common/topic` to the set of types of an object.

`/common/document`
: This type represents a document of some sort. `/common/topic` uses this type to associate documents with topics. The most important property is `content`, which specifies the single `/type/content` object that refers to the document content. Other properties of `/common/document` provide meta-information about the document, such as authors, publication date, and so on.

`/common/image`
: `/type/content` objects that represent images are typically co-typed with this type. `/common/image` defines a `size` property that specifies the pixel dimensions of the image.

`/common/webpage`
: This type is simply the URL of a webpage plus a short description of the page's content.

`/common/phone_number`
: This type has two properties of `/type/rawstring` to hold a phone number and a county code for the phone number.

To view the complete list of types in the `/common` domain visit *http://freebase.com/view/common*. In general, you can browse the contents of a domain by appending the domain id to the URL *http://freebase.com/view*.

The `/type` and `/common` domain are not the only ones that include commonly-used types. There are a few others that you should be aware of:

`/freebase`
: This domain defines types used by the *freebase.com* client. Many of them are quite implementation-specific and not of general interest to applications other than Freebase itself. `/freebase/documented_object` allows short tips and longer-form documentation to be associated with any other Metaweb object, such as domains, types and properties. `/freebase/user_profile` instances hold information about Freebase users.

`/location`
: The `/location` domain contains types related to geographical locations. These include `/location/country`, `/location/administrative_division` (as well as country-specific versions such as `/location/us_state`), `/location/citytown`, `/location/postal_code` and `/location/mailing_address`. The reason that the mailing address type is not in `/common` along with `/common/phone_number` is that it depends on other `/location` types for representing countries, cities, and so on.

`/people`
: This domain defines the important type `/people/person` as well as related types such as `/people/gender` and `/people/marriage`.

`/time`
: This domain defines time and date related types beyond the simple `/type/datetime` primitive. Types include `/time/day_of_week`, `/time/month`, and `/time/day_of_year`.

`/measurement_unit`

This domain defines units of measure. The `/type/unit` type we saw earlier marks an object as a unit so that it can be used with `/type/property/unit`. But it is this `/measurement_unit` domain that provides detailed types to represent units. The `/en/second` object, for example is both a `/type/unit` and a `/measurement_unit/unit_of_time`.

The unit types in this domain are not actually commonly used. The more useful types are the *compound value types*: these are types that define two or more properties (often of primitive type) so that multiple values can be manipulated together as a single value. For example `/measurement_unit/time_interval` has two `/type/datetime` properties and is used to represent the starting and ending point of a period of time. `/measurement_unit/integer_range` is similar, but has properties of type `/type/int` instead. `/measurement_unit/money_value` combines a `/type/float` with a `/finance/currency` property. And `/measurement_unit/dated_money_value` combines those two properties with a `/type/datetime`, tying the amount of money to a specific date (which is useful when dealing with inflation and time-varying currency exchange rates, for example).

2.7. Access Control

Metaweb is completely open for reading. Anyone who can connect to Metaweb servers can read data from them. When adding or editing data, however access control comes into play. We've already seen that the types `/type/user`, `/type/usergroup`, and `/type/permission` are used for access control.

Metaweb's access control model is quite simple. Every object has a `permission` property that refers to a `/type/permission` object. The permission object specifies a set of usergroups whose members have permission to modify the object. If a user is a member of one or more of the specified groups, then that user can edit [5] the object. Otherwise, the user is not allowed to.

This simple access control model is, by default, completely open. In order to allow and encourage free collaboration most Metaweb objects have a permission object that gives edit permission to all Metaweb users. If Metaweb user Fred creates a new object, his friend Jill can freely edit that object, and any other Metaweb user can edit the object as well. (In Chapter 6 we'll learn how to alter the default permission to create objects with restricted access.) The use of Metaweb types is also completely open: any user can create instances of any type, regardless of who "owns" the type.

Although most objects can be freely edited and all types can be freely used, Metaweb namespaces are not usually wide open like this. System namespaces like `/lang` and `/type` are owned by Metaweb administrators, and regular users cannot add keys to them. The most common user-defined namespaces are domains, which serve as the namespace for types, and types, which serve as the namespace for properties. When you create a new domain, a usergroup and permission object are created with it, and the `permission` property of the domain is set so that only members of the usergroup can define keys in the domain. This means that only members of the group can define types in the domain. Newly created types use the same permission object as their domain, which means that only users in the usergroup can define properties in the namespace of the type.

[5] The precise meaning of "edit" is a little complicated, and there is also another form of access control known as per-property access control. Details are in §5.12.

The *freebase.com* client allows users to create personal domains, and to edit the membership of the associated usergroup. This allows Fred to create a new domain and then add Jill and other collaborators to the usergroup so that they can create and modify types within the domain.

We'll explore the topic of access control in more detail in §5.12 after we have learned how to express MQL write queries.

3

The Metaweb Query Language

This chapter explains the Metaweb Query Language, or MQL, [1] which is used to express Metaweb queries. This chapter begins with an explanation of the JSON data format, on which the MQL grammar is based. That prerequisite material is followed by an extended tutorial that teaches MQL by example. You are expected and encouraged to run queries and to experiment with your own queries, using a "query editor" program like the one at *http://www.freebase.com/tools/queryeditor* to submit your queries to Metaweb and obtain the results.

This chapter teaches you to write MQL queries, but does not explain how to issue those queries to and retrieve responses from Metaweb servers: that is the topic of Chapter 4. Also, this chapter does not cover updates, or writes, to Metaweb. Updates are expressed using a variant of MQL that is covered in Chapter 5.

3.1. JavaScript Object Notation

The Metaweb queries and responses we saw in Chapter 1 contained a lot of punctuation: curly braces, quotation marks, colons, and commas. Before we study more queries, it is important to understand this punctuation. Metaweb queries and responses use a plain-text data interchange format known as *JavaScript Object Notation* or, more commonly, JSON. If you are a JavaScript programmer, then this format will be familiar to you since it is a subset of the JavaScript language. [2] If you are not a JavaScript programmer, the format is easy-to-learn, and does not require the use of the JavaScript language.

JSON is formally described in RFC 4627 (*http://www.ietf.org/rfc/rfc4627.txt*), and is also documented at *http://json.org*. The JSON website includes pointers to code, in a variety of programming languages, for serializing data structures into JSON format and for parsing JSON text into data structures. [3]

A JSON-formatted string is a serialized form of an array or object. The array or object may contain numbers, strings, other arrays and objects, and the literal values `null`, `true`, and `false`. These JSON values are illustrated in Figure 3.1 and explained in the sub-sections that follow:

[1] MQL is pronounced "mickle". It rhymes with "pickle", not "sequel".
[2] You should read this section even if you already know JavaScript. JSON is only a subset of JavaScript, and its syntax is stricter than JavaScript syntax.
[3] The JSON syntax diagrams that appear below are also from the JSON website, where they have been placed in the public domain.

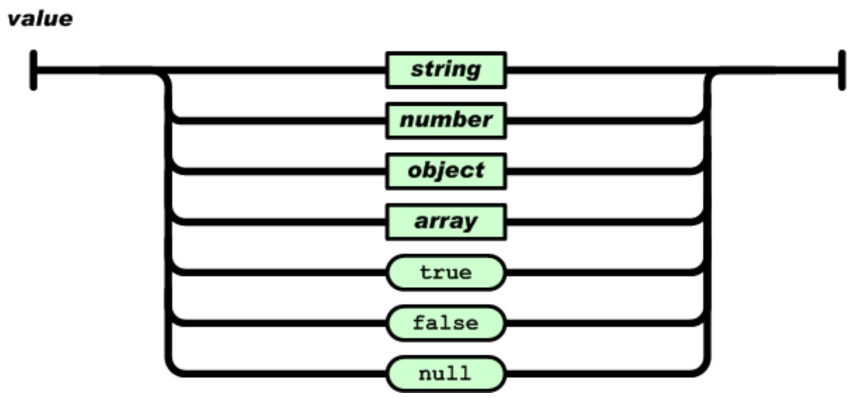


Figure 3.1. JSON Values

3.1.1. JSON Literals: null, true, false

JSON supports three literal values. `null` is a JSON value representing "no value". The literals `true` and `false` a represent the two possible Boolean values.

3.1.2. JSON Numbers

A JSON number consists of an optional minus sign followed by an integer part followed by an optional decimal point and fractional part followed by an optional exponent. This format is the same as the format described for `/type/float` in Chapter 2. All numbers use decimal digits: octal and hexadecimal notation are not supported.

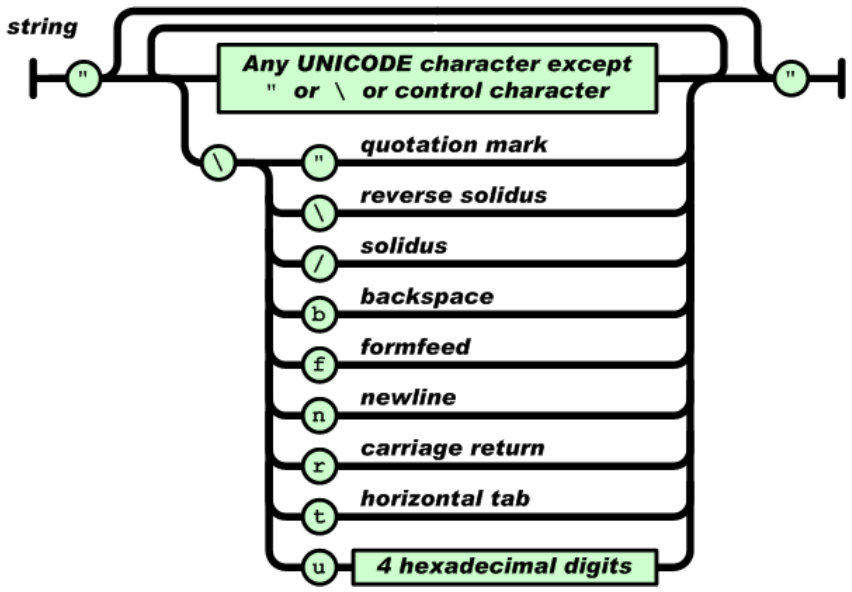


Figure 3.2. JSON string syntax

3.1.3. JSON Strings

A JSON string is much like a string in Java or JavaScript: zero or more Unicode characters [4] between double quotation marks. See Figure 3.2.

A backslash is special: it is an escape character and is interpreted along with the character or characters that follow:

Escape	Character
`\"`	A quotation mark that does not terminate the string
`\\`	A single backslash character that is not an escape
`\/`	A forward slash character. Although it is legal to escape the forward slash character, it is never necessary to do so.
`\b`	The Backspace character
`\f`	The Formfeed character
`\n`	The Newline character
`\r`	The Carriage Return character
`\t`	The Tab character
`\u`*`XXXX`*	The Unicode character whose encoding is the four hexadecimal digits *`XXXX`*. To encode extended Unicode codepoints that do not fit in four hex digits, use two `\u` escapes to encode a UTF-16 surrogate pair.

3.1.4. JSON Arrays

An array is a comma-separated list of JSON values enclosed in square brackets. See Figure 3.3

Arrays may contain any JSON values, including objects and other arrays. The elements of a JSON array need not have the same type (though in MQL they always do). The following JSON array might be returned in response to a MQL query:

```
["Outlandos d'Amour", "Reggatta de Blanc", "Zenyatta Mondatta"]
```

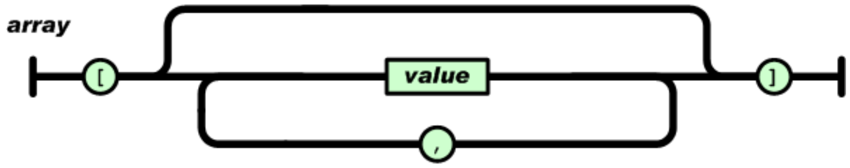


Figure 3.3. JSON array syntax

A JSON array with no elements consists of just the square brackets: `[]`. Empty arrays often appear in MQL queries.

[4] JSON itself supports 32-bit, 16-bit and 8-bit encodings of Unicode text. Metaweb, however, requires the 8-bit UTF-8 encoding.

3.1.5. JSON Objects

A JSON object is named after the JavaScript object type, and is not very much like the objects of strongly-typed object-oriented programming languages. Instead, think of an object as:

- an associative array;
- a hashtable that maps strings to values;
- a dictionary; or
- an unordered set of named values.

JSON objects are written as a comma-separated list of name/value pairs, enclosed in curly braces. A name/value pair is a JSON string (the name) followed by a colon followed by any JSON value, which may include nested objects and arrays. See Figure 3.4

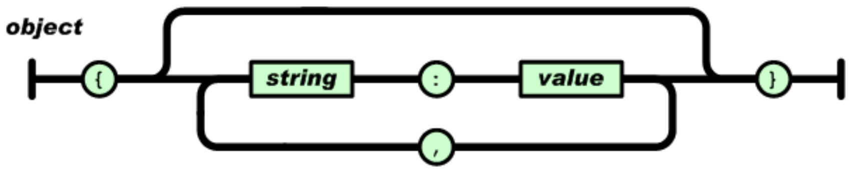


Figure 3.4. JSON object syntax

Here is an example JSON object (which also happens to be a Metaweb query):

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : []
}
```

JavaScript programmers should note that JSON requires property names to appear within double quotes, even though the JavaScript language does not. Arbitrary whitespace is allowed within JSON objects and arrays, but trailing commas (after the final array element or last name/value pair) are not. An empty JSON object, with no properties at all is simply a pair of curly braces: `{}`. As we'll see, empty objects are not uncommon in MQL queries.

3.2. Basic MQL Queries

This section is a tutorial that teaches basic Metaweb queries by example, and uses *freebase.com* as a source of interesting data to query. Try to follow along as you read it by trying out the queries presented. To do this, you need a simple way to submit a query to *freebase.com* and view the result. You can do this with the Freebase query editor at *http://www.freebase.com/tools/queryeditor/*.

3.2.1. Our First Query

Let's begin by revisiting the simple query from Chapter 1. We would like to know what albums The Police have recorded:

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : []
}
```

When we run this query, we get the following JSON object in response (some of the album names are omitted here for brevity):

```
{
  "type": "/music/artist",
  "name": "The Police",
  "album": [
    "Outlandos d'Amour",
    "Reggatta de Blanc",
    "Zenyatta Mondatta",
    "Ghost in the Machine",
    "Synchronicity",
  ]
}
```

To query Metaweb we tell it what we already know by specifying properties and their values:

```
    "type" : "/music/artist",
    "name" : "The Police",
```

And then we tell it what we want to know by specifying properties without values:

```
    "album" : []
```

Sending an empty array in a MQL query tells Metaweb that we'd like to have the array filled in.

Singular or Plural?

Note that the property we query in the example above is named "album" and not "albums", even though bands like The Police may well have more than one album. This reflects the underlying nature of Metaweb: the database object that represents The Police has many links of type "album" that refer to the objects that represent those albums. The type `/music/artist` aggregates these many links into a single set of albums, but retains the singular name "album" because that is the name of the underlying link type.

Also, as we'll see soon when you want to obtain information (such as a list of tracks) about one particular album, you specify a single value (instead of an array) for the `album` property. In this case, the singular name makes a lot of sense.

Although property names are typically singular, there are exceptions to this rule, and sometimes you'll see a plural property name.

3.2.2. Query/Response Symmetry

Let's look one more time at the simple "albums by The Police" query and response from above. This time the query and response are presented side-by-side to emphasize that the query and response objects have the same properties, but the response object has values filled in:

Query	Result
`{ "type" : "/music/artist", "name" : "The Police", "album" : [] }`	`{ "type": "/music/artist", "name": "The Police", "album": [ "Outlandos d'Amour", "Reggatta de Blanc", "Zenyatta Mondatta", "Ghost in the Machine", "Synchronicity" ] }`

```
{                               {
  "type" : "/music/artist",       "type": "/music/artist",
  "name" : "The Police",          "name": "The Police",
  "album" : []                    "album": [
}                                   "Outlandos d'Amour",
                                    "Reggatta de Blanc",
                                    "Zenyatta Mondatta",
                                    "Ghost in the Machine",
                                    "Synchronicity"
                                  ]
                                }
```

This symmetry of queries and responses is a fundamental and elegant part of MQL. We'll use this two-column query/response format throughout the chapter.

3.2.3. Object IDs

Objects can be given fully-qualified names that can be used as unique identifiers. Query the `id` property of an object to obtain a unique identifier for it:

Query	Result
`{`	`{`
`  "type" : "/music/artist",`	`  "type": "/music/artist",`
`  "name" : "The Police",`	`  "name": "The Police",`
`  "id" : null`	`  "id": "/en/the_police"`
`}`	`}`

This query includes the same name and type as the last query. But instead of specifying an empty array of albums, it specifies a `null` id. The `null` value is our query: this is the field we want Metaweb to fill in. The response looks just like the query, but the null has been replaced with a unique fully-qualified name for The Police.

In addition to querying the `id` property of an object, we can also use it to uniquely name the object we want. We could rewrite our musical albums query to use `id` instead of `name`, for example:

```
{
  "type": "/music/artist",
  "id": "/en/the_police",
  "album": []
}
```

Metaweb objects are not required to have a fully-qualified name, but every object is assigned a string of hexadecimal digits that serves as a globally unique identifier or *guid*. If you query the `id` property of an object that does not have a fully-qualified name, Metaweb creates a synthetic identifier by prefixing the object's guid with `/guid/`:

Query	Result
`{`	`{`
`  "type" : "/music/album",`	`  "type": "/music/album",`
`  "artist": "The Police",`	`  "artist": "The Police",`
`  "name" : "Synchronicity",`	`  "name": "Synchronicity",`
`  "id" : null`	`  "id": "/guid/1f8000000002f9e349"`
`}`	`}`

You can use guid-based identifiers as the value of the `id` property to uniquely identify an object.

Freebase GUIDs in this Tutorial

The guids shown in the online HTML-formatted version of this tutorial are valid *freebase.com* guids, and should remain valid in perpetuity. In hardcopy and PDF versions of the tutorial, guids have been shortened to allow queries and responses to fit side-by-side in the two-column format shown above. To use these printed guids online, restore them to validity by inserting `9202a8c0400064` at the start.

3.2.4. Multiple Results and Uniqueness Errors

Now that we know the id of the object representing The Police, let's turn our query around and ask about name and type of the object with that id:

```
{
  "id": "/en/the_police",
  "name" : null,
  "type" : null
}
```

We're telling Metaweb what we have (the id) and asking for the values (name and type) that we don't have. When we submit this query, though, it doesn't work. The response envelope looks like this:

```
{
  "status": "200 OK",
  "code": "/api/status/ok",
  "qname": {
    "code": "/api/status/error",
    "messages": [
      {
        "code": "/api/status/error/mql/result",
        "message": "Unique query may have at most one result. Got 4",
        "info": {
          "count": 4,
          "result": [
            "/music/artist",
            "/common/topic",
            "/music/producer",
            "/music/musical_group"
          ]
        },
        "query": [
          {
            "error_inside": "type",
            "type": null,
            "id": "/en/the_police",
            "name": null
          }
        ],
        "path": "type"
      }
    ]
  }
}
```

This response includes three properties named `code`. The outermost one indicates that the query was well-formed and could be processed. The middle `code` property is specific to our query (which we gave the name `qname`) and specifies that an error occurred. The `messages[0]` object provides details, including a more specific error code in the innermost `code` property. The `message` property of the message object is a human-readable version of this error code, and the `info` property provides further details about the error. The `query` property repeats the query that caused the error, and both the `query.error_inside` and the `path` properties indicate which part of the query caused the problem.

What we learn from this response is that Metaweb could not respond to our query because we asked for a single type and it found four types. Let's try the query again. Now we're requesting a single name and an array of types for this uniquely specified object. This query works:

Query	Result

```
{
  "id": "/en/the_police",
  "name" : null,
  "type" : []
}
```
	```
	{
	  "id": "/en/the_police",
	  "name": "The Police",
	  "type": [
	    "/music/artist",
	    "/common/topic",
	    "/music/producer",
	    "/music/musical_group"
	  ]
	}
	```

The Metaweb object we asked about has the name "The Police" and it is a member of four types including `/common/topic` and `/music/artist`. Recall from Chapter 2 that `/common/topic` is a very generic type. Just about every Metaweb object that represents something an end user would have an interest in is a member of this type. The lesson to draw here is that objects almost always have more than one type, and any queries on the `type` property should use arrays. In general, it is always safe to use `[]` in place of `null` in your queries. If there is only one result the array returned in the response will simply have a single element. When you know that there can only be one result, however, it is usually more convenient and efficient to use `null`.

Single Values and Sets of Values

There is a fundamental asymmetry to MQL: when we query the type of an object, we get an array of types. But when we look up an object by type, we specify only one type. Metaweb objects have a set of types, not one single type. So when we specify the type of an object in a MQL query, all we are saying is that the object has at least one "type" link with that value. Thus, writing `"type":"/music/artist"` in a query does not say "the type *is* /music/artist", but "the set of types *includes* /music/artist". Put another way, we can say that a query provides constraints, and that the response provides values for the unconstrained properties of the query.

Uniqueness errors are a common pitfall for developers crafting Metaweb queries. Recall that `/type/property` allows certain properties to be specified as unique. The `id` and `name` properties are unique and can always be queried without square brackets. As we've seen, however, the `type` property is not unique: objects can (and many objects do) have more than one type. If a property is not guaranteed to be unique, then you should always use square brackets when querying its value.

Although objects can have more than one fully-qualified name, queries of the `id` property never return more than one of these names. The `id` property is unique in another, more important, way: no two objects ever share the same id. Therefore, if a query includes an `id`, you can be confident that no more than one object will match. Therefore, a query like this one is correct:

```
{
  "id": "/en/the_police",
  "name" : null,
  "type" : []
}
```

The `name` property is unique, [5] so it is always safe to query `name` with `null`, as we do above, rather than `[]`. On the other hand, the query that we started this tutorial with is risky:

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : []
}
```

This query worked for us: Freebase only knows about one musical artist named "The Police". Note, however, that there is no guarantee that this will always be the case. There is nothing to prevent someone from adding another band named "The Police" to *freebase.com*. If such an addition were made, our query would suddenly fail.

Depending on the design of your application, a uniqueness failure in this situation might actually be exactly what you want. If you get two results when you expected one, then perhaps the right thing to do is fail and display an error message to the user. In practice, however, most MQL programmers simply get in the habit of enclosing all queries in square brackets:

```
[{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : []
}]
```

When you write queries like this, you must be prepared to handle zero, one, or multiple results.

[5] Objects may actually have more than one name, but may only have one name in any given language. For this reason, simple name queries only return one value. We'll see more about this later in the chapter.

3.2.5. Nested Sub-queries

Let's find out more about our favorite band. What are the names of the tracks on the album *Synchronicity*?

Query

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : {
    "name" : "Synchronicity",
    "track" : []
  }
}
```

Result

```
{
  "type": "/music/artist",
  "name": "The Police",
  "album": {
    "name": "Synchronicity",
    "track": [
      "Synchronicity II",
      "Every Breath You Take",
      "King of Pain",
      "Wrapped Around Your Finger",
      "Tea in the Sahara",
      "Walking in Your Footsteps",
      "Miss Gradenko",
      "Murder by Numbers",
      "O My God",
      "Synchronicity I",
      "Mother"
    ]
  }
}
```

The interesting thing about this query is that it includes a sub-query nested inside curly braces. We're asking for an array of tracks from an album named "Synchronicity" recorded by a band named "The Police".

Now consider the following query which includes a sub-query within a sub-query. It asks: "what artists have recorded the song *Too Much Information*? How long is the song and what album does it appear on?". Some of the results have been omitted here.

Query

```
[{
  "type":"/music/artist",
  "name":null,
  "album": [{
    "name":null,
    "track": [{
      "name":"Too Much Information",
      "length": null
    }]
  }]
}]
```

Result

```
[{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : [{
    "name" : "Ghost in the Machine",
    "track" : [{
      "name" : "Too Much Information",
      "length" : 222.733
    }]
  },{
    "name" : "Message in a Box (disc 3)",
    "track" : [{
      "name" : "Too Much Information",
```

Query	Result

```
    "length" : 222.733
  }]
 }]
},{
  "type" : "/music/artist",
  "name" : "Duran Duran",
  "album" : [{
    "name" : "Duran Duran",
    "track" : [{
      "name" : "Too Much Information",
      "length" : 296.573
    }]
  }]
},{
  "type" : "/music/artist",
  "name" : "Quiet Riot",
  "album" : [{
    "name" : "Alive and Well",
    "track" : [{
      "name" : "Too Much Information",
      "length" : 268
    }]
  }]
}]
```

Let's take a closer look at this query. It involves three different objects: an artist, an album, and a track. We can't tell Metaweb anything interesting about the album (such as a name or id): just that it contains the song we're interested in. We can't tell Metaweb anything about the artist object either: just that they recorded an album that includes the song. Despite the seeming vagueness of this query, Metaweb has no trouble finding the answer we want.

At first glance, it seems as if the only information we're providing to Metaweb with this query is the track name. But the structure of the query contains additional implicit information. We've specified that the outermost object is a `/music/artist`. The definition of this type tells us that objects connected via the `album` property are expected to be of type `/music/album`. And the definition of the `/music/album` type tells us that objects connected through the `track` property should be of type `/music/track`. These additional constraints give Metaweb enough information to find the information we want.

3.2.6. Inverting Queries

With MQL, there is usually more than one way to write a query. This is especially true when a query contains sub-queries – because of the bi-directional nature of Metaweb links, queries can usually be turned "inside out". At the beginning of the last section we wrote a query to ask for the names of tracks on the album Synchronicity. That was an artist-centric query with `"type":"/music/artist"` in the outermost query. We can invert the query and get the same information with a track-centric query:

```
[{
  "type":"/music/track",
  "name":null,
  "album":{
    "name":"Synchronicity",
    "artist":"The Police"
  }
}]
```

This query returns the same list of track names, but the results are much more verbose, since every track object includes a nested. In this case, the simplest way to obtain the list of tracks we want is probably with a non-nested album-centric query:

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : []
}
```

The previous section included another artist-centric query to asked about recordings of the song "Too Much Information". Here's the album-centric version of that query:

```
[{
  "type":"/music/album",
  "name":null,
  "artist":null,
  "track": [{
    "name":"Too Much Information",
    "length": null
  }]
}]
```

We can also invert this query into a track-centric form, of course:

```
[{
  "type":"/music/track",
  "name":"Too Much Information",
  "length":null,
  "album":[{"name":null, "artist":null}]
}]
```

The track, album, and artist-centric versions of this query all return the same basic information. Which version is best depends on how you intend to use the results. Often this will be a question of which result format most closely matches the data structures used by the application that is making the query.

3.2.7. Asking Metaweb For Objects

In our queries so far, we've used `null` and `[]` to ask Metaweb to fill in a single value or an array of values. There are other ways to ask for information as well. Recall the following query:

```
{
  "id" : "/en/the_police",
  "name" : null,
  "type" : []
}
```

It asks for the name and types of a unique object. Both the name, and the individual elements of the `type` array are returned as strings. Recall, however, that the name of an object is of `/type/text` and that types are of `/type/type`. `/type/text` is a value type in the Metaweb object model, but we can treat values as objects if we want to. Let's modify the query to use `{}` and `[{}]` instead of `null` and `[]`. `{}` asks for a single value, expanded as an object, and `[{}]` asks for an array of values expanded into objects:

Query

```
{
  "id": "/en/the_police",
  "name" : {},
  "type" : [{}]
}
```

Result

```
{
  "id" : "/en/the_police",
  "name" : {
    "lang" : "/lang/en",
    "type" : "/type/text",
    "value" : "The Police"
  },
  "type" : [{
    "id" : "/music/artist",
    "name" : "Musical Artist",
    "type" : ["/type/type","/freebase/type_profile"]
  },{
    "id" : "/common/topic",
    "name" : "Topic",
    "type" : ["/type/type","/freebase/type_profile"]
  },{
    "id" : "/music/producer",
    "name" : "Record Producer",
    "type" : ["/type/type","/freebase/type_profile"]
  },{
    "id" : "/music/musical_group",
    "name" : "Musical Group",
    "type" : ["/type/type","/freebase/type_profile"]
  }]
}
```

We learn from this query that the English name of the specified object is "The Police". And, in addition to obtaining the ids of the four types of which The Police is a member, we also obtain the human-readable names of those types.

Let's use this query technique to learn more about the tracks on the album Synchronicity. (The result lists only two tracks for brevity.)

Query

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : [{}]
}
```

Result

```
{
  "type":  "/music/album",
  "name": "Synchronicity",
  "artist": "The Police",
  "track" : [{
    "id" : "/guid/1f800000000120b4ca",
    "name" : "Synchronicity II",
    "type" : [
      "/music/track",
      "/music/song",
      "/music/composition"
    ]
  },{
    "id" : "/guid/1f8000000001275dd7",
    "name" : "King of Pain",
    "type" : ["/music/track"]
  }]
}
```

This query doesn't actually tell us much about the tracks themselves. We already know the type of the tracks. The id might be useful in future queries, but it doesn't tell us anything about the track. The name is useful, but we could have obtained that without using curly braces, just by querying `"track":[]`.

When you ask Metaweb to fill in empty curly braces for you, it returns all the properties if the value is a value type. The `name` property of an object is of `/type/text`, and querying it with `{}` returns all of its properties. If the property is an object type instead of a value type, then Metaweb returns only the `name`, `type` and `id` properties (all of which are defined by `/type/object` and are common to all Metaweb objects). That is, instead of using `[{}]`, we could write out the query explicitly like this:

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : [{
    "name" : null,
    "id" : null,
    "type" : []
  }]
}
```

If we want to know more about an object than its name, id, and types, then we must refine our query to express exactly what it is we would like to know. Here's how we ask for just the name and length of each of the tracks:

Query

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : [{
     "name":null,
     "length":null
  }]
}
```

Result

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : [
    {"name":"Synchronicity II", "length":305.066},
    {"name":"Every Breath You Take", "length":254.066},
    {"name":"King of Pain", "length":299.066},
    {"name":"Wrapped Around Your Finger", "length":313.733},
    {"name":"Tea in the Sahara", "length":255.44},
    {"name":"Walking in Your Footsteps", "length":216.773},
    {"name":"Miss Gradenko", "length":120},
    {"name":"Murder by Numbers", "length":276.8},
    {"name":"O My God", "length":242.226},
    {"name":"Synchronicity I", "length":202.866},
    {"name":"Mother", "length":185.64}
  ]
}
```

3.2.8. Expanded Values and Default Properties

In this tutorial we've said that we query the value of a property `p` with `"p":null` and "expand" that value into an object with `"p":{}`. This is helpful terminology, but it is actually the opposite of what is really going on. Everything in Metaweb is an object (or, in the case of literal values, can be viewed as an object). When you use curly braces, objects are naturally expressed as objects. When you use `null`, however, objects are compressed: instead of returning the complete object, Metaweb returns only one property – called the default property – of the object. The query `"p":null` really means "look up the expected type of the property `p`, then look up the default property of that type and return the value of that default property of the object that `p` refers to".

If the expected type of a property is a primitive type, then the default property is `value`. If the expected type is not a system type (i.e. if it is not in the `/type` domain) then the default property is `name`. If the expected type is a system type, then the default property depends on the type but is usually `id` instead of `name`.

Default properties are not only used when you ask Metaweb to fill in a `null` or a `[]` for you. They are also used when you express the information you already have. Consider the following query:

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : []
}
```

This query could also be expressed more verbosely like this:

```
{
  "type" : "/music/album",
  "name" : {"value":"Synchronicity", "lang":"/lang/en"},
  "artist" : {"name":"The Police"},
  "track" : []
}
```

The verbose form of the query illustrates the fact that the succinct form relies on default properties. The `name` property is expected to be of `/type/text`, whose default property is `value`. The `artist` property is expected to be of type `/music/artist`, whose default property is `name`.

3.2.9. Review: Asking for Values

If you want to ask Metaweb to return a value, use one of the terms listed in Table 3.1 on the right-hand side of a property name:

Table 3.1. Asking for Values

Term	Meaning
`null`	If the property is of value type, return the `value` property. Otherwise, the property refers to an object. If the object is of a system type in the `/type` domain, return the value of its default property (this is usually the `id` property). Otherwise, return the `name` of the object.
`[]`	Like `null`, but return an array of values instead of a single value.
`{}`	If the property is of value type, return an object that represents the value. This object will have `type` and `value` properties. If the property is `/type/text`, the returned object will also have a `lang` property, and if it is of `/type/key`, it will have a `namespace` property. If the property is of object type, return an object that includes its `name`, `id`, and `type` properties. In this case, the term `{}` is equivalent to: `{"name":null,"id":null,"type":[]}`.
`[{}]`	Like `{}`, but return an array of objects instead of a single one.

3.3. Names, Ids, Keys and Namespaces

Metaweb defines a few different ways to name objects. This section begins with a review of names, ids, guids, keys and namespaces, repeating some of the material from §2.3. After this overview, it demonstrates, with queries, many of the important features of names, ids, and namespaces.

Every Metaweb object has a `name` property which can be used to specify or query a human-readable name (such as "The Police") for the object. Names are of `/type/text` and have a language (of `/type/lang`) associated with them. An object may have more than one name, but is only allowed to have one name per language. The `name` property of an object usually behaves, therefore, as if it has a single value. Names do not uniquely identify single objects: multiple objects may (and often do) share the same name.

In addition to its human-readable name or names, every Metaweb object has zero or more fully-qualified names or identifiers. These are hierarchical names that use the `/` character in the way filesystems do. They are intended for use by Metaweb developers and are not typically displayed to end-users of Metaweb-based applications. Fully-qualified names are of `/type/id`, and are values of the `id` property of an object. `/music/album` is a fully-qualified name, and so is `/en/the_police`. A fully-qualified name uniquely identifies a single object, but is not immutable: fully-qualified names can be deleted or re-assigned to other objects. Use the `id` property to query or specify the fully-qualified name of an object.

An object may have more than one fully-qualified name. In a query, you can specify an object by giving any of its fully-qualified names as the value of the `id` property. If you query the `id` of an object that has more than one fully-qualified name, the name that is returned is arbitrary.

Every object in a Metaweb database has exactly one globally unique identifier or *guid*. Guids are machine-readable hexadecimal numbers 32 digits (128 bits) long. They serve as universally-unique identifiers – no two objects (even objects retrieved from different Metaweb databases) will ever have the same guid. If you query the `id` property of an object that does not have any fully-qualified names, Metaweb will return a pseudo-id based on the object's guid. For example:

```
/guid/1f8000000000006df1b
```

The fully-qualified names of an object are defined by its keys. These are the `/type/key` values of the `key` property. Each key consists of an unqualified name (the `value` of the key) and a namespace (the `namespace` of the key). The namespace is another Metaweb object, and it is usually referred to by *its* fully-qualified name. The fully-qualified name `/en/the_police`, for example, has a value (or unqualified name) of "the_police" and a namespace whose id is `/en`. That namespace object has a key whose value is "en" and whose namespace is the special root namespace object whose id is `/`.

A namespace may not contain two keys with the same name – this is another way of saying that a fully-qualified name must uniquely refer to a single object. A namespace may normally define two distinct names that refer to the same object. It is possible, however, to define "unique namespaces" in which there is a one-to-one mapping between names and objects: each name refers to a single object, and each object has only a single name (in that namespace).

Working with keys can be confusing, and there is a shortcut that is sometimes useful. Some types define properties that are known as enumerations. The definition of an enumeration specifies a namespace, and the value of the property serves as a name within that namespace. Setting the value of an enumerated property of an object defines a fully-qualified name for the object.

We'll see examples of names, ids, keys and enumerations in the sub-sections that follow.

3.3.1. Names and Per-Language Uniqueness

The `name` property of any Metaweb object is special because it behaves like a unique property (you can safely query it with `null` instead of `[]`, for example) even though it is not truly unique. Any Metaweb object can have multiple names, but may have only one name in any given language. That is, the `name` property is unique on a per-language basis. When you query the name of an object, Metaweb returns its name (if it has one) in your preferred language. (The desired language is specified as a parameter to the *mqlread* query service: see §4.2.4.3.)

To demonstrate the special behavior of the `name` property, we must choose a topic that has translations into other languages. Let's ask about the name of the object with `id` "/en/united_states":

```
Query                        Result
{                            {
  "id":"/en/united_states",    "id":"/en/united_states",
  "name":null                  "name":"United States"
}                            }
```

The "en" in the namespace of the id stands for English, so it is not surprising that the English name of this object matches the id. Now let's ask for more details about the name:

```
Query                        Result
{                            {
  "id":"/en/united_states",    "id":"/en/united_states",
  "name":{}                    "name" : {
}                                "type" : "/type/text",
                                 "value" : "United States",
                                 "lang" : "/lang/en"
                               }
                             }
```

This confirms that the name "United States" is an English name. Now let's ask for all names of the object:

```
Query                        Result
{                            {
  "id":"/en/united_states",    "id":"/en/united_states",
  "name":[]                    "name":["United States"]
}                            }
```

This query just returns the unique English name in an array. So let's try again and ask for all names, along with the languages in which they are encoded:

Query

```
{
  "id":"/en/united_states",
  "name":[{}]
}
```

Result

```
{
  "id":"/en/united_states",
  "name" : [
    {"lang":"/lang/en","type":"/type/text",
     "value":"United States"},
    {"lang":"/lang/es","type":"/type/text",
     "value":"Estados Unidos de América"},
    {"lang":"/lang/fr","type":"/type/text",
     "value":"États-Unis d'Amérique"},
    {"lang":"/lang/it","type":"/type/text",
     "value":"Stati Uniti d'America"},
    {"lang":"/lang/de","type":"/type/text",
     "value":" Vereinigte Staaten"},
  ]
}
```

Bingo! We find that this object has a number of names (only some of which are listed here).

Here's how we can ask for a name of the object in a specific language other than our preferred language:

Query

```
{
  "id":"/en/united_states",
  "name":{
    "value":null,
    "lang":"/lang/fr"
  }
}
```

Result

```
{
  "id":"/en/united_states",
  "name": {
    "value": "États-Unis d'Amérique",
    "lang": "/lang/fr"
  }
}
```

The default preferred language (and the one used throughout this chapter) is English. We'll learn how to specify a different language in §4.2.4.3.

3.3.2. Ids

The most important thing about the `id` property is that if you specify its value, you are uniquely identifying a single object in the database. A query that specifies `id` need not have square brackets around it: it can never return more than one value. It is also a good rule of thumb (though not always strictly necessary) to put square brackets around any query that does not specify the `id` property.

It is always safe to query the id of an object with `"id":null`. This query will always return exactly one value (even if the object has more than one fully-qualified name). If the object does not have any fully-qualified names, Metaweb returns a pseudo-id using the object's guid and the `/guid` namespace. For example:

Query	Result
`{`	`{`
`  "type" : "/music/album",`	`  "type" : "/music/album",`
`  "artist": "The Police",`	`  "artist" : "The Police",`
`  "name" : "Synchronicity",`	`  "name" : "Synchronicity",`
`  "id" : null`	`  "id" : "/guid/1f8000000002f9e349"`
`}`	`}`

There is a one-to-one mapping between objects and guids: every object has one guid, and every valid guid must refer to a different object. Ids are less strict: no two objects can have the same id, but an object can have zero, one, or more fully-qualified names. We can refer to the Metaweb object that represents the band The Police with any of the following:

```
"id":"/guid/1f800000000006df1b"
"id":"/en/the_police"
"id":"/wikipedia/en_id/57321"
"id":"/wikipedia/en/Police_band"
```

There are some restrictions on what you can do with the `id` property: it cannot be used as a sort key, and it cannot be used with operators such as ~= and < (We'll learn about sorting and operators later in this chapter.)

Also, you cannot query all the fully-qualified names of an object with `"id":[]`:

Query	Result
`{`	`{`
`  "type" : "/music/artist",`	`  "type": "/music/artist",`
`  "name" : "The Police",`	`  "name": "The Police",`
`  "id" : []`	`  "id": ["/en/the_police"]`
`}`	`}`

`"id":[]` returns a single valid id for the object, just as `"id":null` does, but it wraps it in square brackets. To find multiple fully-qualified names of an object, you must query its keys, which are the topic of the next section.

3.3.3. Keys and Namespaces

To ask for multiple fully-qualified names for an object, query its `key` property. The following query asks for all properties of all keys of The Police.

Query

```
{
  "id":"/en/the_police",
  "key":[{}]
}
```

Result

```
{
  "id" : "/en/the_police",
  "key" : [{
    "type" : "/type/key",
    "namespace" : "/en",
    "value" : "the_police"
  },{
    "type" : "/type/key",
    "namespace" : "/wikipedia/en_id",
    "value" : "57321"
  },{
    "type" : "/type/key",
    "namespace" : "/wikipedia/en",
    "value" : "Police_band"
  },{
    "type" : "/type/key",
    "namespace" : "/wikipedia/en",
    "value" : "The_Police_$0028band$0029"
  }]
}
```

The results of this query have been truncated, but there are two things worth noting about the representative keys shown here. First, fully-qualified names can contain numbers and underscores, but they cannot contain punctuation. If a local name contains punctuation such as parentheses, these must be escaped using Unicode codepoints. For example, $0028 in a fully-qualified name represents a left parenthesis and $0029 represents a right parenthesis. (See §2.5.9 for the full list of legal characters in fully-qualified names.)

Second, note that these keys do not include a key in the `/guid` namespace. `/guid` ids are synthesized by Metaweb when no fully-qualified name exists: they do not represent a real key.

The query above returns keys representing multiple fully-qualified names for The Police. Those results are not necessarily a complete list, however. The keys refer to namespaces, and the namespaces themselves can have more than one fully-qualified name. Consider the `/en` namespace:

```
Query          Result
{              {
  "id":"/en",    "id" : "/en",
  "key":[{}]     "key" : [{
}                  "type" : "/type/key",
                   "namespace" : "/topic",
                   "value" : "en"
                 },{
                   "type" : "/type/key",
                   "namespace" : "/",
                   "value" : "en"
                 }]
               }
```

This query tells us that our namespace object has the name "en" in the root namespace `/`, but that it also has the name "en" in the namespace `/topic`. This means that the fully-qualified name `/en/the_police` can also be written as `/topic/en/the_police`.

The keys of an object all refer to namespace objects in which those keys are defined. Since Metaweb links are bi-directional, it must also be possible to query a namespace to find out what names are defined in it. Here's a query on the `/topic` namespace. It queries the `key` property we've already seen to ask for the fully-qualified names of this object. But it also queries the `keys` property to find out what keys are defined in `/topic`. (The `key` property is defined by `/type/object`. The `keys` property is defined by `/type/namespace`, and is plural to avoid naming conflicts with `key`.):

```
Query                            Result
{                                {
  "type":"/type/namespace",        "type" : "/type/namespace",
  "id":"/topic",                   "id" : "/topic",
  "key":[{}],                      "key" : [{
  "keys":[{}]                        "namespace" : "/",
}                                    "type" : "/type/key",
                                     "value" : "topic"
                                     }],
                                   "keys" : [{
                                     "namespace" : "/en",
                                     "type" : "/type/key",
                                     "value" : "en"
                                   }]
                                 }
```

The results tell us that the object `/topic` has only the one fully-qualified name we already know: the local name "topic" in the namespace `/`. They also tell us that there is only a single key defined in the `/topic` namespace. This key has the local name "en", which means that it defines the fully-qualified name `/topic/en`.

Both the key and keys properties return values of /type/key. The namespace properties of these keys have different meanings, however. When you query the key property of an object, the namespace property of each returned key refers to the namespace object in which the key is defined. When you query the keys property of a namespace, however, the namespace property of each returned key specifies the object whose name is defined by that key. So the query above tells us that the /topic namespace defines a key named "en" that refers to an object with id /en. That is /topic/en is another fully-qualified name for /en.

Finally, it is worth noting that any Metaweb object can serve as a namespace, even if it does not have a type property of /type/namespace. Types (such as /music/artist) have ids that use the domain (/music) as a namespace, and properties (such as /music/artist/album have ids that use the type as a namespace. At the time of this writing, [6] Metaweb domains and types very often do not have /type/namespace among their types. Despite these examples, it is not recommended practice to use an object as a namespace without typing it as a namespace.

3.3.4. Namespaces and Uniqueness

A namespace can define any number of keys, but the value property of each key must be different. Otherwise, the same fully-qualified name would be defined twice, and could refer to more than one object. On the other hand, the reverse is not necessarily true: normal namespaces can define multiple keys that refer to the same object. For example, we've already seen that the /wikipedia/en namespace defines "Police_band" and "The_Police_$0028band$0029" as names for the same object.

Not all namespaces allow multiple names for the same object, however. A namespace may be declared to be "unique", and unique namespaces enforce a one-to-one mapping between names and objects: each name can refer to only one object and each object can have only one name. The /user and /lang namespaces are unique, which means that each user and language can have only a single fully-qualified name (assuming that /user and /lang don't have any other ids themselves). Most other namespaces, such as /en are not unique. Unique namespaces have the unique property of /type/namespace set to true. Namespaces that are not unique typically have null for this property.

Query	Result
`{ "id":"/lang", "type":"/type/namespace", "unique":null }`	`{ "id" : "/lang", "type" : "/type/namespace", "unique" : true }`

[6]September, 2008

3.3.5. Enumerations

An *enumeration* is MQL's mechanism for connecting a type with the namespace that defines names for objects of that type. An enumeration takes the form of a property whose expected type is `/type/enumeration`. If a type has a property like this, then setting the value of that property on an object defines a key in the associated namespace. The property value is the name of the key, and the key refers to the object on which the property was set. Similarly, if you create a name in the namespace that refers to an object, then that object automatically gets a value for the property.

The type `/type/lang` is associated with the namespace `/lang` through the enumerated property `iso639`. (ISO639 is the name of an international standard defining short codes such as "en" and "fr" for language names.) This means that for any object of `/type/lang`, the value of the `iso639` property is also a used to define a fully-qualified name in the the `/lang` namespace. The following query (and partial set of results) demonstrates:

Query

```
[{
  "type":"/type/lang",
  "name":null,
  "id":null,
  "iso639":null
}]
```

Result

```
[{
  "type" : "/type/lang",
  "name":"English",
  "id" : "/lang/en",
  "iso639" : "en"
},{
  "type" : "/type/lang",
  "name":"German",
  "id" : "/lang/de",
  "iso639" : "de"
},{
  "type" : "/type/lang",
  "name":"Spanish",
  "id" : "/lang/es",
  "iso639" : "es"
},{
  "type" : "/type/lang",
  "name":"French",
  "id" : "/lang/fr",
  "iso639" : "fr"
}]
```

`/type/lang` defines the `iso639` property so that its value becomes part of the fully-qualified name of the object. In a sense, then, this `iso639` property is the "unqualified name" or "local id" of every `/type/lang` object. So language objects have a human-readable `name`, a fully-qualified, hierarchical `id`, and this local name, the values of which are defined (or "enumerated") by international standard ISO639.

Let's investigate the `iso639` property itself:

Query | **Result**

```
{                             {
  "type":"/type/property",      "type" : "/type/property",
  "id":"/type/lang/iso639",     "id" : "/type/lang/iso639",
  "expected_type":null,         "expected_type" : "/type/enumeration",
  "enumeration":null,           "enumeration" : "/lang",
  "unique":null                 "unique" : true
}                             }
```

We see from this query that the property has expected type `/type/enumeration`, and that it has a property named `enumeration` that refers to associated namespace. Since we already know that the `/lang` namespace is unique (no language can have more than one name in the namespace), it follows that this `iso639` property must also be defined as a unique property: it cannot make sense to have more than one value for the property.

The `userid` property of `/type/user` is an important enumeration that links `/type/user` objects with the `/user` namespace. Another notable example involves taxonomy. The type `/biology/organism_classification` has two enumerated properties: the `itis_tsn` property links to the `/biology/itis` namespace and the `ncbi_taxon_id` links to the `/biology/ncbi` namespace. For example, the Metaweb object representing the species *Equus caballus* (horses) has its `itis_tsn` property set to "180691" and its `ncbi_taxon_id` set to "9796" and can be referred to as: `/biology/itis/180691` or `/biology/ncbi/9796`. For this type, neither the enumerated properties nor the namespaces they refer to are marked unique, which means that either the ITIS or NCBI classification scheme may allow multiple names for the same species.

It should be clear from these examples that MQL enumerations are not really the same thing as the "enumerated types" (a type that has a small, pre-defined set of allowed values) that are supported by some programming languages. The name "enumeration" refers to the fact that the namespace associated with the enumerated property enumerates instances of the type. As the examples above have shown, enumerations are most useful when there is an external authority (such as an international standard) that defines the names.

Recall that the names defined in namespaces are restricted to using letters, numbers, and underscores. Any other characters must be written as a dollar sign followed by the four hexadecimal digits of its Unicode codepoint. MQL does not impose this restriction on enumerated properties. Instead, it transparently escapes and un-escapes names as needed. That is, if you set the value of an enumerated property to a string that contains a punctuation character, that character will automatically be escaped in the namespace. Similarly, if a key contains escaped punctuation but you read it through an enumerated property, the escapes will be replaced by the characters they represent. This can't be demonstrated using the `/lang`, `/user` or `/biology` namespaces we've mentioned here, since none of them define names that include punctuation. But we'll see an example in Chapter 5 when we define our enumerated properties for types of our own.

3.3.6. GUIDs

If you query the `id` of an object, Metaweb returns one of its fully-qualified names, if it has any. If it has none, Metaweb returns a synthetic identifier based on the guid of the object. Fully-qualified names and guids both uniquely identify a single object, but there is an important difference between them. Fully-qualified names are defined by keys that are separate from the object itself, and these keys are mutable. A fully-qualified name may be deleted and refer to no object at all. Or it may be modified so that it refers to an new object. If you run a query that identifies an object by its fully-qualified name, that query may (though this is unlikely) refer to a different object each time you run it, or it may fail because the fully-qualified name now refers to no object at all.

Guids, on the other hand, are intrinsic to the object and are persistent and immutable: a guid is assigned when an object is created, and it always refers to that object. If you want to be sure that a query always refers to exactly the same object, use the object's guid. If an object has a fully-qualified name, you cannot obtain its guid by querying the `id` property. In this case, query the `guid` property:

Query	Result
`{`	`{`
`  "type" : "/music/artist",`	`  "type" : "/music/artist",`
`  "name": "The Police",`	`  "name" : "The Police",`
`  "guid" : null`	`  "guid" : "#1f800000000006df1b"`
`}`	`}`

For historical reasons, the value returned by the `guid` property has a leading `#` character. You can use this value, with its leading hash, as the value of the `guid` property, or you can replace the `#` with `/guid/` and use the modified string as the value of the `id` property. In either case, you'll always refer to the original object, even if the fully-qualified names that used to refer to that object now refer to other objects.

Applications that use guids tend to be more brittle and less resilient to database changes than applications that use fully-qualified names, so the use of the `guid` property is generally discouraged.

3.4. Property Names in MQL Queries

So far in this chapter, we've seen simple property names to the left of the colon in MQL queries. This section explores property names in more depth, explaining how to use qualified property names, property name prefixes, and property name wildcards. In addition to property names, this section also explores `/type/property` objects themselves. Later in this chapter, we'll learn about directives and operators. Directives are special MQL commands that appear to the left of the colon in place of a property name. And operators are punctuation that are added on to the end of the property name.

3.4.1. Simple and Qualified Property Names

Recall from the beginning of this tutorial that most objects in Metaweb have two or more types:

Query	Result

`{`
`  "id":"/en/the_police",`
`  "name":null,`
`  "type":[]`
`}`
	`{`
	`  "id" : "/en/the_police",`
	`  "name" : "The Police",`
	`  "type" : [`
	`    "/music/artist",`
	`    "/common/topic",`
	`    "/music/producer",`
	`    "/music/musical_group"`
	`  ]`
	`}`

What do you do if you want to query one property, such as a list of albums from one type, and another property, such as a list of images, from a second type? MQL addresses this issue by allowing you to specify a fully-qualified property name that includes the name of the type to which it belongs. So here is how we ask for the albums, tracks and pictures by and of The Police:

```
1 {
2   "type":"/music/artist",
3   "name":"The Police",
4   "album":[{
5      "name":null
6      "track":[]
7   }],
8   "/common/topic/image":[{}]
9 }
```

Line 2 specifies that the object to be matched should be of type `/music/artist`. Line 3 specifies the name of the object. `name` and `type` are properties of `/type/object`, and are shared by all objects in the database. These property names (along with `id`, `key`, etc.) can always be used without qualification (although you can qualify them with `/type/object` if you want to). Other types are not allowed to define properties whose names conflict with these.

Line 4 asks for a property named `album`. This property is not defined by `/type/object`, but it is defined by `/music/artist`, and the query has already declared that the object will be an instance of that type, so MQL allows us to use this unqualified property name. Line 5 is like lines 2 and 3: it names a property shared by all objects. Line 6 is an interesting case. `track` is a property of `/music/album`. We can use it here without qualification because the `album` property to which this sub-query is attached was declared to have an "expected type" of /music/album. MQL knows this and assumes that the unqualified property name `track` means `/music/album/track`.

Finally, line 8 asks for a property named `image`. This is not defined by `/type/object` nor by `/music/artist`, and so we must qualify it with the name of its type so that Metaweb can understand it.

For symmetry, and to be explicit, you can rewrite the query (dropping the track portion) to fully-qualify both properties of interest:

```
{
  "type":"/music/artist",
  "name":"The Police",
  "/music/artist/album":[],
  "/common/topic/image":[{}]
}
```

If you do this, you might be tempted to drop the initial `type` specification, since the `album` property is now fully-qualified:

```
[{
  "name":"The Police",
  "/music/artist/album":[],
  "/common/topic/image":[{}]
}]
```

This is probably not the query you want, however: it returns any object whose name is The Police, even if it has no `album` or `image` properties, and even if it is an instance of neither `/music/artist` nor `/common/topic`.

In addition to querying properties from two different types, there is another reason you might need to use fully-qualified names: you might want to query the value of a property without constraining the results to members of the type that defines the property. It may seem surprising, but the Metaweb architecture allows objects to have values for any property, even if the object does not declare itself to be a member of that type. Metaweb type objects are an example: they serve as a namespace for the properties they define, (the `album` property of `/music/artist` is `/music/artist/album`) but are not typically typed as namespaces (the set of types for `/music/artist` does not include `/type/namespace`). It is possible to query the namespace keys of an object, even if that object is not a namespace with a query like this:

```
{
  "id":"/music/album",
  "/type/namespace/keys":[{}]
}
```

Without fully-qualified property names, we'd have to write this:

```
{
  "id":"/music/album",
  "type":"/type/namespace",
  "keys":[{}]
}
```

But this query would simply return `null` because the object with an `id` of `/music/album` does not have `/type/namespace` in its set of types.

3.4.2. Property Prefixes

Suppose we want to find the names of all bands who have an album named "Greatest Hits" AND an album named "Super Hits". We might try this query:

```
[{
  "type":"/music/artist",
  "name":null,
  "album":["Greatest Hits","Super Hits"]  // Invalid MQL
}]
```

But this is not legal MQL. And if it was, it would probably mean find an artist who has recorded exactly two albums, with names "Greatest Hits" and "Super Hits". A musical artist object may have multiple `album` links to album objects. We want to constrain our query so that all result objects have links to two specific album names. Here's a natural way to express this query:

```
[{
  "type":"/music/artist",
  "name":null,
  "album":"Greatest Hits",
  "album":"Super Hits"      // Invalid JSON
}]
```

This query makes sense in the Metaweb object model: find objects that have one "album" link to an album named "Greatest Hits" and another "album" link to an album named "Super Hits". Unfortunately, this query is not valid JSON: it includes the same property name twice, which means that cannot be parsed into object form. (To put this another way, you could not represent this query in a dictionary or hash data structure in a programming language like Python, Ruby or JavaScript.)

MQL's solution to this dilemma is to allow an arbitrary identifier and colon to prefix any property name. The prefix and colon are ignored: they serve simply as a workaround to the JSON limitation just described. With this trick we can rewrite the query above like this:

Query

```
[{
  "type":"/music/artist",
  "name":null,
  "a:album":"Greatest Hits",
  "b:album":"Super Hits"
}]
```

Result

```
[{
  "type": "/music/artist",
  "name": "Alice Cooper",
  "a:album": "Greatest Hits",
  "b:album": "Super Hits"
},{
  "type": "/music/artist",
  "name": "Dan Fogelberg",
  "a:album": "Greatest Hits",
  "b:album": "Super Hits"
}]
```

Note that the arbitrary prefixes we choose for the query are repeated in the result objects. (The results shown here are truncated, of course.) The prefixes are arbitrary, but they must be valid identifiers which means they cannot contain punctuation characters and must not begin with a digit.

Another use of property prefixes is to constrain a property and also query the property at the same time. Here's how we can identify an object by name and type, and also ask for its full set of types:

Query

```
{
  "constraint:type":"/music/artist",
  "name":"The Police",
  "query:type":[]
}
```

Result

```
{
  "constraint:type" : "/music/artist",
  "name" : "The Police",
  "query:type" : [
    "/music/artist",
    "/common/topic",
    "/music/producer",
    "/music/musical_group"
  ]
}
```

Note that although property prefixes are arbitrary, we can choose identifiers like "constraint" and "query" that add meaning to our queries. In practice, it is common to see "a", "b", and "c" used as prefixes.

Here is a more complex example that uses multiple prefixes to constrain and query properties at the same time. It asks for albums by solo artists (objects that are both `/music/artist` and `/people/person`) who have released Greatest Hits and Super Hits albums:

```
[{
  "type":"/music/artist",
  "also:type":"/people/person",
  "name":null,
  "album":[],
  "includes:album":"Greatest Hits",
  "and:album":"Super Hits"
}]
```

Suppose that for symmetry we wanted to use a prefix before both of the type constraints in the query above, so that they looked like this:

```
  "primary:type":"/music/artist",
  "secondary:type":"/people/person"
```

If we do this, the query fails with the message "Type /type/object does not have property album". If we put a prefix in front of the `type` property, MQL will not automatically search the specified type for properties. To make the query work with prefixed type properties, we must fully-qualify the `album` properties like this:

```
  "/music/artist/album":[],
  "includes:/music/artist/album":"Greatest Hits",
  "and:/music/artist/album":"Super Hits"
```

As an interesting aside, let's return to the query with which we started this section. We want to find bands that have released "Greatest Hits" and "Super Hits" albums. There is actually a way to do this without property prefixes. It relies on the fact that Metaweb relationships are always bi-directional and that MQL queries can be "turned inside out":

```
[{
  "type":"/music/artist",
  "name":null,
  "album":[{
    "name":"Greatest Hits",
    "artist":{
      "album":"Super Hits"
    }
  }]
}]
```

Translated into English, this query says: "give me the names of all bands that have released an album named "Greatest Hits", the artist of which has released an album named "Super Hits". The `album` property of a band object refers to an album object. And the `artist` property of the album object refers back to the band object. We can use this fact to further constrain the artist. This technique is worth understanding because it illustrates one of the deep properties of Metaweb objects and MQL queries.

3.4.3. Wildcards

MQL allows you to use the property name "*" as a wildcard. Consider the following query:

Query

```
{
  "id":"/guid/1f8000000002f9e349",
  "*":null
}
```

Result

```
{
  "id":"/guid/1f8000000002f9e349",
  "guid" : "#1f8000000002f9e349",
  "name" : "Synchronicity",
  "type" : ["/music/album", "/common/topic"],
  "key" : ["RELEASE3178", "196871"],
  "creator" : "/user/mwcl_musicbrainz",
  "permission" : "/boot/all_permission",
  "timestamp" : "2006-12-10T12:23:59.0119Z"
}
```

This query identifies a unique object by guid, and then uses a wildcard to ask for all of its properties. Since no type has been specified, the wildcard is expanded with all the properties of `/type/object`, and the result is as shown above.

Note that some of the properties expand to a single value, and others to arrays. Thus the syntax `"*":null` really means `"*":null-or-[]`. We could instead write the query using `"*":[]`. In this case, all of the property are returned as arrays, even unique properties.

Now let's modify the query to specify a type other than the default of `/type/object`:

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "*":null
}
```

In this query, the `*` wildcard expands differently. Since we have specified that the object is of type `/music/album`, Metaweb looks up the properties of that type and queries each one with a `null` or `[]`, depending on whether the property is unique or not. It does this in addition to also querying the common object properties shown in the query result above.

Note that if a property is explicitly listed in a query, a wildcard expansion will not overwrite it. Consider this:

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track":[{}],
  "*":null
}
```

This query explicitly asks for an array of tracks, as objects rather than just as track names. The expansion of the wildcard would normally include `"track":[]`, but in this case that property would conflict with the explicitly specified one and will be left out of the expansion.

Metaweb will expand wildcards in sub-queries based on the type inferred from the expected type of the property. In this query, the `*` expands to `/type/object` and `/music/track` properties:

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track":[{"*":null}]
}
```

Now consider this query:

```
[{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":{
    "type":"/common/topic",
    "*":null
  }
}]
```

Here the artist sub-query is expected to match an object of type `/music/artist`. But we've explicitly specified the type `/common/topic`, so in this case the wildcard expands to the properties of `/type/object` and `/common/topic`. The properties of `/music/artst` are not included in the expansion.

Instead of writing a wildcard query as `"*":null`, you can also use another, more aggressive, form. `"*":{}` expands to query each property with `{}` or `[{}]` instead of `null` or `[]`. Similarly, `"*":[{}]` expands to query each property, even unique properties, with `[{}]`. Contrast the following queries, each of which returns more information than the one before it:

```
// List the names of albums by The Police
{
  "id":"/en/the_police",
  "/music/artist/album":[]
}

// List the names, types and ids of albums by The Police
{
  "id":"/en/the_police",
  "/music/artist/album":[{}]
}

// List all object and album properties of all albums by The Police
{
  "id":"/en/the_police",
  "/music/artist/album":[{"*":null}]
}

// Expand all object and album properties of all albums by The Police
{
  "id":"/en/the_police",
  "/music/artist/album":[{"*":{}}]
}
```

A property name wildcard must be followed by `null`, `[]`, `{}` or `[{}]`. This means that wildcard queries cannot be nested. We cannot take the last query above another step and write:

```
{
  "id":"/en/the_police",
  "/music/artist/album":[{"*":{"*":null}}]  // Illegal
}
```

The `*` property name wildcard queries the value of all properties based on the specified or inferred type of an object, and it queries those properties whether or not they have ever had a value assigned to them. MQL has another wildcard-like feature that allows us to query the value of all the properties of an object that have been defined, regardless of the type that defines those properties. This "reflective" capability is described in §3.7.3.

3.4.4. Inverting a Property with !

If you put an exclamation point before the name of a property, you are asking MQL to run the query using the reciprocal of the property you have named. To put this another way, a MQL property specifies a link between two objects, and the ! prefix asks MQL to follow that link backwards. The following two queries, for example, are equivalent:

```
// Return albums by the police
{
  "id" : "/en/the_police",
  "/music/artist/album" : []
}

// Return albums for which the artist property refers to this object
{
  "id" : "/en/the_police",
  "!/music/album/artist" : []
}
```

All Metaweb links are bi-directional, but not all properties have a reciprocal defined. The ! prefix is useful in these cases where you want to use a reverse property that has not been defined. The type `/people/person` defines a `nationality` property, with an expected type of `/location/country`, for example. But `/location/country` does not (or did not when this was written) define a reciprocal `citizen` property. So to ask Freebase for a list of citizens of Monaco, we could write:

Query

```
{
  "type" : "/location/country",
  "name" : "Monaco",
  "!/people/person/nationality" : []
}
```

Result

```
{
  "type" : "/location/country",
  "name" : "Monaco",
  "!/people/person/nationality" : [
    "Olivier Beretta",
    "Louis Chiron",
    "Sebastien Gattuso",
    "Armand Forcherio",
    "Torben Joneleit",
    "Sophiane Baghdad",
    "Manuel Vallaurio"
  ]
}
```

We could, of course have written an equivalent query like this:

```
[{
  "type" : "/people/person",
  "name" : null,
  "nationality" : "Monaco"
}]
```

This query returns the same set of names, but the result is much more verbose, with `type` and `nationality` properties repeated unnecessarily for each person returned.

3.4.5. Property Objects

As explained in §2.6.1, the properties of a Metaweb type are represented by objects of `/type/property`. Let's use a wildcard to query all the properties of the `album` property of the `/music/artist` type:

Query

```
{
  "id":"/music/artist/album",
  "type":"/type/property",
  "*":null
}
```

Result

```
{
  "id" : "/music/artist/album",
  "type" : "/type/property",
  "name" : "Albums",
  "key" : [ "album" ],
  "expected_type" : "/music/album",
  "schema" : "/music/artist",
  "unique" : null,
  "master_property" : "/music/album/artist",
  "reverse_property" : null,
  "delegated" : null,
  "enumeration" : null,
  "links" : [],
  "unit" : null
}
```

The sub-sections that follow explain the results of this query in detail. (For simplicity, some `/type/object` properties that are not relevant here have been omitted from these results).

3.4.5.1. Property Names and Keys

The property we queried above has the name "Albums" and the key `album`. This key is a name defined in the `/music/artist` namespace, giving the property the fully-qualified name `/music/artist/album`. It is important to understand that when we refer to a "property name" in discussions of MQL, we almost always mean its fully-qualified name, or the unqualified name defined by its key. The name "Albums" appears in the Freebase client when you explore the `/music/artist` type, but it is never used in MQL queries. When writing MQL, we use the key: `album`.

3.4.5.2. Property Type and Schema

Every `/type/property` object has two types associated with it. `expected_type` is the type that the property value is expected to have. (This is called an "expected" type because Metaweb does not enforce it. MQL makes the assumption that property values match their expected type, but in practice any property can refer to an object of any type.) The `schema` is the type of which the property is a part. So the `/music/artist/album` property we queried above has an `expected_type` of `/music/album` and a `schema` of `/music/artist`.

3.4.5.3. Unique Properties

The `unique` property of a property object specifies whether the property is unique. Unique properties have `"unique":true`:

Query	Result

```
{
  "id":"/music/track/length",
  "/type/property/unique":null
}
```
	```
	{
	  "id" : "/music/track/length",
	  "/type/property/unique" : true
	}
	```

Non-unique properties can have the value `false`, but are more likely to have this property unset and return `"unique":null`.

3.4.5.4. Reciprocal Properties

Links between objects in the Metaweb database are inherently bidirectional. Types, like `/music/artist` and `/music/album`, that are designed to work together, often take advantage of this bi-directionality by declaring pairs of reciprocal properties. Any link between an artist and an album result is visible through he `/music/artist/album` property and also through its reciprocal `/music/album/artist`.

The reciprocity of properties is apparent through the `master_property` and `reverse_property` properties. When we queried `/music/artist/album` above, we learned that it has a `master_property` of `/music/album/artist` and a `reverse_property` of `null`. Let's query the reciprocal property now:

Query	Result
`{`	`{`
`  "id":"/music/album/artist",`	`  "id":"/music/album/artist",`
`  "type":"/type/property",`	`  "type":"/type/property",`
`  "master_property":null,`	`  "master_property":null,`
`  "reverse_property":null`	`  "reverse_property":"/music/artist/album"`
`}`	`}`

We can determine from these results that the `artist` property of `/music/album` is the "master" property and the `album` property of `/music/artist` is the "reverse" property. These names imply more of a hierarchy than is really necessary: both properties are real, and you can usually write MQL queries without knowing whether a property is "master" or "reverse". Another way of thinking about master versus reverse properties is to assign a directionality to links. We can say that the link between an artist and an album is directed from the album to the artist. That is, it is an outgoing link from the album object and an incoming link to the artist object.

3.4.5.5. Other Properties of Properties

The `enumeration` property of `/type/property` was explained in §3.3.5: it refers to the namespace within which the value of the property becomes a key:

Query	Result
`{`	`{`
`  "id":"/type/lang/iso639",`	`  "id":"/type/lang/iso639",`
`  "/type/property/enumeration":null`	`  "/type/property/enumeration":"/lang"`
`}`	`}`

When the `expected_type` of a property is a numeric value, such as `/type/float` or `/type/int`, the `unit` property of the property typically refers to a `/type/unit` object that provides an interpretation for the value. Consider the `/music/track/length` property:

Query	Result
`{`	`{`
`  "id":"/music/track/length",`	`  "id" : "/music/track/length",`
`  "type":"/type/property",`	`  "type":"/type/property",`
`  "expected_type":null,`	`  "expected_type":"/type/float",`
`  "unit":null`	`  "unit":"/en/second"`
`}`	`}`

The `links` property of a property is the set of `/type/link` links that the property represents. We'll learn more about links in §3.7.

3.5. MQL Directives

Directives are special MQL commands that specify additional details about a query or request additional processing of query results. Because MQL is based on JSON, MQL directives look just like ordinary properties and values. The names of MQL directives are reserved words in MQL, however, and Metaweb does not allow types to define properties with these names.

The sections that follow document the `limit`, `return`, `sort`, `index` and `optional` directives. MQL also supports a `link` directive, which is covered in §3.7.

3.5.1. Limiting Results

To reduce resource consumption and bandwidth usage, Metaweb never returns more than 100 matches for a query (or for a sub-query) unless you explicitly ask for more. The Freebase database contains thousands of bands, for example, but this query only returns 100 of them:

```
[{
  "type":"/music/artist",
  "name":null
}]
```

To change the number of desired results to a larger, or a smaller, number, use the `limit` directive. Here, for example, is a query that returns the names of up to 2000 bands:

```
[{
  "type":"/music/artist",
  "name":null,
  "limit":2000
}]
```

Although MQL allows you to request arbitrarily large numbers of results, Metaweb does not guarantee that you'll always get an answer. Complicated queries with a large number of results may time out before Metaweb can complete the result. A better solution for large queries is to retrieve the results in batches, using a *cursor*. Cursors are not part of MQL: instead they are part of the *mqlread* service for delivering MQL queries to a Metaweb database. They are documented in §4.2.4.1.

Specifying a limit of 1 tells Metaweb that you're only interested in one result, and allows you to omit square brackets from your query. The following query, for example, is guaranteed not to result in a uniqueness error, even if there are two bands that have the same name:

```
{
  "type": "/music/artist",
  "name": "The Police",
  "id": null,
  "limit": 1
}
```

Specifying a limit of 0 can be useful to prune the result tree of values you aren't really interested in. The following query, for example, asks "What are the names of three bands who have recorded the song *Masters of War*? I'm only interested in the band name, so don't include the name of the song in the results":

Query

```
[{
  "type":"/music/artist",
  "name":null,
  "track":{
    "name":"Masters of War",
    "limit":0
  },
  "limit":3
}]
```

Result

```
[{
  "type" : "/music/artist",
  "name" : "Kevn Kinney",
  "track" : null
},{
  "type" : "/music/artist",
  "name" : "Timesbold",
  "track" : null
},{
  "type" : "/music/artist",
  "name" : "Bob Dylan",
  "track" : null
}]
```

Notice that the result of the query does not include a `limit` property. MQL responses normally have the same structure as their query, but most directives are not considered part of this structure and are not included in responses.

Since the `limit` directive must appear within curly braces, limiting a query sometimes requires you to transform a simple query into a more complex one (with more complex results). Consider this query to list all albums by The Police:

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : []
}
```

If we want to limit the result to five albums, we must rewrite the query as follows:

Query

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : [{"name":null, "limit":5}]
}
```

Result

```
{
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : [
    {"name" : "Outlandos d'Amour"},
    {"name" : "Reggatta de Blanc"},
    {"name" : "Zenyatta Mondatta"},
    {"name" : "Ghost in the Machine"},
    {"name" : "Synchronicity"}
  ]
}
```

Finally, note that a limit of `n` means "return the first `n` results" not "pick `n` results at random". Results of Metaweb queries are unordered (unless you use the `sort` directive, which will be introduced shortly) so the results returned are effectively arbitrary. They are repeatable, however: re-running the same query will typically return the same `n` results.

3.5.2. Counting Results

MQL supports three directives for counting or estimating the number of matches for a query. This section explains the `return`, `count`, and estimate-`count` directives.

Sometimes you don't care *what* the results of a query are: you just want to know *how many* results there are. To find out, use the `return` directive. Here's how we'd ask how many albums The Police have released:

Query	Result
`{`	`{`
`  "type" : "/music/artist",`	`  "type" : "/music/artist",`
`  "name" : "The Police",`	`  "name" : "The Police",`
`  "album" : { "return":"count" }`	`  "album" : 22`
`}`	`}`

The value of the `return` directive is `"count"`. Notice that this directive goes inside curly braces in the query, but the count is returned as a plain integer value. If we were interested in the names of the albums, we'd obviously have to use square brackets in the query so that Metaweb could return an array of results to us. In this case, the square brackets are unnecessary. (Though if we'd used them, Metaweb would return `[22]`.)

If `return:count` appears at the top-level of a query, then Metaweb returns just the count:

Query	Result
`{`	`22`
`  "type" : "/music/album",`	
`  "artist" : "The Police",`	
`  "return":"count"`	
`}`	

Notice that in this case, the presence of the `return` directive causes the result to have a completely different JSON structure than the query does.

If a top-level query finds no matches, it returns a count of 0. Let's ask how many albums named "Arrested" have been released by The Police:

Query **Result**

```
{                            0
  "type" : "/music/album",
  "artist" : "The Police",
  "name" : "Arrested",
  "return":"count"
}
```

Note, however, that we get a different result when we ask the question this way:

Query **Result**

```
{                             null
  "type" : "/music/artist",
  "name" : "The Police",
  "album" : {
    "name": "Arrested",
    "return":"count"
  }
}
```

The result of this query without the `return:count` directive is `null`: it does not match anything. So there is nothing to count, and the query returns `null` even with the directive. Adding an `optional` directive to the sub-query solves this problem. We'll learn about the `optional` directive in §3.5.5.

If your query is complex, or if there are many results, the request may timeout before Metaweb can count the exact number of matches. If timeouts are a concern, use `return:estimate-count` instead of `return:count`. As its name implies, this version of the directive returns an estimate of the number of matches rather than an exact count. The following query, for example, asks for the approximate number of musical artists in the database:

Query **Result**

```
{                             354200
  "type" : "/music/artist",
  "return":"estimate-count"
}
```

With a dataset this large, asking for an exact count with `return:count` is likely to result in a timeout.

In some circumstances, you may want to ask for the results of a query (up to the explicit limit you specify or up to the implicit limit of 100 results) and also ask for a count or estimate of the total number of results available. You might want to do this, for example, if you were displaying the results in paged form and wanted to include a message of the form "Results 1-20 of 317" on the first page. (The trick to retrieving subsequent pages of results is to use a *cursor*. This is explained in §4.2.4.1).

If you want to retrieve results and a count, use the `count:null` directive or the `estimate-count:null` directive instead of the `return` directive. For example, to ask for the names of tracks by The Police, and a count of the total number of tracks, you might use this query:

```
Query                        Result
[{                           [{
  "type" : "/music/track",     "type" : "/music/track",
  "artist" : "The Police",     "artist" : "The Police",
  "name" : null,               "name" : "Can't Stand Losing You",
  "count" : null               "count" : 133
}]                           },{
                               "type" : "/music/track",
                               "artist" : "The Police",
                               "name" : "Message in a Bottle",
                               "count" : 133
                             },
                             // 98 more results omitted...
                             ]
```

This query returns 100 results, and the `count` property tells us that more results are available. Notice that the `count` property appears over and over again in each of the results. (Just as the `type` and `artist` properties do).

We could also have used `estimate-count:null` in the query. In that case, each of the 100 results would have included `estimate-count:135`.

3.5.3. Sorting Results

Use the `sort` directive if you'd like the Metaweb server to sort the results of your query before returning them. For example, to ask for the names of the tracks on an album in alphabetical order, sort them by name:

```
// Tracks on the album Synchronicity, in alphabetical order
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track": [{
    "name":null,
    "sort":"name"
  }]
}
```

As you can see, the `sort` directive simply specifies the name of the property by which the sort is to be done. To order these same tracks from shortest to longest, use "length" as the sort key:

```
// Tracks on the album Synchronicity, from shortest to longest
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track": [{
    "name":null,
    "length":null,
    "sort":"length"
  }]
}
```

Note that the query above includes `"length":null`. If you want to use a property as a sort key, you must query that property.

To reverse this order, precede the name of the sort key by a minus sign:

```
// Tracks on the album Synchronicity, from longest to shortest
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track": [{
    "name":null,
    "length":null,
    "sort":"-length"
  }]
}
```

The sorts shown above are convenient, but could easily be duplicated on the client side. That is, you could request unordered results from Metaweb and sort them yourself. One situation in which the `sort` directive cannot be duplicated on the client is when it interacts with the `limit` directive. Result sets are truncated to the specified limit after the sort is applied. Use `sort` and `limit` together in queries like this:

```
// What is the longest track on Synchronicity?
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track": {
    "name":null,
    "length":null,
    "sort":"-length",
    "limit":1
  }
}
```

Sorting need not be limited to a single sort key. To specify more than one key, use an array on the right-hand side of the `sort` directive:

```
// List tracks by The Police, sorted from shortest to longest.
// Tracks of the same length should be in alphabetical order.
[{
  "type":"/music/track",
  "artist":"The Police",
  "name":null,
  "length":null,
  "sort":["length","name"]
}]
```

If your query includes sub-queries, then the properties of those sub-queries can also be used as sort keys. The query below uses this kind of hierarchically-named sort key. Note also that it has two distinct sort clauses.

```
// List all albums by The Police, along with the name of their longest track.
// Order the albums from longest longest track to shortest longest track.
[{
  "type":"/music/album",
  "artist":"The Police",
  "name":null,
  "track":{
    "name":null,
    "length":null,
    "sort":"-length",
    "limit":1
  },
  "sort":"-track.length"
}]
```

Finally, here is a complex example that uses multiple sort keys, hierarchically-named sort keys, and a sort key that includes a fully-qualified property name (see §3.4.1):

```
// Return a list of performances (character/actor pairs) in George Lucas films.
// Sort them by character name. If the character appears in more than
// one film, sort by film name. If more than one actor portrays the character
// in a single film, sort them by actor birthdate (most to least recent).
[{
  "type":"/film/performance",
  "film":{
    "name":null,
    "directed_by":"George Lucas"
  },
  "character":null,
  "actor":{
     "name":null,
     "/people/person/date_of_birth":null
  },
  "sort":["character",
          "film.name",
          "-actor./people/person/date_of_birth"]
}]
```

3.5.4. Ordered Collections

If you do not include a `sort` directive in a query then Metaweb makes not guarantees about the order in which results are returned. If you don't ask for the data to be sorted, you should treat the result as an unordered set of values rather than an ordered list. [7]

Some data, such as the tracks on an album, have a natural order: the order in which they are arranged in the album. If you want results to be sorted according to this natural ordering, use `"sort":"index"`. (Or, to reverse the natural ordering, use `"sort":"-index"`.

```
// Return the tracks on the album Synchronicity in the order that they appear
{
  "type":"/music/album",
  "artist":"The Police",
  "name":"Synchronicity",
  "track":[{
    "name":null,
    "index":null,
    "sort":"index"
  }]
}
```

Since we've used "index" as a sort key, we must query the value of "index" as well. `index` is a MQL directive that looks like a property, because it is queried with `null`, and because, unlike other directives, it is included in query results. When you include the `"index":null` directive in a query, the results include `index` properties whose values are the integers between 0 and `n`-1, where `n` is the number of ordered results. It is important to understand that indexes do not apply to objects in Metaweb, but to the relationships between objects. It is the link between the album "Synchronicity" and the track "Mother" that has an index of 3 (because it is the fourth song on the album), not the track itself. This becomes clear when you consider the case of a track that appears on more than one album: if "Mother" also appears on an album named "Greatest Hits" it is likely to have a different index on that album.

`index` is a directive, not a property. MQL read queries may use `index` as a sort key, and they may query the index with `"index":null`, but may not use the keyword in any other way. You cannot write `"index":1` to ask for the second item in a set, for example. Similarly, `index` cannot be used with operators such as `<` to select indexes less than a given value. (We'll learn about `<` and other operators in §3.6.) The `index` directive can be used in other ways in write queries, however, and we'll learn about that in Chapter 5.

The index of a track on an album is an important and useful detail. And because a single track can appear on more than one album, it is not possible to capture this ordering with a property on the track object. It is for this reason that the `index` directive is useful here. But this is a somewhat unusual case, however, and links in Freebase do not typically have an order associated with them. Consider the set of albums by The Police instead of the set of tracks on Synchronicity. The most natural order for albums by a band is probably by release date. But this is captured by the `re-`

[7] Metaweb's ordered collections are sometimes described as lists, but this term is inaccurate because lists are allowed to have duplicate elements. Metaweb's ordered collections are still fundamentally sets, and duplicates are not allowed.

lease_date property of the album object, and there is no need to sort on index to obtain a chronological list of albums.

While most Freebase data is unordered, bear in mind that some is partially ordered. Any time you use the "index":null directive, you should be prepared for returned indexes of null to be returned along with the numbered indexes. The /film/film and /film/performance types provide an example. The starring property of a film links to performance objects that specify the cast (the actors and the characters they portray) of the film. Indexes are sometimes used to capture the billing order of the top stars in the movie, while minor performances are left unordered.

The index directive can be used in conjunction with the sort and limit directives. Consider the following query, which ask for the top two stars in the movie *Psycho*:

Query

```
[{
  "type" : "/film/film",
  "name" : "Psycho",
  "directed_by":"Alfred Hitchcock",
  "starring" : [{
    "actor" : null,
    "character" : null,
    "index" : null,
    "sort" : "index",
    "limit" : 2
  }]
}]
```

Result

```
[{
  "type" : "/film/film",
  "name" : "Psycho",
  "directed_by" : "Alfred Hitchcock",
  "starring" : [{
    "actor" : "Anthony Perkins",
    "character" : "Norman Bates",
    "index" : 0
  },{
    "actor" : "Janet Leigh",
    "character" : "Marion Crane",
    "index" : 1
  }]
}]
```

3.5.4.1. Indexes are Relative

The query above correctly returns the names of the final two tracks on the album Synchronicity. Look carefully, however at the index values it returns: the last track is given an index of 1 and the penultimate track an index of 0. This is not a bug: this query simply reveals the true nature of ordered collections in Metaweb. Metaweb does not include an absolute index for each link. The implementation is able to say whether any link is greater-than or less-than another, but it cannot tell you the absolute position of that link within the complete set of links.

The number that Metaweb returns as the value of the index property is a synthetic one, generated by Metaweb as a simple way to express the order of elements. If Metaweb returns an array holding *n* indexed elements, then it generates index values for those elements that range from 0 to *n*-1. (There may be additional elements in the array that have an index of null, however.) For example, if you ask for the last two tracks on an album, the resulting values have indexes 0 and 1. If you ask for tracks that are shorter than 2 minutes (we'll learn how to do this in §3.6) and Metaweb finds three of them, then it will assign them index values of 0, 1, and 2. If you want to know the track number for the tracks on a particular album, you must query the complete set of tracks. Then add one to the index value to get the track number. If you want to know the track numbers of the short songs, you must query the complete set of tracks, and search for the short songs yourself.

3.5.5. Optional Matches

In addition to the `limit`, `return`, `sort` and `index` directives, MQL also includes an `optional` directive. If part of your query is not required to match, add `"optional":"optional"` to it. For example, we can use the `optional` directive to ask the question: "What bands have recorded the song "Masters of War", and do they have a Greatest Hits album?". The query looks like this:

```
Query                           Result
[{                              [{
  "type" : "/music/artist",       "type" : "/music/artist",
  "name" : null,                  "name" : "Kevn Kinney",
  "track" : "Masters of War",     "track" : "Masters of War",
  "album" : {                     "album" : null
    "name" : "Greatest Hits",   },
    "optional" : "optional"     {
  }                               "type" : "/music/artist",
}]                                "name" : "Timesbold",
                                  "track" : "Masters of War",
                                  "album" : null
                                },
                                {
                                  "type" : "/music/artist",
                                  "name" : "Bob Dylan",
                                  "track" : "Masters of War",
                                  "album" : { "name" : "Greatest Hits" }
                                }]
```

Without the `optional` directive, the query would only return bands whose have recorded Masters of War *and* have released a Greatest Hits album. With the optional directive, we get all bands who have recorded the song, and additionally, we find out whether or not they have released a Greatest Hits album.

Optional queries can be nested inside optional queries. The following query is an extension to the one above. It further asks whether "Masters of War" appears on the Greatest Hits album. Without the nested `optional` directive, only Greatest Hits albums that include the song would be matched.

```
[{
  "type" : "/music/artist",
  "name" : null,
  "track" : "Masters of War",
  "album" : {
    "name" : "Greatest Hits",
    "optional" : "optional",
    "track" : {
      "name" : "Masters of War",
      "optional" : "optional"
    }
  },
}]
```

MQL allows `"optional":true` instead of `"optional":"optional"`. You can also write `"optional":"required"` or `"optional":false` to indicate that a match is required, but this is the default and is never necessary or useful.

3.5.5.1. When to use the optional Directive

In order to understand when to use the `optional` directive, it is important to understand when a sub-query requires a match and when it does not. Suppose we want to ask for a list of artists who have recorded "Masters of War" and for the nicknames of those artists, if they have any:

```
[{
  "type" : "/music/artist",
  "track" : "Masters of War",
  "name" : null,
  "/common/topic/alias" : []
}]
```

This request for aliases is implicitly optional: if an artist has no nicknames, `[]` will be returned. The same is true if we write `"/common/topic/alias":[{}]`. If none of the matching artists has more than one nickname, then we could even write `"/common/topic/alias":null`. In that case, each result would include either the single nickname or `null`. These queries simply ask for aliases, they do not constrain the query to match only artists that do have nicknames. Suppose, however that we're only interested in English nicknames:

```
[{
  "type" : "/music/artist",
  "track" : "Masters of War",
  "name" : null,
  "/common/topic/alias" : [{"value":null, "lang":"/lang/en"}]
}]
```

We've now introduced a constraint into the sub-query, and only artists who have at least one English-language alias will match. Even though this alias sub-query is in square brackets, it still requires at least one match unless we add an `optional` directive:

```
[{
  "type" : "/music/artist",
  "track" : "Masters of War",
  "name" : null,
  "/common/topic/alias" : [{
    "optional":true,
    "value":null,
    "lang":"/lang/en"
  }]
}]
```

The queries `[]` and `[{}]` do not require a match, but putting any property inside the curly braces transforms the sub-query into one that is required. This is true even if that property is a query rather than a constraint. So while `"/common/topic/alias":[]` will match objects that have no aliases, the same is not true of this query:

```
"/common/topic/alias":[{"value":null}]
```

By explicitly stating our request for the value of each alias, we've transformed the query from one that is implicitly optional to one that requires at least one match.

3.5.5.2. Optional and return:count

Queries that use `return:count` can add `optional:true` so that they can return a count of zero instead of returning `null` to indicate a failure to find a match:

```
Query                          Result
{                              {
  "type" : "/music/artist",      "type" : "/music/artist",
  "name" : "The Police",         "name" : "The Police",
  "album" : {                    "album" : 0
    "name": "Arrested",        }
    "return":"count",
    "optional":true
  }
}
```

Without the `optional` directive, this query would return `null` because no match would be found at all. Note that using `optional` is only necessary in sub-queries: `return:count` at the toplevel returns 0 when no match is found.

3.5.6. Forbidden Matches

The `optional` directive can be used in another important way. If we include `"optional":"forbidden"` in a sub-query, then the results will not include any values for which the sub-query matches. Here, for example, is how we find bands that have recorded "Masters of War" and have *not* released a Greatest Hits album:

Query	Result

`[{`
`  "type" : "/music/artist",`
`  "name" : null,`
`  "track" : "Masters of War",`
`  "album" : {`
`    "name" : "Greatest Hits",`
`    "optional" : "forbidden"`
`  },`
`}]`
	`[{`
	`  "type" : "/music/artist",`
	`  "name" : "Kevn Kinney",`
	`  "track" : "Masters of War",`
	`  "album" : null`
	`},`
	`{`
	`  "type" : "/music/artist",`
	`  "name" : "Timesbold",`
	`  "track" : "Masters of War",`
	`  "album" : null`
	`}]`

Note that if a sub-query includes `"optional":"forbidden"`, the response to that sub-query will always be `null`. Note also that there is no boolean alternative to the string "forbidden" – a value of `false` means "required", not "forbidden".

A query can include multiple forbidden sub-queries. This query, for example, finds bands that have not released albums named "Greatest Hits" or "The Best Of":

```
[{
  "type":"/music/artist",
  "name":null,
  "neither:album":{"optional":"forbidden", "name":"Greatest Hits"},
  "nor:album":{"optional":"forbidden", "name":"The Best Of"}
}]
```

When we use `null` in a MQL query, we're making a request for the value of a property, rather than constraining that property to have the value `null`. You can use `optional:forbidden` to write queries that constrain a property to be `null`. Suppose we wanted to ask Freebase for a list of bands that have no known albums (note that an `optional:forbidden` clause must always accompany another clause, so `id:null` has been added to the query):

```
[{
  "type" : "/music/artist",
  "name" : null,
  "album" : { "optional" : "forbidden",
  "id" : null,
  }
}]
```

MQL also supports a `!=` operator, which excludes results from a query in a different way. We'll learn about `!=` in §3.6.4.

3.6. MQL Operators

A property in a MQL query that is not itself a query (i.e. one whose value is not `null` or `[]`) expresses a constraint. A normal property/value pair constrains results to objects for which that property is *equal to* that value. But "equal to" is not the only constraint we can express in MQL. In this section we introduce `operators` such as <, ~=, != and |= which express constraints "less than", "matches", "not equal to", and "one of".

If MQL was not based on JSON, these operators could naturally appear between a property name and its value. Since MQL uses JSON, however, operators are instead appended to a property name and appear inside the quotation marks. To express a "less than" constraint, for example, we might write:

```
"date_of_birth<" : "2000"
```

When a property name is followed by an operator, the value that follows must be a JSON literal (or, for the `|=` operator an JSON array): MQL sub-queries in curly braces are not allowed with operators. Finally, note that properties that include operators are like MQL directives and do not appear in the results of the query.

3.6.1. Order Constraints

We know how to ask "what are the names and lengths of the tracks on the album *Synchronicity* by The Police?". The query looks like this:

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track":[{"name":null, "length":null}]
}
```

Metaweb also allows us to ask "What are the names and lengths of the *long* songs on the album?" The query below includes a numeric constraint on the `length` property, and the *freebase.com* response only includes the two songs on the album that are longer than 300 seconds:

Query

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track":[{
    "name":null,
    "length":null,
    "length>":300
  }]
}
```

Result

```
{
  "type" : "/music/album",
  "name" : "Synchronicity",
  "artist" : "The Police",
  "track" : [{
    "name" : "Synchronicity II",
    "length" : 305.066
  }, {
    "name" : "Wrapped Around Your Finger",
    "length" : 313.733
  }]
}
```

The line "length>":300 in the query expresses a constraint to Metaweb: it specifies that the track must be longer than 300 seconds. In addition to >, you can also use < for less-than, and <= and >= for less-than-or-equal and greater-than-or-equal. Note, however, that no spaces are allowed before or after these punctuation characters.

Note that constraining the length property with > does not automatically query the property. We must include a separate "length":null line in our query if we want to know the exact length of the returned tracks. Note that according to JSON syntax length> is a different property than length, so there is no need to use a property prefix to distinguish the constraint from the query.

You can include more than one numeric constraint on the same property, restricting the value to a range. Here's how we ask for songs that are at least three minutes long, but less than four:

```
{
  "type":"/music/album",
  "name":"Synchronicity",
  "artist":"The Police",
  "track":[{
    "name":null,
    "length>=":180,
    "length<":240
  }]
}
```

Note that this query constrains the value of the length property, but does not ask Metaweb to return the exact value of the property.

Numbers are not the only type that can be constrained with these operators. Here, for example, is a query that constrains a /type/datetime property to obtain a list of albums released in January 1999:

```
[{
   "type":"/music/album",
   "name":null,
   "artist":null,
   "release_date>=":"1999-01-01",
   "release_date<=":"1999-01-31"
}]
```

When the <, >, <=, and >= operators are used with strings, they compare in case-insensitive, Unicode-aware alphabetical order. For example, to find bands whose name begins with the letter A or B, use this query:

```
[{
  "type" : "/music/artist",
  "name" : null,
  "name>=" : "A",
  "name<" : "C"
}]
```

It is not legal to use any of these order operators with the `id` (or `guid`) property. Fully-qualified names and guids cannot be compared alphabetically or numerically.

3.6.2. Pattern Matching with the ~= Operator

The MQL pattern matching operator ~= tests a property to see if it contains a specified word or phrase. [8] To try this out, let's find some short songs about love:

```
[{
   "type":"/music/track",
   "artist":null,
   "name":null,
   "name~=":"love",
   "length":null,
   "length<":120
}]
```

Here's a query for songs about love recorded by bands whose name begins with "The":

```
[{
   "type":"/music/track",
   "artist":null,
   "artist~=":"^The",
   "name":null,
   "name~=":"love"
}]
```

Results include *Love Shack* by The B-52's and *For Your Love* by The Yardbirds. Notice that the constraint on the `artist` property in the query above uses the `^` character to specify that the word The must appear at the beginning of the artist's name. (This is like the anchor syntax used in regular expressions, but note that MQL patterns are not nearly as general as regular expressions.)

Here's a query to find all bands whose name is two words long and begins with the word The (such as The Police and The Clash).

```
[{
  "type" : "/music/artist",
  "name" : null,
  "name~=" : "^The *$"
}]
```

This query is interesting in several ways. First, it uses `^` again to anchor the match to the beginning of the string. And it uses `$` to anchor to the end of the string. The `*` character matches any string of characters (other than spaces).

Table 3.2 summarizes MQL pattern matching syntax.

[8] If you've done programming with languages like Perl or Ruby, this syntax should look familiar. If you've worked with SQL queries for relational databases, `~=` is like the SQL `%` operator. Otherwise, think of "~=" as meaning "approximately equal" or "like".

Table 3.2. MQL Pattern Matching Syntax

Pattern	Matches
`love`	Matches any string that contains the word "love". Does not match strings containing "glove" or "lover".
`love you`	Matches any string that contains the exact phrase "love you", such as "Hello, I Love You" but not "All You Need is Love".
`love*`	Matches any string containing a word that begins with "love", such as "love", "lover" or "lovely". Does not match "glove".
`*love`	Matches any string containing a word that ends with "love", such as "love" or "glove".
`*love*`	Matches any string that contains "love", such as "love", "glove", "lover" and "glover".
`^`	Matches the beginning of a string. For example, `^the` matches any string that begins with the word "the", and `^the*` matches any string that begins with a word that begins with "the", such as "they" or "there".
`$`	Matches the end of a string. For example, `hits$` matches any string that ends with the word "hits", and `*love$` matches "Sunshine of your Love" and "Smell the Glove".
`*`	Matches a single word. `^*$`, matches any single-word string, for example, and `I * you` matches any string that contains a 3-word phrase beginning with "I" and ending with "you".
`-`	A hyphen or other punctuation matches an optional space. For example, `bi-directional` matches "bi directional", "bi-directional", or "bidirectional".
`\`	Use a backslash to escape any punctuation character that you want to match literally. `bi\-directional` matches any string that contains the hyphenated word "bi-directional", for example. Note, however, that JSON string literals require backslashes like this to be doubled. If you type a JSON query "by hand" or use string manipulation techniques to create a query, be sure to double the backslashes. If you use a JSON serializer to create the query, it should double the backslashes for you.
numbers	When the pattern to be matched looks like a number, any numbers in the text that is being matched are first converted to normalized form. This means that leading zeros are removed, trailing zeros after the decimal point are removed, a zero is added before the decimal point if there is no digit there, and so on. If the match against the normalized text does not succeed, it is tried again with the numbers in their original, unnormalized form. This means that the pattern "7" matches "Agent 007", "July 07, 2008", and "7.0". But the pattern "007" does not match "7", "07", or "7.0".

Here's another example: what bands have three-word names that begin with "the" and end with a plural (e.g. The Beach Boys, The Doobie Brothers)?

```
[{
  "type" : "/music/artist",
  "name" : null,
  "name~=" : "^The * *s$"
}]
```

Note that when a pattern includes multiple words (or even a word and an asterisk), Metaweb doesn't just attempt to match each word individually: it looks for a matching phrase. The pattern

"I love" only matches strings that contain those two words in that order. It does not match "I want your love", for example. If you want to match any string that contains the words "I" and "love", regardless of order, you should use two separate properties:

```
[{
   "type":"/music/track",
   "name":null,
   "a:name~=":"I",
   "b:name~=":"love"
}]
```

Here are two final notes about MQL pattern matching. First, all searches are case-insensitive. Second, it is not legal to perform pattern matching on the `id` (or `guid`) property.

3.6.3. The "one of" Operator |=

MQL uses the `|=` operator to restrict the value of a property to a set of possible values, which are expressed as a JSON array of JSON literals. The constraint says "match any one of the values in this array". Here's how we can find a list of bands who have recorded an album named "Greatest Hits" or an album named "Super Hits":

```
[{
  "type":"/music/artist",
  "name":null,
  "album|=":["Greatest Hits","Super Hits"],
  "album":[]
}]
```

The `album` property in the response will include one or more album names, but the names will all be either "Greatest Hits" or "Super Hits".

The values in the array can be numbers instead of strings. Here's how we look up the names of the first three chemical elements:

Query	Result
`[{`	`[{`
`  "type":"/chemistry/chemical_element",`	`  "type" : "/chemistry/chemical_element",`
`  "name":null,`	`  "name" : "Hydrogen",`
`  "atomic_number|=":[1,2,3],`	`  "atomic_number" : 1`
`  "atomic_number":null,`	`},{`
`  "sort":"atomic_number"`	`  "type" : "/chemistry/chemical_element",`
`}]`	`  "name" : "Helium",`
	`  "atomic_number" : 2`
	`},{`
	`  "type" : "/chemistry/chemical_element",`
	`  "name" : "Lithium",`
	`  "atomic_number" : 3`
	`}]`

Unlike the order and pattern-matching operators, the `|=` operator can be used on the `id` property, and this is a useful way to run the same query over multiple objects specified by id. The following query asks for the properties of three types, for example:

```
[{
  "id|=":["/type/type", "/type/property", "/type/key"],
  "id":null,
  "/type/type/properties":[]
}]
```

Finally, here is an example that uses the `|=` constraint in two different places. It asks for the French and Spanish translations of the countries named "England" and "France"

Query	Result

```
[{
  "type":"/location/country",
  "english:name|=": ["England",
                     "France"],
  "english:name": null,
  "foreign:name": [{
    "value":null,
    "lang":null,
    "lang|=":["/lang/fr",
              "/lang/es"]
  }]
}]
```
	```
	[{
	  "type" : "/location/country",
	  "english:name" : "England",
	  "foreign:name" : [
	    {"lang" : "/lang/fr", "value" : "Angleterre"},
	    {"lang" : "/lang/es", "value" : "Inglaterra"}
	  ]
	},{
	  "type" : "/location/country",
	  "english:name" : "France",
	  "foreign:name" : [
	    {"lang" : "/lang/fr",  "value" : "France"},
	    {"lang" : "/lang/es",  "value" : "Francia"}
	  ]
	}]
	```

Most MQL operators expect a single JSON literal as their value. The `|=` operator instead expects a JSON array of JSON literals. Only literals are allowed in the array: the following query, for example, is *not* legal:

```
[{
  "type":"/music/artist",
  "name":null,
  "album|=":[
    {"name":"Greatest Hits", "lang":"/lang/en"},   // Invalid MQL!
    {"name":"Super Hits", "lang":"/lang/en"}
  ],
  "album":[]
}]
```

3.6.4. The "but not" Operator !=

The != operator says that the constrained property can be *anything but* the specified value. (It does require that the property be *something*, however: it does not match object for which the property is null.) Here, for example is how we list albums by The Police, but not Greatest Hits:

```
{
  "type":"/music/artist",
  "name":"The Police",
  "album":[{
    "name":null,
    "name!=":"Greatest Hits"
  }]
}
```

And here we use != to list the names of chemical elements other than elements 1, 2, and 3:

```
[{
  "type":"/chemistry/chemical_element",
  "name":null,
  "atomic_number!=":1,
  "a:atomic_number!=":2,
  "b:atomic_number!=":3
}]
```

The != operator is like the optional:forbidden directive in that it excludes values from the results. The operator and the directive are quite different, however, and it is important to understand when to use each. Contrast the album query above that uses != with the following query, which uses optional:forbidden to say "list albums by The Police, excluding any that contain the song Roxanne":

```
{
  "type":"/music/artist",
  "name":"The Police",
  "album":[{
    "name":null,
    "track":{
      "name":"Roxanne",
      "optional":"forbidden"
    }
  }]
}
```

The != operator excludes a single JSON literal and is useful with unique properties, like atomic_number and name (which behaves like a unique property even though it technically isn't). The optional:forbidden directive, on the other hand, excludes any match of the sub-query that contains it. It works with non-unique properties and expresses the idea *may not include*.

The query with which we began this section can be simplified as follows:

```
{
  "type":"/music/artist",
  "name":"The Police",
  "album":[],
  "album!=":"Greatest Hits"
}
```

This makes the query more difficult to understand, however. It appears as if the `!=` operator is constraining the non-unique property `album`. In fact, however, it is expressing a constraint on the default `name` property of any albums.

It is important to understand that when you use `!=` on a property you are implicitly constraining the results to objects for which that property exists and has a value different than the one you specify. Even though a property value of `null` is different than the one you specify, it is not matched. Consider the following three queries, for example:

```
// How many albums have The Police released?
{
  "return":"count",
  "type":"/music/album",
  "artist":"The Police",
}
// How many live albums have they released?
{
  "return":"count",
  "type":"/music/album",
  "artist":"The Police",
  "release_type":"Live Album"
}
// How many non-live albums have they released?
{
  "return":"count",
  "type":"/music/album",
  "artist":"The Police",
  "release_type!=":"Live Album"
}
```

The first query returns a count of 22, and you might expect that since every album is either a live not live the sum of the counts of the second and third queries would be 22. But, in fact, the second query returns 3 and the third returns 8. There are 8 Police albums that have a defined `release_type` whose value is something other than "Live Album". The other 11 Police albums do not have a `release_type` defined in Freebase (or did not when this was written). It turns out that `optional:forbidden` is what we want here. This query returns the 19 albums we expected:

```
{
  "return":"count",
  "type":"/music/album",
  "artist":"The Police",
  "release_type" : {"optional":"forbidden", "name":"Live Album"}
}
```

3.6.5. Expressing AND, OR, and NOT in MQL

We conclude this section on MQL operators with a discussion of Boolean operations in MQL queries. Boolean AND is the default operation in MQL: each of the properties in a MQL query expresses a constraint, and these constraints are implicitly ANDed together. Consider:

```
[{
  "type":"/music/artist",
  "name":null,
  "name~=":"^The",
  "album":"Greatest Hits"
}]
```

This query says: tell me the names of objects which have type "/music/artist" AND which have a name that begins with "The" AND which have an album named "Greatest Hits".

The ability to use property prefixes in MQL allows us to express an AND on the value of a single property, as in the following query which asks "What are the names of objects that are both musical artists AND people AND have recorded a Christmas album?":

```
[{
  "name":null,
  "type":"/music/artist",
  "and:type":"/people/person",
  "album~=":"Christmas"
}]
```

The `|=` operator was introduced as the "one of" operator, but note that we could also have called it the "OR" operator, as in the following query, which asks "Return the names of albums recorded by The Police OR Sting"

```
[{
  "type":"/music/album",
  "name":null,
  "artist|=":["The Police","Sting"]
}]
```

Note that `|=` is a specialized operator, and expresses OR in a much less general way than the implicit AND of MQL. First of all, `|=` applies to only one property: it is not possible to request a list of albums released before 1990 or with genre "alternative rock". To obtain such a list, you must simply make two queries and combine the results. (It is possible to send two distinct queries in a single HTTP request. We'll learn how to do this in Chapter 4.)

MQL provides two distinct ways to express a Boolean NOT in a query. The `!=` operator says that a property must be *anything but* the specified value. The `optional:forbidden` directive instead specifies that the set of values for a property *may not include* an object that matches the sub-query of which it is a part. The distinction is subtle and it is important to understand when to each form of NOT, so we'll repeat some previously-seen examples.

First, here is how you can use `!=` to say "list the albums by The Police but NOT Greatest Hits":

```
{
  "type":"/music/artist",
  "name":"The Police",
  "album":[{
    "name":null,
    "name!=":"Greatest Hits"
  }]
}
```

Contrast that with the following query which asks for a list of bands that do NOT have a Greatest Hits album:

```
[{
  "type":"/music/artist",
  "name":null,
  "album":{
    "optional":"forbidden",
    "name":"Greatest Hits"
  }
}]
```

Let's conclude with an example that combines AND, OR, and NOT into a single query:

```
[{
  "type" : "/music/album",
  "name" : null,
  "name!=" : "Greatest Hits",
  "release_date|=" : ["1978","1979"],
  "a:genre" : "New Wave",
  "b:genre|=" : ["Punk Rock", "Post-punk"]
  "artist" : {
    "name" : null,
    "type" : {
      "id" : "/people/person",
      "optional" : "forbidden"
    }
  },
}]
```

This query asks for the names of albums which:

- are NOT named "Greatest Hits" AND
- were released in 1978 OR 1979 AND
- have a genre of "New Wave" AND
- also have a genre of either "Punk Rock" OR "Post-punk" AND
- were recorded by an musical artist who is NOT a person (i.e. by a band and not a solo artist).

Results include "Outlandos d'Amour" by The Police and "Go 2" by XTC.

3.7. Links, Reflection and History

This section of the chapter covers three related advanced features of MQL. The `link` directive and the type `/type/link` enable us to write MQL queries that return details about the links between objects rather than about the objects themselves. The first related feature is *reflection*. Reflection allows us to ask for all links to or from an object. It is something like MQL wildcards, but is link-based rather than type-based. The second advanced feature related to links is history. Metaweb databases are journal-based and retain a record of every link ever made, even after those links are deleted or replaced. It is possible, therefore, to use properties of `/type/link` to query the modification history of any Metaweb object.

3.7.1. Links to Sub-queries

Here's a query that we come back to time and again in this chapter. It asks for details about the album Synchronicity by The Police:

```
{
  "id" : "/en/the_police",
  "/music/artist/album" : {
    "name" : "Synchronicity",
    "track":[]
  }
}
```

Instead of asking about the tracks on the album, let's now ask for details on the link between the band and the album:

Query	Result
`{`	`{`
`  "id" : "/en/the_police",`	`  "id" : "/en/the_police",`
`  "/music/artist/album" : {`	`  "/music/artist/album" : {`
`    "name" : "Synchronicity",`	`    "name" : "Synchronicity",`
`    "link" : {}`	`    "link" : {`
`  }`	`      "type" : "/type/link",`
`}`	`      "master_property" : "/music/album/artist",`
	`      "reverse" : true`
	`    }`
	`  }`
	`}`

There are many points to note about this query and its results. We'll start by saying that `link` is a MQL directive, not an object property. A `link` directive requests information about the link between the object (the album in this case) that matches the query that the directive appears in and the object (the band) that matches the parent query. The `link` directive can only appear in a nested query – it makes no sense in a toplevel query. (Toplevel link queries require a different syntax, explained below.)

When we query a Metaweb object with `{}`, we get back the name, type, and id of the object. When we query a primitive with `{}`, we get back the type and value of the primitive (and also the `lang` and `namespace` properties of `/type/text` and `/type/key`). The results of the query above demonstrate that links are neither objects nor primitive values: they are something completely different. A link represents a relationship between two objects, it is not an object itself. Nor is it a primitive value: it carries too much information to be a simple primitive. Links do not have names, ids, or guids. Metaweb reports the type of a link as `/type/link`, but this is a synthetic type (like `/type/object`) that simply serves as a collection of properties: actual Metaweb objects are never assigned this type.

In addition to noting the absence of the `name` and `id` properties in the link results above, let's consider the `master_property` and `reverse` properties. The `master_property` property of a link identifies the fully-qualified name of the master property that connects the two objects. In this case we learn than the band The Police and the album Synchronicity are connected through the `/music/album/artist` property. The `reverse` property of the link tells us whether the link was followed forward or "in reverse". In this case, `reverse` is `true`, because we started with the band and followed the property `/music/artist/album` to the album. Recall that `/music/artist/album` is the reciprocal of `/music/album/artist`.

Links are different from objects and primitives in another way, too. The default property of a link is not `name`, `id`, or `value`. Instead, when we query a link using `"link":null`, it is the `master_property` of the link that is returned:

Query	**Result**
`{`	`  "id" : "/en/the_police",`
`  "id" : "/en/the_police",`	`  "/music/artist/album" : {`
`  "/music/artist/album" : {`	`    "name" : "Synchronicity",`
`    "name" : "Synchronicity",`	`    "link" : "/music/album/artist"`
`    "link" : null`	`  }`
`  }`	`}`
`}`	

The `master_property` and `reverse` properties of a link aren't terribly useful in queries like these, since the very structure of the query shows us the property that is being followed. They are useful in a different style of link query (shown below), however, and are also useful with reflective queries (explained later in this section). Links do have properties other than `master_property` and `reverse`, however, and we can use a wildcard to discover them:

Query

```
{
  "id" : "/en/the_police",
  "/music/artist/album" : {
    "name" : "Synchronicity",
    "link" : { "*" : null}
  }
}
```

Result

```
{
  "id" : "/en/the_police",
  "/music/artist/album" : {
    "name" : "Synchronicity",
    "link" : {
      "type" : "/type/link",
      "master_property" : "/music/album/artist",
      "reverse" : true,
      "source" : "Synchronicity",
      "target" : "The Police",
      "target_value" : null,
      "operation" : "insert",
      "valid" : true,
      "timestamp" : "2006-12-10T12:23:59.0685Z",
      "creator" : "/user/mwcl_musicbrainz"
    }
  }

}
```

The results of this query show a number of link properties. We've already seen `type`, `master_property` and `reverse`. `source` and `target` refer to the source and target of the link. Since we queried them here with a `"*":null` wildcard, we get the names of the source and target objects, rather than the objects themselves. If the target had been a primitive value (such as a name instead of an album), then the `target_value` property would have held the value of that primitive. (If the target is of `/type/text`, then the `target` property is the language of the text. If the target is of `/type/key`, then the `target` property is the namespace of the key.)

The `valid` and `operation` properties of a link specify the current validity of the link and the operation (such as insertion or deletion) that was performed on it. They are used when querying the history of a link, and are explained in §3.7.4. For now you just need to know that if you omit these properties from your link queries, Metaweb will only return links that *are* currently valid.

The `timestamp` and `creator` properties are like those defined by `/type/object` but they specify the creator and creation time for the link rather than for either of the two linked objects. Here is a query, for example, that asks about the creation (when and by who) of the objects that represent The Police, their album Synchronicity and of the link between those two objects:

```
Query                       Result
{                           {
  "id":"/en/the_police",      "id":"/en/the_police",
  "timestamp":null,           "timestamp":"2006-10-22T10:02:03.0012Z",
  "creator":null,             "creator":"/user/metaweb",
  "/music/artist/album": {    "/music/artist/album": {
    "name":"Synchronicity",     "name":"Synchronicity",
    "timestamp":null,           "timestamp":"2006-12-10T12:23:59.0119Z",
    "creator":null,             "creator":"/user/mwcl_musicbrainz",
    "link": {                   "link": {
      "timestamp":null,           "timestamp":"2006-12-10T12:23:59.0685Z",
      "creator":null,             "creator":"/user/mwcl_musicbrainz"
    }                           }
  }                           }
}                           }
```

Here's another link query that demonstrates the `/type/link/creator` property as well as `/type/link/target_value`. It asks: "what is the name of the country Spain in French, and what Metaweb user contributed the translation?":

```
Query                        Result
{                            {
  "id" : "/en/spain",          "id" : "/en/spain",
  "name" : {                   "name" : {
    "link" : {                   "link" : {
      "target" : "French",         "target" : "French",
      "target_value" : null,       "target_value" : "Espagne",
      "creator" : null             "creator" : "/user/mwcl_wikipedia_en"
    }                            }
  }                            }
}                            }
```

The link object described in the response represents a link between a `/location/country` object and a `/type/text` value. Since `/type/text` is a primitive, the `target_value` property holds the `value` of the primitive – the text itself. Since the target is of `/type/text`, the `target` property represents the language of the text, and we use this fact in the query to specify that we're interested in the French version of the name. It is worth noting that we specify the language by name here, rather than by id with `/lang/fr` as we usually do. The reason is that the `/type/link/target` property has an expected type of `/type/object` (since it can represent any object). The default property of `/type/lang` is `id`, which is why we normally specify languages by id. But in this case, it is the default property of `/type/object` that matters, which is why we must specify the name of the desired language rather than its id. (We could also have written `"target":{"id":"/lang/fr"}`

3.7.2. Toplevel Links

We saw above that the `link` directive allows us to query the link between the object or value that matches a sub-query and the object that matches its parent query. We can't use the `link` directive in a toplevel query, however. If you want to write a toplevel link query you must take a different, but simple, approach. Just include this constraint in your query:

```
"type":"/type/link"
```

`/type/link` is not a real type and no Metaweb objects have this type. But putting the line above into a MQL query tells Metaweb that you're interested in links rather than objects. When you write link queries of this sort, the `source` and `target` properties of `/type/link` typically become important to either constrain or query the source and destination of the link. (But note that you must write `"type":"/type/link"` in order to use the `source` and `target` properties: MQL does not allow you to use fully-qualified names for link properties, so you cannot omit the type specification and just query `/type/link/target` and `/type/link/source`.)

Here is how we query all outgoing links from The Police, asking for the name of the property that represents the link and primitive value or the name, type and id of the object or value at the other end of the link.

```
[{
  "type" : "/type/link",
  "source" : {
    "id" : "/en/the_police"
  },
  "master_property" : null,
  "target" : {},
  "target_value" : null
}]
```

And here is how we ask instead for the name, type, and id of all objects that are linked to The Police:

```
[{
  "type" : "/type/link",
  "target" : {
    "id" : "/en/the_police"
  },
  "master_property" : null,
  "source" : {}
}]
```

We can also query links to primitive values. The following query, for example, asks for objects that are linked to the date July 4th, 1776:

```
[{
  "type" : "/type/link",
  "target_value" : {
    "type" : "/type/datetime",
    "value" : "1776-07-04"
  },
  "master_property" : null,
  "source" : {}
}]
```

Toplevel link queries can be used in conjunction with `return:count`. This query asks: "how many links did the user "wp_typer" create to type objects as `/music/artist`?":

```
{
  "return" : "count",
  "type" : "/type/link",
  "creator" : "/user/wp_typer",
  "master_property" : "/type/object/type",
  "target" : {
    "id" : "/music/artist"
  }
}
```

3.7.3. Reflection

Reflection is a MQL feature that is closely related to links. It is a mechanism for querying the properties of an object, regardless of the type that defines those properties. The `*` wildcard described earlier in this chapter is a type-based wildcard: it queries the value of all properties defined by a type, plus the common properties defined by `/type/object`. Reflection is different: it is a link-based wildcard mechanism that queries the outgoing or incoming links of an object, regardless of the type associated with those links.

Reflection is done with `/type/reflect`. Like `/type/link`, `/type/reflect` is a pseudo-type and objects are never assigned this type. `/type/reflect` exists simply to serve as a holder for three special properties: `/type/reflect/any_master`, `/type/reflect/any_reverse`, and `/type/reflect/any_value`. These properties must always be used by their fully-qualified names. The word "any" in these property names indicates their wildcard behavior. `/type/reflect/any_master` matches any outgoing link to another object. `/type/reflect/any_reverse` matches any incoming link from another object. And `/type/reflect/any_value` matches any link to a primitive value such as `/type/text`, `/type/datetime` or `/type/float`.

The following query uses all three of these properties on The Police. The results shown are dramatically pruned to fit. Note that each of the sub-queries uses the `link` directive to ask for the name of the property that was matched (recall that the default property of links is `master_property`):

Query

```
{
  "id":"/en/the_police",
  "/type/reflect/any_master":[{
    "link":null,
    "name":null
  }],
  "/type/reflect/any_reverse":[{
    "link":null,
    "name":null
  }],
  "/type/reflect/any_value":[{
    "link":null,
    "value":null
  }]
}
```

Result

```
{
  "id" : "/en/the_police",
  "/type/reflect/any_master" : [{
    "link" : "/type/object/type",
    "name" : "Musical Artist"
  },{
    "link" : "/music/artist/genre",
    "name" : "Rock music"
  },{
    "link" : "/music/artist/origin",
    "name" : "London"
  },{
    "link" : "/common/topic/webpage",
    "name" : null
  },{
    "link" : "/music/artist/label",
    "name" : "Polydor Records"
  }],
  "/type/reflect/any_reverse" : [{
    "link" : "/music/album/artist",
    "name" : "Outlandos d'Amour"
  },{
    "link" : "/music/album/artist",
    "name" : "Reggatta de Blanc"
  },{
    "link" : "/music/track/artist",
    "name" : "Message in a Bottle"
  },{
    "link" : "/music/track/artist",
    "name" : "Can't Stand Losing You"
  }],
  "/type/reflect/any_value" : [{
    "link" : "/type/object/name",
    "value" : "The Police"
  },{
    "link" : "/common/topic/alias",
    "value" : "Police"
  },{
    "link" : "/music/artist/active_start",
    "value" : "1977-01"
  },{
    "link" : "/music/artist/active_end",
    "value" : "1986-06"
  }]
}
```

Note that the `/type/reflect/any_value` does not actually return every value of an object. The `id`, `key`, `timestamp` and `creator` properties are not matched by `/type/reflect/any_value`. Since every object has values for these properties, however, it is easy to write queries that ask for them explicitly. Furthermore, `/type/reflect/any_value` never matches *any* property whose value is a `/type/id` or a `/type/key`. In particular, this means that an `any_value` query on a namespace object will not match the /type/namespace/keys properties that identify the names in the namespace.

Another difficulty with /type/reflect/any_value is that it is tricky to ask for the lang property of text values. The expected type of any_value is /type/value which does not have a lang property. This means that you can't use the unqualified lang property. But MQL will not allow you to use the full property name /type/text/lang in a reflective query. Also, a wildcard query "*":null in any_value expands only to the type and value properties. If you're interested in reflecting on text values only, you can just do this:

```
{
  "id":"/en/the_police",
  "/type/reflect/any_value": [{
    "type":"/type/text",
    "link":null,
    "value":null,
    "lang":null
  }]
}
```

But if you want to query the language id of text values without restricting your reflective query to return only text values, you must do something like this:

```
{
  "id" : "/en/the_police",
  "/type/reflect/any_value" : [{
    "*" : null,
    "link" : {
      "master_property" : null,
      "target" : {
        "id" : null,
        "optional" : true
      }
    }
  }]
}
```

We'll end this discussion of reflection with a more advanced query. It asks for the id and type of objects that have outgoing links objects named The Police and Sting. Two results are shown, including the important /music/group_membership object that specifies that Sting was a member of The Police:

Query

```
[{
  "id":null,
  "type":[],
  "first:/type/reflect/any_master": {
    "name":"Sting",
    "link":null
  },
  "second:/type/reflect/any_master": {
    "name":"The Police",
    "link":null
  }
}]
```

Result

```
[{
  "id":"/guid/1f8000000003924426",
  "type":["/music/group_membership"],
  "first:/type/reflect/any_master": {
    "link":"/music/group_membership/member",
    "name":"Sting"
  },
  "second:/type/reflect/any_master": {
    "link":"/music/group_membership/group",
    "name":"The Police"
  }
},{
  "id":"/user/saraw524",
  "type": [
    "/type/user",
    "/type/namespace",
    "/freebase/user_profile"
  ],
  "first:/type/reflect/any_master": {
    "link":"/freebase/user_profile/favorite_music_artists",
    "name":"Sting"
  },
  "second:/type/reflect/any_master": {
    "link":"/freebase/user_profile/favorite_music_artists",
    "name":"The Police"
  }
}]
```

A similar query could be written to find objects that link to both Arnold Schwarzenegger and Maria Shriver: it would find objects representing their marriage and their children.

3.7.4. History

Objects in a Metaweb database live forever: once created they can never be deleted. The closest we can come to deleting an object is to remove all links from and to it. Somewhat more surprising is the fact that links live on forever in Metaweb, too. A link may be deleted or replaced with a new value, but the historical existence of that link is retained. We've already seen that we can query the creation timestamp of any object or any link. But there is another kind of history query we can express with MQL as well. The `valid` property of a link specifies whether the link is currently valid or not. The following query, for example, finds the most recently invalidated link between an object and a name. The `target_value` property returns the old name of the object, and `source.name` returns the name that has replaced it.

```
[{
  "type" : "/type/link",
  "valid" : false,
  "master_property" : "/type/object/name",
  "source" : {},
  "target_value" : null,
  "limit" : 1,
  "timestamp" : null,
  "sort" : "-timestamp"
}]
```

It is important to understand that MQL link queries normally only return valid links. You can explicitly request links that are no longer valid with `"valid":false`, and you can request links that are either valid or invalid with `"valid":null` or by using a wildcard like `"link":{"*":null}`. All other link queries (with one exception involving the `operation` property that we'll learn about below) are made with an implicit `"valid":true`. As an example, consider the following query for the name of the `/finance/currency` object and the date that it was given its name:

```
Query                         Result
{                             {
  "id" : "/finance/currency",   "id" : "/finance/currency",
  "name" : {                    "name" : {
    "value" : null,               "value" : "Currency",
    "link" : {                    "link" : {
      "timestamp" : null            "timestamp" : "2007-03-25T00:33:28.0000Z"
    }                             }
  }                             }
}                             }
```

The name of the object is "Currency" and it has been since March 25th, 2007. Now let's alter the query slightly to ask for links of any validity:

```
Query                         Result
{                             {
  "id" : "/finance/currency",   "id" : "/finance/currency",
  "name" : [{                   "name" : [{
    "value" : null,               "value" : "currency",
    "link" : {                    "link" : {
      "valid" : null,               "valid" : false,
      "timestamp" : null            "timestamp" : "2006-10-22T07:34:51.0008Z"
    }                             }
  }]                            },{
}                                 "value" : "Currency",
                                  "link" : {
                                    "valid" : true,
                                    "timestamp" : "2007-03-25T00:33:28.0000Z"
                                  }
                                }]
                              }
```

This query tells us that on October 22nd, 2006 the `/finance/currency` object was given the name "currency" (with a lowercase c), but that that name was changed to "Currency" on March 25th of the following year. Note that the name query is now written using square brackets. When querying link history, we must use square brackets even with unique properties because the property may have had more than one value over time.

Metaweb makes a record of every insertion, deletion and update of a link, and the `operation` property of a link allows us to use this in queries. The possible values of this property are "insert", "delete" and "update". Links that correspond to unique properties (including the `name` property) can be inserted, deleted, or updated. Links that correspond to non-unique properties, however, are never updated: they can only be inserted and deleted.

We use the `operation` property in the following query to find types that have been deleted (that is: objects that used to be linked to `/type/type`, but are no longer) and also ask when they were deleted and by whom. (Note that the `creator` property of a link can also refer to the user who deleted or updated a link as well).

```
[{
  "type" : "/type/link",
  "operation" : "delete",
  "master_property" : "/type/object/type",
  "source" : {},
  "target" : { "id" : "/type/type" },
  "timestamp" : null,
  "creator":null,
}]
```

Note that this query explicitly asks for links that have been deleted. This means that it returns invalid links even though it does not include `"valid":null` or `"valid":false`. If you merely query the `operation` property with `"operation":null`, you will not get invalid links unless you also query or constrain the `valid` property. If you query the `operation` property without using the `valid` property, the results you get will only include insertions and updates, not deletions, because a link that has been deleted is, by definition, no longer valid. The `valid` property of a deleted link is actually `null`, not `false`, so writing a query for links that are invalid will only return links that have been updated, not those that have been deleted. To find deleted links, you must explicitly use `"operation":"delete"`.

Here's a complex query that asks for type objects that have had their English names changed. It also asks when those changes were made, and by who. It tells us, for example, that the type `/location/province` was originally named "Province", but that `/user/colin` updated the name to "Canadian Province" in January 2007. Then `/user/jeff` updated the name to "CA Province" in August 2007 and then updated it back to "Canadian Province" in November 2007. Rather than using a top-level `/type/link` query, this query uses two different `link` directives to find both the original insertion of the name and also all subsequent updates to the name.

```
[{
  "type":"/type/type",
  "id":null,
  "original:name":[{
    "value":null,
    "link": {
       "operation":"insert",
       "valid":false,
       "timestamp":null,
       "creator":null
    }
  }],
  "new:name": [{
    "value":null,
    "link": {
        "operation":"update",
        "valid":null,
        "timestamp":null,
        "creator":null
      }
    }]
  }]
```

Note again that the name queries are surrounded by square brackets because historical queries must be expected to return multiple results. This query does not ask about name deletions: it assumes that names are changed by updates rather than deletions and re-insertions. We can write a simpler and more general query to ask about the complete name history of type objects:

```
[{
  "type" : "/type/type",
  "id" : null,
  "name" : [{
    "value" : null,
    "sort" : "link.timestamp",
    "link" : {
      "valid" : null,
      "operation" : null,
      "creator" : null,
      "timestamp" : null
    }
  }]
}]
```

The problem with this query is that it matches any type with a name, even types that have never had the name changed. We can fix this by requiring that the type object have at least one currently invalid name link:

```
[{
  "type" : "/type/type",
  "id" : null,
  "number_of_invalid:name" : [{
    "return" : "count",
    "link" : { "valid" : false  }
  }],
  "name" : [{
    "value" : null,
    "sort" : "link.timestamp",
    "link" : {
      "valid" : null,
      "operation" : null,
      "creator" : null,
      "timestamp" : null
    }
  }]
}]
```

4

Metaweb Read Services

Chapter 3 explained how to express Metaweb queries using MQL. This chapter explains how to deliver those queries to Metaweb servers and retrieve their response using the *mqlread* service. It also explains how to search Metaweb with the *search* service and how to retrieve chunks of data (such as images and HTML documents) using the *trans* service. The chapter includes example applications and libraries written in Perl, Python, PHP, and JavaScript and concludes with a sophisticated Python library for interacting with Metaweb's read services.

4.1. Basic mqlread Queries with Perl

Metaweb's services are all implemented on top of the HTTP protocol. Submitting a MQL query and retrieving the response, therefore, is simply a matter of constructing the appropriate URL and fetching its content via an HTTP request.

The basic URL for submitting MQL queries to *freebase.com* is:

```
http://api.freebase.com/api/service/mqlread
```

To submit a query to the *mqlread* service, follow these steps:

- Place the query inside an "envelope" object.
- Serialize the envelope object to a JSON string.
- Escape punctuation characters in the JSON string using standard URL encoding.
- Concatenate the basic URL above with "?query=" and the escaped and encoded JSON string to form a complete mqlread URL.
- Fetch the contents of the URL with an HTTP GET request.

Example 4.1 is a command-line utility that lists the albums released by any band you specify. It uses the Metaweb API to retrieve data from *freebase.com*. It is written in Perl, and demonstrates how to nest an MQL query within an envelope and send that envelope to to the *mqlread* service. (The structure of the envelope object will be explained in §4.2.1.)

Example 4.1. albumlist.pl: submitting MQL queries in Perl

```
#!/usr/bin/perl
use URI::Escape;  # This module provides the uri_escape function used below.

# Build the Metaweb query, using string manipulation.
# CAUTION: the use of string manipulation here makes this script vulnerable
# to MQL injection attacks when the command-line argument includes JSON.
$band = $ARGV[0]; # This is the band or musician whose albums are to be listed
$query='{"type":"/music/artist","name":"' . $band . '","album":[]}';

# Now place the query in a JSON envelope, and URL encode the envelope.
$envelope = '{"query":' . $query . '}';
$escaped = uri_escape($envelope);

# Construct the URL that represents the query.
$baseurl='http://api.freebase.com/api/service/mqlread'; # Base URL for queries
$url = $baseurl . "?query=" . $escaped;

# Use the command-line utility curl to fetch the content of the URL.
$result = `curl -s $url`;

# Use regular expressions to extract the album list from the HTTP response.
$result =~ s/^.*"album"\s*:\s*\[\s*([^\]]*)\].*$/$1/s;
$result =~ s/[ \t]*"[ \t,]*//g;

# Finally, display the list of albums.
print "$result\n";
```

You run this program from the command line. An invocation might look like this:

```
$ perl albumlist.pl 'Spinal Tap'
Break Like the Wind
This Is Spinal Tap
The Majesty of Rock
Smell the Glove
```

4.1.1. A Better Perl Album Lister

The first thing to notice about Example 4.1 is that it does not use a JSON serializer or parser: a JSON-encoded MQL query is constructed with string concatenation and the desired results are extracted with regular expressions. These shortcuts keep the example simple and allow us to focus on how the *mqlread* URL is built and its content fetched. More sophisticated applications, however, use a JSON encoder to serialize the query and a JSON decoder to parse the result. Building queries with string manipulation can be reasonable in the simplest applications (though caution is required to avoid MQL injection attacks), but attempting to extract results with regular expressions is brittle and not a technique to emulate in your own code!

Example 4.2 is a higher-level version of Example 4.1. It uses a JSON serializer and parser and also a higher-level API for URL manipulation. To use it, you must have the JSON.pm module (version 2 or higher) installed[1]. This version of the program also uses a somewhat more sophisticated query to sort albums by their release date, and also does error checking and error reporting in case anything goes wrong with the query. Finally, it also adds an additional input parameter to the query envelope to specify that HTML escaping should not be done on the query results. With this option, ampersands in album names are returned as `&` rather than as `&amp;`, which is what we want since this application runs in a terminal rather than in a browser.

Notice that the *mqlread* service returns the query results in a response envelope object that is similar to the query envelope. This response envelope has a property named `result` whose value is the result of the MQL query.

Example 4.2. albumlist2.pl: a better Perl album lister

```
#!/usr/bin/perl -w
use strict;           # Don't allow sloppy syntax
use JSON;             # JSON encoding and decoding with to_json and from_json
use URI::Escape;      # URI encoding with uri_escape
use LWP::UserAgent;   # High-level HTTP API

# Some constants for this script
my $SERVER = 'http://api.freebase.com';            # The Metaweb server
my $QUERYURL = $SERVER . '/api/service/mqlread';   # Path to mqlread service

# What band did the user ask about?
my $band = $ARGV[0];

# Construct a Metaweb query as a Perl data structure.
my $query =  {
    type => "/music/artist",    # We're looking for a band.
    name => $band,              # This is the name of the band.
    album => [{                 # Return some albums.
        name => undef,          # undef is Perl's null.
        sort => "release_date", # Sort by release date.
        release_date => undef   # Return release date, too.
    }]
};

# Put the query in an envelope object.
my $envelope = {
    query => $query,             # The "query" property holds the query.
    escape => JSON::false        # Don't HTML escape result text
};

# Convert the envelope object from Perl hash to JSON string, and URI encode it.
my $encoded = to_json($envelope);    # Serialize object to string.
my $escaped = uri_escape($encoded);  # URI encode the string.

# Build the complete query url.
my $url = $QUERYURL . "?query=" . $escaped;
```

[1]Find it at *http://search.cpan.org* or install it automatically by running `# cpan -i JSON`

```
# Create the HTTP "user agent" we'll use to send the query.
my $ua = LWP::UserAgent->new;

# Send request to the server and get the response.
my $response = $ua->get($url);

# Now handle the mqlread response.
if ($response->is_success) {                      # If we get HTTP 200 OK...
    my $responsetext = $response->content;        # Get result as JSON text.
    my $response = from_json($responsetext);      # Parse text to a Perl hash.

    if ($response->{code} ne "/api/status/ok") {  # If the query was not okay:
        my $err = $response->{messages}[0];       # get the error message obj
        die $err->{code}.': '.$err->{message};    # and exit with error message
    }

    # If there was no error, the MQL result is in the response envelope
    my $result = $response->{result};   # Open response envelope, get result.
    my $albums = $result->{album};      # Get albums array from result.
    for my $album (@$albums) {          # Loop through albums.
        print "$album->{name}";         # Print the name of each.
        if ($album->{release_date}) {   # Print release date, if there is one.
            print " [" . substr($album->{release_date},0,4) . "]"; # Year only.
        }
        print "\n";                     # Add a newline.
    }
}
else {                                  # If query failed...
    die "Server returned error code " . $response->code . "\n";
}
```

4.2. The mqlread Service

Now that we've seen some working code, this section explains more formally how *mqlread* works. Like all Metaweb services, *mqlread* is a web-based service: it takes an HTTP request as input and returns an HTTP response as its output.

The path to the *mqlread* service on a Metaweb server is */api/service/mqlread*. To send a *mqlread* query to the Metaweb server running at *api.freebase.com*, for example, you'd use the following URL:

```
http://api.freebase.com/api/service/mqlread
```

mqlread works with both GET and POST request methods. GET requests are preferred unless the query is so long that POST must be used instead.

There are two sources of input to *mqlread* in an HTTP request. The first is the request parameters. For GET requests, these parameters are encoded in the URL itself, following a ? character. For POST requests, the request parameters appear in the body of the request. In both cases, the parameters are URI encoded in the standard way that web browsers encode HTML form submissions. *mqlread* recognizes request parameters `query`, `queries` and `callback`, and these are documented in sub-sections below. Every *mqlread* request must include either the `query` or `queries` request parameters (but not both). The value of these parameters is a JSON-serialized object known as a *query envelope*. In addition to holding the actual MQL query that is being submitted, this envelope object may also hold additional *mqlread* input in the form of "envelope parameters". Envelopes and envelope parameters are covered in detail below.

The second source of *mqlread* input in an HTTP request is HTTP cookies. *mqlread* looks for a cookie named `mwLastWriteTime`. This cookie is only necessary in applications that perform MQL writes as well as MQL reads, and ensures that recent writes are always visible to subsequent read requests performed by the same application (or by the same web browser). The `mwLastWriteTime` cookie is covered in Chapter 6 rather than in this chapter. In general, Metaweb-enabled applications need not track individual cookies. Instead, they can behave like web browsers do: any cookies returned as output by a Metaweb service should be included as input to subsequent requests.

The output of the *mqlread* service is the HTTP response body. This body is always (even when errors occur because of bad input) a JSON serialized object in text/plain encoding. This JSON serialized object is known as a *response envelope*, and it is explained in §4.2.3 below.

4.2.1. The query Request Parameter

The simplest *mqlread* request includes a single request parameter named `query`. The value of this parameters is a JSON-serialized object known as a query envelope. This envelope object must have a property named `query`, and may also have additional properties that specify "envelope parameters" – see §4.2.4. The value of the `query` property in the envelope is your JSON-serialized MQL query. Thus a simple *mqlread* invocation uses a URL like this:

```
http://api.freebase.com/api/service/mqlread?query={"query":[{Your MQL here}]}
```

Notice that the word "query" appears twice in this URL. The first (without quotes) is the request parameter, and the second (with quotes) is a property of the envelope object. In this example, the square brackets are part of the MQL query itself, not part of the envelope syntax. Note also that everything after the equals sign should, in practice, be URI-encoded, which transforms quotation marks into `%22`, and so forth.

4.2.2. The queries Request Parameter

mqlread allows you to submit more than one MQL query at a time. To do this, you must use the `queries` request parameter instead of `query`. (Every invocation of *mqlread* must include one or the other, but not both.) The value of the `queries` parameter is not a simple query envelope as it is for the `query` parameter. Instead, it is a JSON-serialized object known as an *outer envelope*. The outer envelope contains named query envelopes. To submit two queries at once the outer envelope would have two properties. The names of the properties might be `q1` and `q2`, and their values would be the two query envelopes that describe the two queries to be run:

```
{                                     # Start the outer envelope
  "q1": {                             # Query envelope for query named q1
    "query":{First MQL query here}    # Query property of query envelope
  },                                  # End of first query envelope
  "q2": {                             # Start query envelope for query q2
    "query":[{Second MQL query here}] # Query property of q2
  }                                   # End of second query envelope
}                                     # End of outer envelope.
```

The property names used within an outer envelope are arbitrary, but they appear again in the *mqlread* response.

4.2.3. The Response Envelope

The output of the *mqlread* service is an HTTP response, which consists of a set of HTTP response headers and a response body. The headers are not typically interesting (though Metaweb engineers might be interested in the X-Metaweb-TID header if you're submitting a bug report). In particular *mqlread* is not expected to return cookies as part of its output.

The body is the interesting part of the *mqlread* response. It is a UTF-8 encoded JSON-serialized object. This object is known as the response envelope, and its structure mirrors that of the query envelope with the `query` property replaced with a `result` property. If the request used the `query` parameter to submit a single query envelope, then the result is a single response envelope. The following are side-by-side views of a query and response envelope:

Query Envelope	**Response Envelope**

`{`
`  "query": [{ MQL Query Here }]`
`}`
	`{`
	`  "result": [{ MQL Response Here }],`
	`  "status": "200 OK",`
	`  "code": "/api/status/ok",`
	`  "transaction_id":[opaque string value]`
	`}`

If the request used the `queries` parameter to submit multiple named query envelopes within an outer envelope, then the response is an outer envelope that uses the same names to refer to multiple response envelopes:

Query Envelopes	Response Envelopes

`{`
`  "q1": {`
`    "query":{First MQL query here}`
`  },`
`  "q2": {`
`    "query":[{Second MQL query here}]`
`  }`
`}`
	`{`
	`  "q1": {`
	`    "result":{First MQL result here},`
	`    "code": "/api/status/ok"`
	`  },`
	`  "q2": {`
	`    "result":[{Second MQL result here}],`
	`    "code": "/api/status/ok"`
	`  },`
	`  "status": "200 OK",`
	`  "code": "/api/status/ok",`
	`  "transaction_id":[opaque string value]`
	`}`

Notice that response envelopes include `code`, `status`, and `transaction_id` properties. The `code` property is the most important: it specifies a success or failure status code (as a string) for the query. If the query was successful then the value of this property will be "/api/status/ok". If `code` does not equal "/api/status/ok", then there was an error of some sort, and the response envelope will include additional details in a `messages` array. See §4.2.6 for further details about status codes and error messages, including details on the `status` and `transaction_id` properties.

4.2.4. Envelope Parameters

If you use the `query` request parameter you specify a single query envelope as its value. If you use `queries` instead, you specify one or more query envelopes within an outer envelope. In either case, each query envelope must include a property named `query` that specifies the MQL query to be executed. Each envelope may also include additional properties, known as "envelope parameters" that provide additional input to *mqlread* and specify how the query should be run. *mqlread* supports envelope parameters named `cursor`, `escape`, `lang` `as_of_time` and `uniqueness_failure`. These parameters are described below.

4.2.4.1. Fetching Large Result Sets with Cursors

Recall that MQL queries are implicitly limited to returning 100 results. You can use the MQL `limit` directive to specify a different limit, but when there is a very large result set, specifying a very large limit may cause your query to time out. When you expect that your query will have many results, and you want to retrieve all of those results, you should use a *cursor* [2]. A cursor is simply a way of keeping track of your position within a large set of results, and it enables you to retrieve the results of a query batch by batch with multiple sequential *mqlread* invocations. Cursors are demonstrated later in this chapter in §4.8.

[2] A Metaweb cursor is related to, but not the same as a cursor used in SQL with a relational database.

To begin a new query that uses a cursor, include a `cursor` property with the value `true` in the query envelope. The response envelope (see §4.2.3 will then contain a `cursor` property. If the value of the `cursor` in the response is `false`, it means that all results have been delivered. Otherwise, that property will be a long string of opaque data. Use this string as the value of the `cursor` property in the query envelope, and submit that query envelope again (leaving the query itself unchanged). This time the response envelope will contain the second batch of results and a new value for the `cursor` property. Repeat these steps until the `cursor` property of the response envelope is `false`.

It is important to understand that cursors only work when multiple results are expected at the top-level of the query. The `cursor` property is part of the `mqlread` query envelope syntax, not part of MQL itself, and it cannot be applied to sub-queries of a query. Another way to say this is that it only makes sense to include `"cursor":true` in an envelope if the first character following `"query":` in the envelope is `[`. The query must be expressed as an array in order for a cursor to be meaningful. It is legal, but never useful, for example, to use a cursor in this query envelope:

```
{
  "cursor":true,
  "query": {
    "type":"/music/artist",
    "name":"The Police",
    "album":[]
  }
}
```

If you want to use a cursor to retrieve a list of albums in batches, the array of albums must be at the toplevel of the query:

```
{
  "cursor":true,
  "query": [{
    "type":"/music/album",
    "artist":"The Police",
    "name":null,
    "limit":10
  }]
}
```

Note the addition of the `limit` directive to the query to specify the size of each batch we want returned.

Cursor values remain valid after they are used. Once you have downloaded result batches sequentially, you can reuse saved cursor values to download those batches again in whatever order you like. (Except for the first batch which does not have a cursor.) Results retrieved with cursors are based on the state of the database as it existed when the first query (with `"cursor":true`) was issued. [3] So results retrieved with a given cursor will always be the same, and will ignore any insertions or deletions that occurred after the original query. Cursors can be assumed to have a

[3] See the discussion of the `as_of_time` envelope parameter in §4.2.4.4 for an explanation of how past results can be retrieved. The `as_of_time` parameter also allows you to re-retrieve the first batch of query results even though the first batch does not have a cursor value.

lifetime at least as long as that of your application. But updates to Metaweb's backend software can invalidate cursors, so you should not assume that they live forever. Cursors are not intended to be stored in databases or files or encoded into long-lived URLs, for example. They should not be considered "permalinks" or persistent bookmarks to a past state of the database.

4.2.4.2. Disabling HTML Escapes

By default, *mqlread* uses HTML entities &lt;, &gt;, and &amp; in its responses in place of the characters <, >, and &, and this means that text returned by *mqlread* is safe for display in web browsers. To disable this escaping, add an `escape` parameter to the query envelope, and set its value to `false`. (You can also explicitly request HTML escaping with `"escape":"html"`, but this is the default behavior and is not required.) If you do disable HTML escaping, you should be careful never to display the *mqlread* output in a web browser, since it could contain `<script>` tags that execute arbitrary JavaScript code, for example.

The `escape` envelope parameter was demonstrated in Example 4.2 and we'll see it again in Example 4.3.

4.2.4.3. Specifying Your Preferred Language

As you know, Metaweb objects can have more than one value for the `name` property, but can have only one value in any given language. When you request the name of an object, it returns the name in your preferred language. The default language is English, but you can specify a different preference with the envelope parameter `lang`. The value of this parameter should be a language id in the `/lang` namespace. The following query envelope, for example, asks for the Spanish name of the French language:

```
{
  "lang":"/lang/es",
  "query": {
    "id":"/lang/fr",
    "name":null
  }
}
```

At the time of this writing, [4] *mqlread* supports only a single preferred language. If no name exists in the specified language, then `null` is returned. In the future, the `lang` envelope parameter is likely to evolve to support language fallbacks, and you should be able to request, for example, a name in Spanish, or in English if no Spanish name exists.

[4]September, 2008

4.2.4.4. Making Queries in the Past

Use the `as_of_time` envelope parameter in a *mqlread* query to specify that the query should be performed historically, against the Metaweb database as it existed at the specified moment in the past. The Metaweb database has a journaled structure, making this kind of historical query relatively simple and efficient to perform. Note, however, that type and schema information used in processing the query is current rather than historical.

The value of the `as_of_time` parameter should be a timestamp in the ISO 8601 format (see §2.5.8) used by `/type/datetime` values in MQL. For example, the value "2007-02-03" represents midnight on February 3rd, 2007, and the value "2008-01-01T17:00Z" represents 5PM on January 1st, 2008. Metaweb timestamps are always stored in UTC (or GMT) time, and the `as_of_time` parameter assumes that your timestamp is specified in UTC. You may append Z to your timestamp to make this timezone explicit but you may not explicitly specify any other timezone. That is, you cannot add -08:00 to specify US Pacific time, for example.

Here are two queries (in their query envelopes, and named "now" and "then" in an outer envelope) that allow us to see how the number of defined types has grown over time:

```
{
  "now": {
    "query": {
        "return":"count",
        "type":"/type/type"
    }
  },
  "then": {
    "query": {
        "return":"count",
        "type":"/type/type"
    },
    "as_of_time":"2008-01-01"
  }
}
```

On 2008-04-25, freebase.com returned the following outer response envelope, showing the addition of over 900 types in under 5 months:

```
{
  "status" : "200 OK",
  "code" : "/api/status/ok",
  "now" : {
    "code" : "/api/status/ok",
    "result" : 5548
  },
  "then" : {
    "code" : "/api/status/ok",
    "result" : 4623
  }
}
```

In addition to the ability to run queries "in the past", Metaweb allows you to query the modification history of any object. See §3.7.4 for details.

4.2.4.5. Preventing Uniqueness Errors

When a MQL query or sub-query returns more than one result, but is not enclosed in square brackets to indicate that an array of results is expected, Metaweb normally returns an error code indicating that a uniqueness error has occurred. We saw in §3.2.4, for example, that the following query causes an uniqueness error because the object representing The Police has more than one type:

```
{"id":"/en/the_police", "type":null}
```

You can prevent this kind of error by setting the envelope parameter `uniqueness_failure` to "soft". (The default value is "hard"). With this parameter set to "soft", Metaweb simply returns one of the matching results, discards the others, and does not return an error or give any other indication that additional results are available.

Picking one (effectively random) result from a set and discarding all the others is not usually a useful strategy for handling multiple results, and the `uniqueness_failure` envelope parameter is not intended for use with queries like the one above. As a general rule, if a property is allowed to have more than one value, then queries of that property should be placed within square brackets.

When a property definition is changed to make the property unique after it is initially defined as non-unique, then it is possible (but rare) to find multiple values for a nominally unique property. You may want to use the `uniqueness_failure` envelope parameter when working with such a theoretically-unique property that is not yet unique in practice.

4.2.5. The callback Request Parameter

Every *mqlread* request must have a `query` or `queries` request parameter. They may also optionally have a `callback` parameter (this is a request parameter, not an envelope parameter). This parameter is used in JavaScript-based Metaweb applications and allows *mqlread* to be invoked using dynamically-generated `<script>` tags. (This `<script>`-based technique for client-server communication is commonly known as JSONP and we'll see examples in §4.5.)

The value of the `callback` parameter should be the name of a JavaScript function (without parentheses), such as `processMQLResponse`. Including the `callback` parameter in a *mqlread* invocation causes a small but very important change in the *mqlread* output. In order to understand these changes, recall that *mqlread* always returns a JSON-serialized object in the HTTP response body. Also recall that JSON is a subset of JavaScript which means that any JSON-serialized object that can be parsed by a JavaScript interpreter to re-create the object it represents. If you include a `callback` parameter in a *mqlread* invocation, *mqlread* returns a JSON-serialized object inside a JavaScript function invocation of the callback function you specify. Suppose, for example, that you invoke *mqlread* with a URL like this:

```
http://api.freebase.com/api/service/mqlread?callback=cb&query={"query":[{...}]}
```

In this case, the response will look like this:

```
cb({
  "status":"200 OK",
  "code":"/api/status/ok",
  "result":[{...MQL result here...}]
})
```

The JSON-serialized result envelope object is prefixed with the name of the callback function and an open parenthesis and is suffixed with a matching close parenthesis. This seems like a trivial change, but it transforms a bare JSON object into a JavaScript method invocation with that object as its argument. If the *mqlread* query URL is used as the `src` attribute of a `<script>` tag, the *mqlread* response becomes executable JavaScript content, and the callback function you name gets passed the response envelope object to process however it wants.

There is one other effect of using the `callback` parameter in a request. It forces `mqlread` to always return an HTTP status code of "200 OK", even when the query envelope is malformed and unparseable. The true HTTP status (which would otherwise have been returned) is available from the `status` property of the response envelope, and this enables a callback function to handle errors as well as successful queries.

See §4.5 for practical examples that use the `callback` request parameter.

4.2.6. mqlread Error Codes

The *mqlread* response envelope does not always include the results of your query. Errors can occur if you invoke *mqlread* incorrectly, if you specify an invalid MQL query, if your query times out, or if there if there is an internal error on the Metaweb server. This section explains how to check for and handle *mqlread* errors.

If you invoke *mqlread* without either the `query` or `queries` parameter, or if the value of that parameter is not a valid JSON string, it responds with an HTTP status "400 Bad Request". Even though this is an HTTP error code, the response still includes a response body and that body is still a JSON object that you can parse. It looks something like this:

```
{
  "status": "400 Bad request",
  "code": "/api/status/error",
  "messages": [
    {
      "info": {
        "value": null
      },
      "message": "one of query=, or queries= must be provided",
      "code": "/api/status/error/input/invalid"
    }
  ],
  "transaction_id":"cache;cache01.sandbox.sfo1:8101;2008-09-12T21:33:30Z;0001"
}
```

If you invoke *mqlread* with correct parameters and a parseable query envelope, then it will always return an HTTP status code of "200 OK". This does not mean, however, that no error has occurred. If the query envelope is valid JSON but does not have a `query` property, for example, then you get this response:

```
{
  "status": "200 OK",
  "code": "/api/status/error",
  "messages": [
    {
      "info": {
        "key": "query"
      },
      "message": "Missing 'query' parameter",
      "code": "/api/status/error/envelope/parse"
    }
  ],
  "transaction_id":"cache;cache01.sandbox.sfo1:8101;2008-09-12T21:35:19Z;0001"
}
```

And if the invocation is correct and the envelope is correct but the MQL query is invalid, then you might get a response like this:

```
{
  "status": "200 OK",
  "code": "/api/status/error",
  "messages": [
    {
      "info": {
        "expected_type": "/music/artist",
        "property": "albums"
      },
      "query": {
        "albums": [],
        "type": "/music/artist",
        "id": "/en/the_police",
        "error_inside": "."
      },
      "message": "Type /music/artist does not have property albums",
      "code": "/api/status/error/mql/type",
      "path": ""
    }
  ],
  "transaction_id":"cache;cache01.sandbox.sfo1:8101;2008-09-12T21:36:51Z;0001"
}
```

If your query is simply too big (such as asking for the names and discographies of 10,000 bands) the query will timeout and *mqlread* will return a response like this:

```
{
  "status": "200 OK",
  "code": "/api/status/error",
  "messages": [
    {
      "info": {
        "detail": [
          "timed out"
        ],
        "timeout": 8.0
      },
      "message": "Query timeout",
      "code": "/api/status/error/mql/timeout"
    }
  ],
  "transaction_id":"cache;cache01.sandbox.sfo1:8101;2008-09-12T21:39:01Z;0004"
}
```

Each of these error response envelopes includes the properties `status`, `transaction_id`, `code`, and `messages`. The `status` property simply repeats the HTTP status code. If you check the HTTP status code, you can ignore the `status` property. But if you use the `callback` parameter in your request, then *mqlread* will return a HTTP status of "200 OK", even when invocation errors occur. In this case the `status` property can tell you that an invocation error occurred.

The `transaction_id` property is always present in the response envelope whether or not an error occurred. Its value is a unique identifier for your request and enables Metaweb engineers to look it up in their internal log files. You should include the value of this property whenever you report a bug or ask a question about a query on the Metaweb developers mailing list.

The `code` property specifies the error code for the query. It is present in every response envelope, and you should always check this property after parsing the response from *mqlread*. The value of this property is always a string: if it is "/api/status/ok", then the query was successful and no error occurred. Otherwise something went wrong.

When the value of the `code` property is "/api/status/ok", the response envelope contains the query results as the value of a property named `result`. When `code` has any other value, the response envelope includes a `messages` property instead of a `result` property. The value of the `messages` property is an array (usually of length 1) of message objects each of which has the following properties:

`code`	A more detailed error code that more precisely specifies the nature of the error. Note that this `code` property of the message object is usually distinct from, and more informative than, the `code` property of the response envelope.
`message`	A human-readable description of the error.
`info`	An object that provides additional details about the error. The properties of this object depend on the error `code`.

`query` For errors that result from an invalid MQL query, this property is a copy of the query object with the addition of a special `error_inside` property, to indicate where error occurs.

`path` When the message object contains a `query` property, it also contains a `path` property that specifies the "path" of property names from the root of the MQL query to the the location of the error. This is an alternative to the `error_inside` property for locating the source of the error. If the error is in the outermost object of the query, then this property is just an empty string.

The descriptions above of error-related properties are valid when you use the `query` request parameter. If you use `queries` instead, there are a few differences you should be aware of. First, the `status` property appears only in outer envelope, not in the individual response envelopes it contains. Second, there are `code` properties both in the outer envelope and in the individual response envelopes. The outer `code` will only indicate an error, however, if there was an invocation error (such as a unparseable query envelope), and in this case there won't be response envelopes inside the outer envelope. As long as *mqlread* is correctly invoked, the `code` property of the outer envelope will be "/api/status/ok", even if there were errors in one or more (or all) of the queries. It is the `code` property of the individual response envelopes that specify the status of each individual query. Third (and this really follows from the second), the `messages` property only appears in the outer envelope for invocation errors. Otherwise, `messages` properties always appear within the response envelopes.

Example 4.2 included example code for *mqlread* error handling, and many examples that follow will also include error handling code. The general rule is to check the `code` property of any response envelope before "opening" it to extract the `result`. (And if the response envelope is inside an outer envelope, you must also check the `code` of that outer envelope before opening it to extract the response envelope.) If you find a `code` that is not "/api/status/ok", you can typically construct a suitable error message with `messages[0].code` and `messages[0].message`. If you have reason to expect a certain class of errors, you can refine your error reporting based on `messages[0].code` and `messages[0].info`.

4.3. A Python Album Lister

Now that we've seen how *mqlread* works in more formal detail, let's return to example code, and re-write our album listing script in Python. Example 4.3 is a Python program that:

- expresses a MQL query as a Python data structure
- wraps the query in an envelope object;
- sets the value of envelope parameters in the envelope object;
- serializes the envelope object to a JSON string;
- URI encodes the serialized envelope;
- Uses the serialized and encoded envelope as the value of the `query` parameter in a *api.free-base.com* URL;
- obtains the query result, in text form, by fetching the contents of the URL;
- parses the JSON string returned by *mqlread* into a response envelope object;
- checks the `code` property in the response envelope to determine if the query was successful (if not, it extracts the error message from the envelope, prints the message and exits);
- gets the query result from the envelope, extracts the array of albums, and prints their name and release dates.

This code relies on the `simplejson` module for JSON encoding and parsing. You can find the simplejson code at *http://cheeseshop.python.org/pypi/simplejson*.

Example 4.3. albumlist.py: listing albums with Python

```
import sys           # Command-line arguments, etc.
import simplejson    # JSON encoding.
import urllib        # URI encoding.
import urllib2       # High-level URL content fetching.

# These are some constants we'll use.
SERVER = 'api.freebase.com'             # Metaweb server
SERVICE = '/api/service/mqlread'        # Metaweb service

# Compose our MQL query as a Python data structure.
# The query is an array in case multiple bands share the same name.
band = sys.argv[1]                      # The desired band, from command line.
query = [{'type': '/music/artist',      # Our MQL query in Python.
          'name': band,                 # Place the band in the query.
          'album': [{ 'name': None,     # None is Python's null.
                      'release_date': None,
                      'sort': 'release_date' }]}]
```

```
# Put the query in an envelope
envelope = {
    'query': query,              # The query property specifies the query.
    'escape': False              # Turns off HTML escaping.
    }

# These five lines are the key code for using mqlread
encoded = simplejson.dumps(envelope)           # JSON encode the envelope.
params = urllib.urlencode({'query':encoded})    # Escape request parameters.
url ='http://%s%s?%s' % (SERVER,SERVICE,params) # The URL to request.
f = urllib2.urlopen(url)                        # Open the URL as a file.
response = simplejson.load(f)                   # Read and JSON parse response.

# Check for errors and exit with a message if the query failed.
if response['code'] != '/api/status/ok':                   # If not okay...
    error = response['messages'][0]                        # First msg object.
    sys.exit('%s: %s' % (error['code'], error['message'])) # Display code,msg.

# No errors, so handle the result
result = response['result']           # Open the response envelope, get result.

# Check the number of matching bands
if len(result) == 0:
    sys.exit('Unknown band')
elif len(result) > 1:
    print "Warning: multiple bands named " + band + ". Listing first only."

result = result[0]                    # Get first band from array of matches.
if not result['album']:               # Exit if band has no albums
    sys.exit(band + ' has no known albums.')

for album in result['album']:         # Loop through the result albums.
    name = album['name']              # Album name.
    date = album['release_date']      # Release date timestamp or null.
    if not date: date = ''
    else: date = ' [%s]' % date[0:4]  # Just the 4-digit year in brackets.
    print "%s%s" % (name, date)       # Print name and date.
```

4.4. A Metaweb-enabled PHP Application

In this section, we'll demonstrate how to create an online version of our album-lister application. We'll use the the server-side scripting language PHP to create the web application that was shown in Figure 1.2 of Chapter 1. Example 4.4 is a PHP file that defines a class named `Metaweb`. This class has a single method, named `read` that takes a MQL query as a PHP data structure and returns the query result as a PHP data structure. Although the language and library details differ, the code that implements this `read` method is much like the code you've seen in Example 4.2 and Example 4.3.

The code in Example 4.4 is commented and you should be able to follow it even if you are not familiar with PHP. One point to note is that in PHP the data structure known as an array works as both a sequential array and as an associative array. That is, JSON objects and JSON arrays are both arrays in PHP. Example 4.4 depends on an external module for JSON serialization and parsing. The module used here is from http:*//pear.php.net*.

Example 4.4. metaweb.php: using mqlread with PHP

```
<?php
/*
 * The Metaweb class defines a read() method for invoking the Metaweb
 * mqlread service on api.freebase.com. read() takes a MQL query (as a
 * PHP array), sends that query to the mqlread service and retrieves
 * the response. It parses the response to a PHP array, and extracts
 * the query result from the response envelope and returns it. If the
 * query fails, it returns null (without providing useful diagnostics).
 */
require "JSON.php"; // A JSON encoder/decoder from http://pear.php.net

class Metaweb {
  var $json;  // Holds the JSON encoder/decoder object
  var $URL = "http://api.freebase.com/api/service/mqlread";

  // Our constructor function sets up the JSON encoder/decoder
  function Metaweb() {
    // Set up our JSON encoder and decoder object
    $this->json = new Services_JSON(SERVICES_JSON_LOOSE_TYPE);
  }

  // This method submits a query and synchronously returns its result.
  function read($queryobj) {
    // Put the query into an envelope object
    $envelope = array("query" => $queryobj);

    // Serialize the envelope object to JSON text
    $serialized = $this->json->encode($envelope);

    // Then URL encode the serialized text
    $encoded = urlencode($serialized);
    // Now build the URL that represents the query
    $url = $this->URL . "?query=" . $encoded;

    // Use the curl library to send the query and get response text
    $request = curl_init($url);

    // Return the result instead of printing it out.
    curl_setopt($request, CURLOPT_RETURNTRANSFER, TRUE);

    // Now fetch the URL
    $responsetext = curl_exec($request);
    curl_close($request);

    // Parse the server's response from JSON text into a PHP array
    $response = $this->json->decode($responsetext);

    // Return null if the query was not successful
    if ($response["code"] !== "/api/status/ok")
        return null;

    // Otherwise, return the query result from the envelope
    return $response["result"];
  }
}
?>
```

With the PHP utility function defined in Example 4.4, it becomes easy to write simple Metaweb-enabled web applications in PHP. Example 4.5 demonstrates. It displays an HTML form in which the user can enter the name of a band. When the form is submitted, it lists the albums by that band.

Example 4.5. albumlist.php: A Metaweb-enabled web application in PHP

```
<html>
<body>
<form>Band: <input type="text" name="band"><input type="submit"></form>
<?php
$band = $_GET["band"];        // What band is specified in the URL?
if ($band) {                  // Only list albums if a band has been specified.
  require "metaweb.php";      // Import Metaweb utility code.
  $metaweb = new Metaweb();   // Create a Metaweb object.

  // Build a MQL request for the list of albums by the band
  $query = array("type" => "/music/artist", // We want a musical artist.
                 "name" => $band,           // This is its name.
                 "album" => array());       // Fill in this empty albums array!

  // Submit the query using the utility function defined earlier
  $result = $metaweb->read($query);

  // Now output the query results in HTML
  if ($result) {                              // If we got a result...
      echo "<hr><h1>Albums by " . $band . "</h1>"; // Display page title.
      $albums = $result["album"];                  // Get the array of albums.
      foreach ($albums as $album)                  // For each album...
          echo $album . "<br>";                    // ...display album name.
  }
  else {                                      // Otherwise, print error msg.
      echo "<hr><b>Unknown band: " . $band . "</b>";
  }
}
?>
</body>
</html>
```

4.5. Metaweb Queries with JavaScript

Since the MQL syntax is based on JSON, Metaweb queries are most gracefully expressed in JavaScript. We haven't seen a JavaScript-based Metaweb application so far for one important reason: the *same-origin policy*. The same-origin policy is a sweeping (but necessary) security restriction in JavaScript that says that code embedded in a document that was served by server A can only interact with content that is also served by server A. This restriction applies to the `XMLHttpRequest` object which is what is typically used to fetch the contents of a URL. A web application hosted at *api.freebase.com* can use `XMLHttpRequest` to submit MQL queries to the *mqlread* service on that same server, but this is not allowed for web applications hosted on any other server.

There are two workarounds to this restriction. The first, and most obvious, is to run a proxy script on your own site that behaves like the *mqlread* service but simply forwards your query to *api.freebase.com*.

The second workaround is known as JSONP and relies on the fact that a query result, in JSON format, is valid JavaScript code. This means that a *mqlread* URL can be used as the value of the `src` attribute of a `<script>` tag. When the server returns its result, the `<script>` tag evaluates the JSON text as JavaScript code. Evaluating the JSON text creates the JavaScript object we want, but to make this scheme work, the script then has to be able to do something with that object. The solution is to use the `callback` request parameter (which was described in §4.2.5). If the URL for your *mqlread* invocation includes a `callback` parameter, then *mqlread* will take the value of that parameter to be the name of a JavaScript function. Then, instead of simply returning a JSON text, it will return the specified function name, an open parenthesis, the JSON text and a close parenthesis. When used this way with a `<script>` tag, the JSON text is evaluated, and the object that results is passed to the specified function (which you must have defined previously).

We use the `<script>`-based JSONP technique in this chapter: it is simple, elegant and in common use across the internet.

One thing you'll notice about our JavaScript examples here is that they are asynchronous: when you submit a query you do not get the result immediately. Instead, the callback function you specify is invoked when the result is available. This asynchronous programming model is common in client-side JavaScript, but is substantially different from the synchronous model demonstrated in Example 4.4 and other examples.

4.5.1. Invoking mqlread with the jQuery Library

Let's jump right in. Example 4.6 is an album lister web application like Example 4.5, but it communicates with *freebase.com* using JavaScript on the client side instead of PHP on the server side. It uses the popular *jQuery* library to do JSONP-based interaction with *mqlread* and to build and traverse the HTML document that displays the query results. Download *jquery.js* (version 1.2 or later) from *http://jquery.com*. Example 4.6 also relies an external library named *json2.js*. This file defines JavaScript functions for parsing and serializing JSON. The code for this module is in the public domain and is available online at .

Example 4.6 is an HTML file that consists mainly of HTML code. There is a short HTML body that defines an input field into which you can type the name of a band and an HTML `<div>` tag into which the names of albums will be inserted. An event handler on the input field calls the JavaScript function `listAlbums()` whenever a new band name is entered.

There are several points to notice in the `listAlbums()` function. First, it includes a MQL query enclosed in a *mqlread* query envelope expressed in JavaScript. Since JSON is a subset of JavaScript, the JavaScript representation is very close to the pure-JSON queries we've seen elsewhere. Note, however, that JavaScript allows but does not require double quotes around most property names, and they've been omitted in this example. Second, the `listAlbums()` uses the jQuery function named `$()`. This tersely-named function is the central part of the jQuery API. It is used to find elements within an HTML document and also to create new elements. The return value of `$()` is a `jQuery` object which represents one or more document element and defines useful methods for operating on those elements. Third, the code that displays the list of album names uses `jQuery.each()` to iterate through the elements of the albums array and invoke the specified function on each of the album objects.

Finally, and most importantly, `listAlbums()` uses `jQuery.getJSON()` to invoke *mqlread*. The first argument specifies the URL of *mqlread*. The second argument specifies the names and values of the query parameters, and the third argument specifies a callback function to be invoked when the query results are ready. The only tricky detail here is that `jQuery.getJSON()` normally uses an `XMLHttpRequest`. To force it to use JSONP instead, we have to append the string "callback=?" to the URL. When it sees that string in the URL, it generates a unique JSONP callback name and inserts it into the URL in place of the question mark. The code for this `jQuery`-based album lister follows. Later, in Example 4.8 we'll see how you can perform JSONP "manually" in your own custom JavaScript library.

Example 4.6. simplelist.html: Listing albums with jQuery

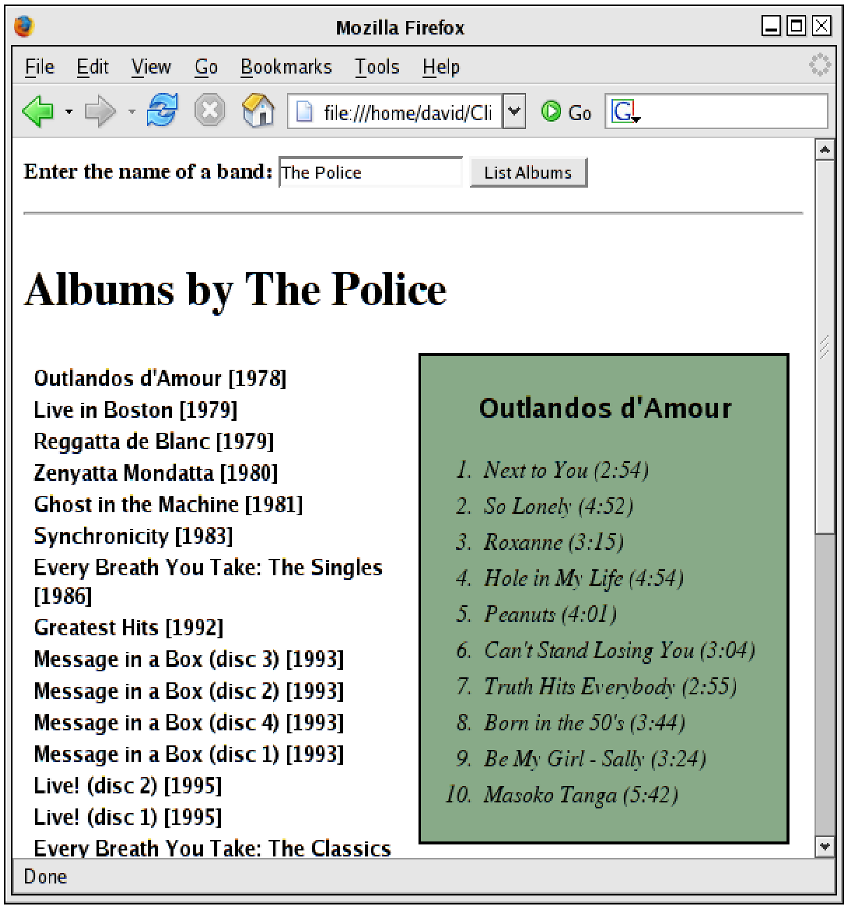
```
<html>
<head>
<script src="jquery.js"></script>    <!-- Ajax and DOM magic -->
<script src="json2.js"></script>     <!-- JSON.stringify() -->
<script>
function listAlbums(band) {     // Display albums by the specified band.
    var envelope = {                       // The mqlread query envelope
```

```
        query : {                          // The MQL query
            type: "/music/artist",         // Find a band
            name: band,                    // With the specified name
            album: [{                      // We want to know about albums
                name:null,                 // Return album names
                release_date:null,         // And release dates
                sort: "release_date",      // Order by release date
                "release_type!=":"single"  // Don't include singles
            }]
        }
    };

    var output = $("#output");                           // Output goes here
    output.html("<h1>Albums by " + band + "</h1>");     // Display a title

    // Invoke mqlread and call the function below when it is done.
    // Adding callback=? to the URL makes jQuery do JSONP instead of XHR.
    jQuery.getJSON("http://api.freebase.com/api/service/mqlread?callback=?",
                   {query: JSON.stringify(envelope)},   // URL parameters
                   displayResults);                      // Callback function

    // This function is invoked when we get the result of our MQL query
    function displayResults(response) {
        if (response.code == "/api/status/ok" &&
            response.result && response.result.album) { // Check for success...
            var list = $("<ul>");                        // Make <ul> tag.
            output.append(list.hide())                   // Keep it hidden
            var albums = response.result.album;         // Get albums.
            jQuery.each(albums, function() {             // Loop through albums.
                list.append($("<li>").html(this.name)); // Make <li> for each.
            });
            list.show("normal");                         // Reveal the list
        }
        else {                                           // On failure...
            output.append("Unknown band: " + band);     // Display message.
        }
    }
}
</script>
</head>
<body>
<b>Enter the name of a band: </b>                    <!-- Prompt for input -->
<input type="text" onchange="listAlbums(this.value)"> <!-- Get band name -->
<hr><div id="output"></div>                           <!-- Display output -->
</body>
</html>
```

4.5.2. Listing Albums and Tracks with JavaScript

Example 4.7 is a more complicated example than Example 4.6: in addition to listing albums by a band, it can also list tracks on an album. Its output is pictured in Figure 4.1. In addition to its more sophisticated output, it does not rely on the *jQuery* library, instead performing HTML document creation and manipulation using lower-level DOM calls.

Example 4.7 depends on two external modules. *json2.js* is the public-domain JSON parser and serializer that we used in Example 4.6. The second module of external code is *metaweb.js*. This module, whose code listed in Example 4.8, defines the utility function `Metaweb.read()` that submits a MQL query, through a `<script>` tag, to the *mqlread* service. This `Metaweb.read()` function performs its own JSONP networking without relying on *jQuery*.



Enter the name of a band: The Police

Albums by The Police

Outlandos d'Amour [1978]
Live in Boston [1979]
Reggatta de Blanc [1979]
Zenyatta Mondatta [1980]
Ghost in the Machine [1981]
Synchronicity [1983]
Every Breath You Take: The Singles [1986]
Greatest Hits [1992]
Message in a Box (disc 3) [1993]
Message in a Box (disc 2) [1993]
Message in a Box (disc 4) [1993]
Message in a Box (disc 1) [1993]
Live! (disc 2) [1995]
Live! (disc 1) [1995]
Every Breath You Take: The Classics

Outlandos d'Amour

1. Next to You (2:54)
2. So Lonely (4:52)
3. Roxanne (3:15)
4. Hole in My Life (4:54)
5. Peanuts (4:01)
6. Can't Stand Losing You (3:04)
7. Truth Hits Everybody (2:55)
8. Born in the 50's (3:44)
9. Be My Girl - Sally (3:24)
10. Masoko Tanga (5:42)

Figure 4.1. Listing albums and tracks

Example 4.7 lists the albums (it uses the `!=` constraint to exclude singles) by a specified band, and also displays the tracks on an album when the user clicks on the name of the album. There are several features worth noting in this example. First, notice that this code uses the `Metaweb.read()` function to send queries to the *mqlread* service. We'll see how `Metaweb.read()` is implemented in the next section. Second, note that the code displays a "Loading..." message to the user while the queries are pending and also displays an appropriate message if a query fails. element. Finally, in addition to querying album and track names, the queries in this example also ask for album release date and track length. The example includes utility functions to massage this data, extracting a year from a `/type/datetime` string and converting a track length in seconds into the more familiar *mm:ss* format.

Example 4.7. albumlist.html: a JavaScript album and track lister

```
<html>
<head>
<script src="json2.js"></script>     <!-- Defines JSON.stringify() -->
<script src="metaweb.js"></script>   <!-- Defines Metaweb.read() -->
<script>
/* Display albums by the specified band */
function listAlbums(band) {
    // Find the document elements we need to insert content into
    var title = document.getElementById("title");
    var albumlist = document.getElementById("albumlist");
    var tracklist = document.getElementById("tracklist");

    title.innerHTML = "Albums by " + band;           // Set the page title
    albumlist.innerHTML = "<b><i>Loading...</i></b>" // Album list is coming...
    tracklist.style.visibility = "hidden";           // Hide any old tracks

    var query = {                      // This is our MQL query
        type: "/music/artist",         // Find a band
        name: band,                    // With the specified name
        album: [{                      // We want to know about albums
            name:null,                 // Return album names
            id:null,                   // Also ids
            release_date:null,         // And release dates
            sort: "release_date",      // Order by release date
            "release_type!=":"single"  // Don't include singles
        }]
    };

    // Issue the query and invoke the function below when it is done
    Metaweb.read(query, displayAlbums);

    // This function is invoked when we get the result of our MQL query
    function displayAlbums(result) {
        // If no result, the band was unknown.
        if (!result || !result.album) {
            albumlist.innerHTML = "<b><i>Unknown band: " + band + "</i></b>";
            return;
        }

        // Otherwise, the result object matches our query object,
        // but has album data filled in.
        var albums = result.album;  // the array of album data
        // Erase the "Loading..." message we displayed earlier
        albumlist.innerHTML = "";
        // Loop through the albums
        for(var i = 0; i < albums.length; i++) {
            var name = albums[i].name;                   // album name
            var year = getYear(albums[i].release_date);  // album release year
            var text = name + (year?(" ["+year+"]"):""); // name+year

            // Create HTML elements to display the album name and year.
            var div = document.createElement("div");
            div.className = "album";
            div.appendChild(document.createTextNode(text));
            albumlist.appendChild(div);

            // Add an event handler to display tracks when an album is clicked
            div.onclick = makeHandler(name, albums[i].id);
        }

        // This function returns a function.  We do it this way to create
        // a closure that captures the band name and id.
        function makeHandler(name, id) {
            return function(e) { listTracks(name, id); }
        }
    }

    // A utility to return the year portion of a Metaweb /type/datetime
    function getYear(date) {
        if (!date) return null;
        if (date.length == 4) return date;
        if (date.match(/^\d{4}-/)) return date.substring(0,4);
        return null;
    }
}

/* Display the tracks on the specified album by the specified band */
function listTracks(albumname, albumid) {
    // Begin by displaying a Loading... message
    var tracklist = document.getElementById("tracklist");
    tracklist.innerHTML = "<h2>" + albumname + "</h2><p>Loading...";
    tracklist.style.visibility = "visible";

    // This is the MQL query we will issue
    var query = {
        type: "/music/album",
        id: albumid,
        // Get track names and lengths, sorted by index
        track: [{name:null, length:null, index:null, sort:"index"}]
    };

    // Issue the query, invoke the nested function when the response arrives
    Metaweb.read(query, function(result) {
                     if (result && result.track) { // If result is defined
                         var tracks = result.track;  // array of tracks
                         // Build an array of track names + lengths
                         var listitems = [];
                         for(var i = 0; i < tracks.length; i++) {
                             var n = tracks[i].name + " (" +
                               toMinutesAndSeconds(tracks[i].length) + ")";
                             listitems.push(n);
                         }
                         // Display the track list by setting innerHTML
                         tracklist.innerHTML = "<h2>" + albumname + "</h2>" +
                             "<ol><li>" + listitems.join("<li>") + "</ol>";
                     }
                     else {
                         // If empty result display error message
                         tracklist.innerHTML = "<h2>" + albumname + "</h2>" +
                             "<p>No track list is available.";
                     }
                 });

    // Convert track length in seconds to minutes:seconds format
    function toMinutesAndSeconds(seconds) {
        var minutes = Math.floor(seconds/60);
        var seconds = Math.floor(seconds-(minutes*60));
        if (seconds <= 9) seconds = "0" + seconds;
        return minutes + ":" + seconds;
    }
};
</script>
<!-- A CSS stylesheet to make the output look nice -->
<style>
#albumlist { width:50%; padding: 5px; }
#tracklist {
  width: 45%; float:right; visibility:hidden;
  padding: 5px; border: solid black 2px; margin-right: 10px;
  background-color: #8a8;
}
#tracklist h2 { font: bold 16pt sans-serif;  text-align: center;}
#tracklist p { text-align: center; font: italic bold 12pt sans-serif; }
#tracklist li { font-style: italic;}
div.album {  font: bold 12pt sans-serif; margin: 2px;}
div.album:hover {text-decoration: underline;}
</style>
</head>
<body>
<!-- The HTML form in which the user can enter the name of a band -->
<!-- It invokes listAlbums() when the user hits Return or clicks the button -->
<form onsubmit="listAlbums(this.band.value); return false;">
<b>Enter the name of a band: </b>
<input type="text" name="band">
<input type="submit" value="List Albums">
</form>
<hr>
<!-- This is where we insert the results of our Metaweb queries -->
```

```
<h1 id="title"></h1>        <!-- display band name here -->
<div id="tracklist"></div>  <!-- list tracks here -->
<div id="albumlist"></div>  <!-- list of albums here -->
</div>
</body>
</html>
```

4.5.3. Client-side MQL Queries with <script>

In this section, we develop the `Metaweb.read()` utility function used by Example 4.7. The code in Example 4.8 is short but somewhat complicated because it performs JSONP networking. The key to understanding it is to realize that each call to `Metaweb.read()` defines a function with a name like `Metaweb._3()` (the number is different on each invocation). This function does the work of processing the response from the Metaweb server. In order to get this function invoked, `Metaweb.read()` adds a `callback` parameter to the *mqlread* query URL, like this:

```
&callback=Metaweb._3
```

When the *mqlread* service is invoked with this `callback` parameter, it does not return the result as a pure JSON object. Instead it returns JavaScript code. The code is simply a function invocation of the function named by the parameter. The invocation includes a JSON object as the single argument to the function:

```
Metaweb._3(/* JSON object goes here */)
```

Since JSON is a subset of the JavaScript object and array literal syntax, any JSON object is a valid function argument. By simply wrapping a function invocation around the JSON object, we've converted the *mqlread* response into a form suitable for use with a `<script>` tag.

Note that `Metaweb.read()` uses `JSON.stringify()` to serialize the query object into JSON form. This utility function is defined in the *json2.js* module from *http://www.JSON.org/json2.js*. The accompanying `JSON.parse()` function is not required, however, since the JavaScript interpreter that processes the `<script>` tag serves as our JSON parser.

Once you've understood the use of the `callback` parameter, and the `JSON.stringify()` function, the other important feature to note about `Metaweb.read()` is that it can submit multiple MQL queries in a single *mqlread* request. Each query must have a corresponding function to which results are passed, and you can pass any number of query/function pairs to `Metaweb.read()`.

The implementation of `Metaweb.read()` shows how this is done: the function uses the `queries` parameter to *mqlread* instead of the `query` parameter, and it places each individual query envelope into an outer envelope object, giving each query a name: `q0`, `q1`, and so on.

Example 4.8. metaweb.js: Metaweb queries with script tags

```
/**
 * metaweb.js:
 *
 * This file implements a Metaweb.read() utility function using a <script>
 * tag to generate the HTTP request and the URL callback parameter to
 * route the response to a specified JavaScript function.
 **/
var Metaweb = {};                         // Define our namespace
Metaweb.HOST = "http://api.freebase.com"; // The Metaweb server
Metaweb.MQLREAD = "/api/service/mqlread"; // The mqlread service on that server

// This function submits one or more MQL queries to the mqlread service.
// When the results are available, it asynchronously passes them to
// the specified callback functions.  The function expects an even number
// of arguments: each pair of arguments consists of a query and a
// callback function.
Metaweb.read = function(/* q0, f0 [, q1, f1...] */) {
    // Figure out how many queries we've been passed
    if (arguments.length < 2 || arguments.length % 2 == 1)
        throw "Wrong number of arguments to Metaweb.read()";
    var nqueries = arguments.length / 2;

    // Place each query in a query envelope, and put each query envelope
    // in an outer envelope.  Also, store the callbacks in an array for
    // later use.
    var envelope = {}                          // The outer envelope
    var callbacks = new Array(nqueries);       // An array to hold callbacks
    for(var i = 0; i < nqueries; i++) {        // For each query/callback pair
        var inner = {"query": arguments[i*2]}; // Make inner query envelope
        var qname = "q" + i;                   // Property name for the query
        envelope[qname] = inner;               // Put inner envelope in outer
        callbacks[i] = arguments[i*2 + 1];     // Callback for the query
    }

    // Serialize and encode the envelope object.
    var serialized = JSON.stringify(envelope);    // http://json.org/json2.js
    var encoded = encodeURIComponent(serialized); // Core JavaScript function

    // Start building the URL
    var url = Metaweb.HOST + Metaweb.MQLREAD +  // Base mqlread URL
        "?queries=" + encoded;                  // Queries request parameter

    // Get a callback function name for this url
    var callbackName = Metaweb.makeCallbackName(url);

    // Add the callback parameter to the URL
    url += "&callback=Metaweb." + callbackName;

    // Create the script tag that will fetch the contents of the url
```

```
var script = document.createElement("script");

// Define the function that will be invoked by the script tag.
// This function expects to be passed an outer response envelope.
// It extracts query results and passes them to the corresponding callback.
// The function throws exceptions on errors. Since it is invoked
// asynchronously, those exceptions can't be caught, but they will
// appear in the browser's JavaScript console as useful diagnostics.
Metaweb[callbackName] = function(outer) {
    // Throw an exception if there was an invocation error.
    if (outer.code != "/api/status/ok") {  // Should never happen
        var error = outer.messages[0];
        throw outer.status + ": " + error.code + ": " + error.message;
    }

    var errors = [];  // An array of error messages to be thrown later

    // For each query, get the response envelope, test for success,
    // and pass query results to the corresponding callback function.
    // If any query (or callback) fails, save an error to throw later.
    for(var i = 0; i < nqueries; i++) {
        var qname = "q" + i;            // Query property name
        var inner = outer[qname];       // Extract inner envelope
        // Check for query success or failure
        if (inner.code == "/api/status/ok") {
            try {
                callbacks[i](inner.result); // On success, call callback
            } catch(ex) {
                // Remember any exceptions caused by the callback
                errors.push("Exception from callback #" + i + ": " + ex);
            }
        }
        else {
            // If it failed, add all of its error messages to errors[].
            for(var j = 0; j < inner.messages.length; j++) {
                var error = inner.messages[j];
                var msg = "mqlread error in query #" + i +
                    ": " + error.code + ": " + error.message;
                errors.push(msg);
            }
        }
    }

    // Now perform some cleanup
    document.body.removeChild(script);   // Remove the <script> tag
    delete Metaweb[callbackName];        // Delete this function

    // Finally, if there were any errors, raise an exception now so they
    // at least get reported in the JavaScript console.
    if (errors.length > 0) throw errors.join("\n");
};
```

```
    // Now set the URL of the script tag and add that tag to the document.
    // This triggers the HTTP request and submits the query.
    script.src = url
    document.body.appendChild(script);
};

// This function returns a callback name that is not currently in use.
// Ideally, to support caching, the name ought to be based on the URL so the
// same URL always generates the same name.  For simplicity, however, we
// just increment a counter here.
Metaweb.makeCallbackName = function(url) {
    return "_" + Metaweb.makeCallbackName.counter++;
};
Metaweb.makeCallbackName.counter = 0; // Initialize the callback name counter.
```

4.6. The Metaweb Search Service

The *freebase.com* website includes a search text box in the upper-right corner. Type in some text and hit return, and freebase will list database entries that match your query. While you type it offers a drop-down menu of suggestions (or auto-completions). Both the search function and the auto-complete function are powered by Metaweb's *search* service. For every object in the database, the *search* service indexes the name and aliases (`/common/topic/alias`) of the object, as well as any other properties of `/type/text`, and any documents associated (such as through `/common/topic/article`) with the object. Search results are ordered according to an opaque, but well-tuned ranking system to yield the most relevant results first. At the time of this writing [5] Metaweb's search engine is not language-aware, and indexes all text without regard to its source language.

This section describes the search API so that you can use it in your own web applications. Note, however that if you simply want to duplicate on your own website the searching and drop-down suggestion functionality of *freebase.com*, you should consider using the open-source *freebase-suggest* library, which is available at *http://code.google.com/p/freebase-suggest/*.

Keep in mind that the *search* service is a sophisticated full-text search engine that indexes Metaweb objects and the documents associated with them. Many simpler searches (such as looking for objects with particular words in their names) are best expressed as MQL queries (using the ~= operator for pattern matching) using the *mqlread* service.

The *search* service is a web-based service, like `mqlread`. The base URL for requesting search results from freebase.com is:

```
http://api.freebase.com/api/service/search
```

Unlike *mqlread*, the *search* service does not encode queries as JSON objects, and instead passes the query text and other search variables as URL parameters. For example, to search for people named "Smith", you could use this URL:

```
http://api.freebase.com/api/service/search?query=Smith&type=/people/person
```

[5] September, 2008

The URL parameters to *search* are described in §4.6.1. If you enter the above URL into a browser, you'll see that the *search* service, like *mqlread*, returns its results as a JSON object. The format of the results are covered in §4.6.2.

The sub-sections that follow describe the input and output of the *search* service and then demonstrate its use with a JavaScript example.

4.6.1. Search Input

Input to the *search* service takes the form of HTML form-encoded parameters appended to the URL. There are four categories of parameters:

- parameters that specify the text to be matched;
- parameters that narrow the field of search by domain or type;
- parameters that specify the number or offset of the desired results; and
- parameters that affect the format of the returned results.

Every invocation of the *search* service must include either a `query` parameter or a `prefix` parameter (but not both). Both specify text to be searched for. If you use the `query` parameter, your text will only match complete words. If you specify `prefix`, however, any word that begins with your text matches. Searches are case-insensitive and ignore punctuation and accents on characters. The search target you specify may include multiple words, but you may not include multiple `query` or `prefix` parameters in a search URL.

You can narrow the field of search (or at least change the way the *search* service ranks results) with the `domain` and `type` parameters. The value of the `domain` parameter should be the id of a Metaweb domain. For example:

```
http://api.freebase.com/api/service/search?query=Smith&domain=/film
```

This query looks for topics that match the word Smith and have at least one type in the `/film` domain. Results might include the `/film/actor` Will Smith, and the `/film/film` "Mr. Smith Goes to Washington".

It is unusual to do a domain constraint alone as in the example above. Here is an alternative (with the URL truncated to fit on one line) that looks for film-related people named Smith. It matches actors and producers, but not films:

```
/api/service/search?query=Smith&type=/people/person&domain=/film
```

The `type` parameter specifies the id of a Metaweb type. In the search URL above, it specifies that each of the matches should be a `/people/person` object.

A *search* URL may include more than one `type` parameter to specify more than one type. The way that `type` parameters affect search results is controlled by the `type_strict` parameter, which must have one of the following three values:

`all`	Search results will only include objects that are members of all of the types described.
`any`	Search results will only include objects that are members of at least one of the specified types. Objects that match more types will have increased relevance scores, and may appear earlier in the search results. This is the default type matching mode when no `type_strict` parameter is given.
`should`	The specified types will be used in computing the relevance of the search results, but results may include objects that do not match any of the specified types.

Here are some more search URLs (abbreviated so they fit on one line) with comments indicating what they do:

```
// Find objects matching "Smith" that are either films or actors.
// type_strict=any is implicit in this search
search?query=Smith&type=/film/film&type=/film/actor

// Find objects matching "Smith" that are both films and actors.
// Results, if any, probably represent typing errors in the database.
search?query=Smith&type=/film/film&type=/film/actor&type_strict=all

// Find objects matching "Smith"; give priority to films and actors
search?query=Smith&type=/film/film&type=/film/actor&type_strict=should
```

By default, the *search* service returns the 20 most relevant results. You can request a different number of results with the `limit` parameter. If you want to retrieve another page of less-relevant results, you can use the `start` parameter:

```
search?query=Smith                   // Results 1-20
search?query=Smith&limit=10          // Results 1-10
search?query=Smith&start=10&limit=10 // Results 11-20
```

The final category of URL parameters are those miscellaneous parameters that affect the formatting of the results. Use `escape=false` to turn off HTML escaping of the &, <, and > characters in the results. Use `indent=true` if you want the JSON string of search results to be pretty-printed so that it is more human-readable. This parameter is useful when experimenting with search URLs in a web browser, but is not necessary when executing searches from scripts. The search results shown in the next section are pretty-printed, even though the `indent` parameter is omitted from the search URLs.

If you want to use the *search* service from client-side JavaScript code, you'll need to use the `callback` parameter. This parameter enables JSONP just as it does for *mqlread*: it wraps a JavaScript function invocation around the JSON result string. Using the *search* service with JavaScript is demonstrated in Example 4.9 and Example 4.10.

Finally, the `mql_output` parameter specifies which properties of each matching object are to appear in the search results. See §4.6.2 for details and examples.

4.6.2. Search Output

The *search* service returns a JSON-encoded envelope object just as the *mqlread* service does. This object has a code property that specifies whether the search succeeded or failed. If this code is anything other than "/api/status/ok", then the search failed. In this case, the envelope object has a messages property whose value is an array of one or more message objects. The messages array returned by *search* is just like that returned by *mqlread*: each element of this array includes a code property that identifies the specific kind of error and a message property that can be used to generate a diagnostic message. Certain message code values also have an associated info property that provides error details, but the list of codes and the format of their associated info objects is not documented.

If you get an error from the *search* service, it typically means that you invoked it incorrectly. Errors can occur if (for example) you don't specify either the query or prefix parameter, or if you specify an illegal value for the type_strict parameter, or if you specify an undefined id as the value of the type or domain parameter. If a search URL is properly constructed, the search is considered a success, even if it matches nothing and returns no results.

If the code property of the envelope object is "/api/status/ok", then the query was a success, and the envelope has a result property that holds a (possibly empty) array of search results.

The following search URL, for example:

```
http://api.freebase.com/api/service/search?query=Smith&limit=1
```

might return these results:

```
{
  "status": "200 OK",
  "code": "/api/status/ok",
  "transaction_id":"cache;cache01.p01.sjc1:8101;2008-09-16T21:47:14Z;0006",
  "result": [
    {
      "id": "/guid/1f80000000004f16a0",
      "name": "William Smith",
      "alias": [],
      "type": [
        { "id": "/common/topic", "name": "Topic" },
        { "id": "/people/person", "name": "Person" },
        { "id": "/people/deceased_person", "name": "Deceased Person" }
      ],
      "article": { "id": "/guid/1f80000000004f16a5" },
      "image": null
    }
  ]
}
```

The envelope object includes status, code and transaction_id properties just *mqlread* response envelopes do. The interesting part of the envelope object is the result array. For this example query, we explicitly specified a limit of 1, so the array has only one element, but in general each element of the `result` array provides information about a single Metaweb object that matched the search query. By default (if you do not specify a `mql_output` parameter in the search query), each element is an object with the following properties:

`id`	The Metaweb id of the matched object.
`name`	The name of the matched object. (At the time of this writing, [6] there is no way to specify the preferred language in which the name should be returned.)
`alias`	An array of nicknames for the object. These are the values of the `/common/topic/alias` property.
`type`	An array that specifies the types of the object. Each element of this array is an object with `id` and `name` properties that specify the Metaweb id and the human-readable name of the type.
`article`	If the matched object has at least one associated document as the value of the `/common/topic/article` property, then this result property refers to an object with a single `id` property. This `article.id` property is the Metaweb id of the most recent document associated with the object. A blurb from this article can be a useful addition to search results. (§4.7 shows how to retrieve article blurbs.) If the matched object has no associated documents, then this property is `null`.
`image`	If the matched object has at least one associated image as the value of the `/common/topic/image` property, then this property refers to an object with nothing but an `id` property. `image.id` is a Metaweb id of the first image (the image with `index:0`, see §3.5.4). It may be useful to include a thumbnail of this image in a listing of search results. (§4.7 shows how to retrieve image thumbnails.) If the matched object has no associated images, then this `image` property is `null`.

These default properties of the elements of the `results` array can be overridden with the `mql_output` URL parameter. The value of this parameter should be a MQL query in square brackets. The *search* service adds the following properties to the query you specify:

```
"guid": null
"guid|=": [guids of all matching objects here]
```

The `guid|=` property specifies the guids of all objects that matched the search. The query is passed to *mqlread*, and the results become the `results` array of the search. Consider this query (which wraps onto two lines), for example:

```
http://api.freebase.com/api/service/search?query=love&type=/music/track&limit=3
&mql_output=[{"name":null,"/music/track/artist":null}]
```

It returns results like these:

[6] September, 2008

```
{
  "status": "200 OK",
  "code": "/api/status/ok",
  "transaction_id":"cache;cache01.sandbox.sjc1:8101;2008-09-16T23:13:07Z;0001",
  "result": [{
    "guid" : "#1f8000000001268f44",
    "name" : "Tainted Love",
    "/music/track/artist" : "Soft Cell"
  },{
    "guid" : "#1f80000000012b0704",
    "name" : "Endless Love",
    "/music/track/artist" : "Diana Ross &amp; Lionel Richie"
  },{
    "guid" : "#1f800000000129a206",
    "name" : "The Power of Love",
    "/music/track/artist" : "Huey Lewis &amp; the News"
  }]
}
```

4.6.3. Example: Searching for Band Names

To demonstrate the *search* service, let's extend the *metaweb.js* module of Example 4.8. Example 4.9 defines a JavaScript function named `Metaweb.search()` that invokes the *search* service. The code in this example is intended as an extension of Example 4.8. It depends on the `Metaweb` object, `Metaweb.HOST` constant and `Metaweb.makeCallbackName()` function defined in that previous example.

Example 4.9. metaweb.js: searching Metaweb

```
Metaweb.SEARCH = "/api/service/search";  // URL path to the search service

// Invoke the Metaweb search service for the specified query.
// Asynchronously pass the array of results to the specified callback function.
//
// The first argument can be a string for simple searches or an object
// for more complex searches.  If it is a string, it should take the form
//    [type:]text[*]
// That is: the text to be searched for, optionally prefixed by a type id
// and a colon and optionally suffixed with an asterisk.  Specifying a type
// sets the type parameter for the search, and adding an asterisk makes it a
// prefix search.
//
// If query argument is an object, then its properties are translated into
// search parameters.  In this case, the object must include either
// a property named query (for an exact match) or a property named prefix
// (for a prefix match).  Other legal properties are the same as the
// allowed parameters for the search service: type, type_strict, domain,
// limit, start, and so on.  To specify multiple types, set the
// type property to an array of type ids.  To specify a single type, set
// the type property to a single id.
```

```
Metaweb.search = function(query, callback) {
    var q = {};  // The query object

    if (typeof query == "string") {
        // If the query argument is a string, we must convert it to an object.
        // First, see if there is a type prefix
        var colon = query.indexOf(':');
        if (colon != -1) {
            q.type = query.substring(0, colon);
            query = query.substring(colon + 1);
        }

        // Next see if there is an asterisk suffix
        if (query.charAt(query.length-1) == '*') // prefix match
            q.prefix = query.substring(0, query.length-1);
        else
            q.query = query;
    }
    else {
        // Otherwise, assume the query argument is an object and
        // copy its properties into the q object.
        for(var p in query) q[p] = query[p];
    }

    // With mqlread, we would JSON-encode the query object q.  For the search
    // service, we convert the properties of q to an array of URL parameters
    var parameters = [];
    for(var name in q) {
        var value = q[name];

        if (typeof value != "object") { // A single value for the parameter
            var param = name + "=" + encodeURIComponent(value.toString());
            parameters.push(param);
        }
        else { // Otherwise, there is an array of values: multiple types
            for(var index in value) {
                var elt = value[index];
                var param = name + "=" + encodeURIComponent(elt.toString());
                parameters.push(param);
            }
        }
    }

    // Now convert the array of parameters into a URL
    var url = Metaweb.HOST + Metaweb.SEARCH + "?" + parameters.join('&');

    // Generate a name for the function that will receive the results
    var cb = Metaweb.makeCallbackName(url);

    // Add the JSONP callback parameter to the url
    url += "&callback=Metaweb." + cb;
    // Create the script tag that will fetch that URL
    var script = document.createElement("script");

    // Define the function that handles the results from that URL
    Metaweb[cb] = function(envelope) {
        // Clean up by erasing this function and deleting the script tag
        document.body.removeChild(script);
        delete Metaweb[cb];

        // If the query was successful, pass results to the callback
        // Otherwise, throw an error message
        if (envelope.code == "/api/status/ok")
            callback(envelope.result);
        else {
            throw "Metaweb.search: " + envelope.messages[0].code +
                ": " + envelope.messages[0].message;
        }
    }

    // Now set the URL of the script tag and add that tag to the document.
    // This triggers the HTTP request and submits the search query.
    script.src = url
    document.body.appendChild(script);
};
```

Example 4.10 demonstrates how this `Metaweb.search()` function might be used in practice. It is a new version of the JavaScript-based album listing application shown in Example 4.7. The relevant new feature of this version is that if the user enters a name that is not a known band name, the application uses that name a in a search query and lists the results. For brevity, the album-listing features of this new version have been simplified, and the track-listing features have been removed.

Example 4.10. albumlist2.html: Searching for bands

```
<html>
<head>
<script src="json2.js"></script>     <!-- Defines JSON.stringify() -->
<script src="metaweb.js"></script>   <!-- Defines Metaweb.read() -->
<script>
/* Display albums by the specified band */
function listAlbums(band) {
    // Find the document elements we need to insert content into
    var title = document.getElementById("title");
    var albumlist = document.getElementById("albumlist");

    var query = [{             // This is our simple MQL query
        type: "/music/artist", // Find a band
        name: band,            // With the specified name
        album: []              // And return all album names
    }];
    // Issue the query and invoke the function below when it is done
    Metaweb.read(query, function(result) {
        // If no result, the band was unknown, so search for matches
        if (!result || result.length == 0) searchForBands(band);
        // Otherwise, we found a band, so list its albums
        else {

            title.innerHTML = "Albums by " + band; // Display title
            if (result[0].album.length == 0)       // If no albums, say so
                albumlist.innerHTML = "No albums found.";
            else                                   // Display the list
                albumlist.innerHTML = result[0].album.join("<br>");
        }
    });
}

// Find names of bands matching the user's partial input
function searchForBands(band) {
    // The Metaweb.search function will translate this object into
    // URL parameters for the search service.
    var query = {
        prefix: band,            // Prefix search
        type: "/music/artist",   // for bands (using default type_strict:all)
    };

    Metaweb.search(query, function(results) {
        // If the search returns no results then we don't know what band
        if (results.length == 0) {
            document.getElementById("title").innerHTML = "Unknown Band"
            document.getElementById("albumlist").innerHTML = "";
        }
        // Otherwise, display a list of links to possible bands
        else {
            document.getElementById("title").innerHTML = "Do you mean..."
            var links = new Array(results.length);
            for(var i = 0; i < results.length; i++) {        // For each result
                var band = results[i].name;                  // Get band name
                links[i] = '<a href="javascript:listAlbums(\'' + // make link
                           band.replace("'","\\'") + '\')">' +
                           band + '</a>';
            }
            // Output list of links
            document.getElementById("albumlist").innerHTML=links.join("<br>");
        }
    });
}
</script>
</head>
<body>
<!-- The HTML form for entering a band name -->
```

```
<form onsubmit="listAlbums(this.band.value); return false;">
<b>Enter the name of a band: </b>
<input type="text" name="band">
<input type="submit" value="List Albums">
</form>
<hr>
<!-- This is where we insert the results of our Metaweb queries -->
<h1 id="title"></h1>        <!-- display band name here -->
<div id="albumlist"></div>  <!-- list of albums here -->
</div>
</body>
</html>
```

4.7. Fetching Content with trans

As explained in Chapter 2, Metaweb is really two databases in one. One database is the graph of nodes and relationships. The second is the content store that holds chunks (or "blobs") of data such as HTML documents and graphical images. We use *mqlread* service to retrieve data from the graph, and we use the *trans* service to retrieve content from the content store.

The *trans* service is so named because in addition to fetching the requested data, it can also *translate* it for you. For example, it can "translate" an image to thumbnail size.

The *trans* service is HTTP based, just as *mqlread* is. Content is retrieved by specifying the desired translation and the content id, with a URL of this form:

`http://api.freebase.com/api/trans/`*`translation/id`*

Here, for example, is an actual *trans* URL:

`http://api.freebase.com/api/trans/raw/guid/1f8000000003c1978c`

The *translation* portion of a *trans* URL must be one of the following:

`raw`
: Use `raw` to request that no translation is to be done on the data: it should be returned as is. (Note, however that HTML content is not completely raw: it is "sanitized" by stripping executable content such as JavaScript.)

`image_thumb`
: Use `image_thumb` to request a thumbnail-sized version of an image. You can add request parameters `maxwidth` and `maxheight` to the URL to specify the desired pixel dimensions of the thumbnail. The aspect ratio of the original image is always preserved, and, by default, the image is not cropped, so if you specify both `maxwidth` and `maxheight`, the resulting image will typically match only one of those dimensions. The default value of both parameters is 75.

: If you want to specify the exact size of both dimensions of the thumbnail, add `mode=fillcrop` to the URL: this will cause one dimension to be cropped as necessary, while still preserving the image's aspect ratio.

`blurb`
: Use `blurb` to request an excerpt of document content. This provides a kind of a preview, of the kind you might see in a list of search results. In HTML documents, only content within `<p>` tags is returned, and all HTML tags are normally stripped. The default blurb length is 200 bytes, but you can alter this with the `maxlength` request parameter. If a blurb spans multiple paragraphs, the paragraph breaks are usually removed (along with other HTML tags). Add the request parameter `break_paragraphs=true` to preserve HTML paragraph breaks in the blurb.

The path component that follows the *translation* is a Metaweb id. The id passed to *trans* must identify an object of type `/type/content`, `/common/image` or `/common/document`. These three types are closely related:

`/type/content`
: A `/type/content` object is the representation in the Metaweb graph of an entry in the Metaweb content store.

`/common/image`
: When an image is added to the content store, the `/type/content` object for the image is co-typed `/common/image`, in order to add a `size` property that supplies the image dimensions. For images, therefore, the `id` of the `/type/content` and `/common/image` objects are the same.

`/common/document`
: When document content is added to the content store, a `/type/content` object is created to represent the entry in the content store. Typically a separate `/common/document` object is also created. The `content` property of the document object refers to the content object. Other properties of the `/common/document` object provide additional meta-information about the document. The reason that `/common/document` and `/type/content` are separate objects in this case (instead of just one object with two types) is for versioning: the `content` property of the `/common/document` can easily be updated to refer to a different `/type/content` object when the document changes.

  `/common/document` objects can also represent Wikipedia document content (which is not stored in the Metaweb content store). Documents that represent Wikipedia entries have `content` properties of `null` (and have a key in the `/wikipedia/en_id` namespace that defines the Wikipedia id of the document).

  The *trans* service works with both Wikipedia and non-Wikipedia documents. For non-Wikipedia documents, you can use either the id of the `/common/document` object or of the `/type/content` object it refers to.

The following MQL query asks Metaweb for the ids of documents and images related to our favorite band, The Police:

Query

```
{
  "id": "/en/the_police",
  "type":"/common/topic",
  "article":[{"id":null}],
  "image":[{"id":null}]
}
```

Result

```
{
  "id" : "/en/the_police",
  "type" : "/common/topic",
  "article" : [{
    "id" : "/guid/1f800000000006df25"
  }],
  "image" : [{
    "id" : "/wikipedia/images/en_id/982873"
  },{
    "id" : "/wikipedia/images/commons_id/3520500"
  }],
}
```

Given these results, we can retrieve the document and the first image with these URLs:

```
http://api.freebase.com/api/trans/raw/guid/1f800000000006df25
http://api.freebase.com/api/trans/raw/wikipedia/images/en_id/982873
```

And the following URLs (truncated on the left so they fit on a line) retrieve a short blurb and a big thumbnail:

```
/api/trans/blurb/guid/1f800000000006df25?maxlength=20
/api/trans/image_thumb/wikipedia/images/en_id/982873?maxwidth=200&maxheight=200
```

In web applications it is often easiest to use the *trans* service with `<img>` and `<iframe>` tags. To retrieve and display an image, simply use a *trans* URL as the `src` attribute of an `<img>` tag. And to retrieve and display the HTML content of a document, use a *trans* URL as the `src` attribute of an `<iframe>` or the `href` attribute of a hyperlink.

It is also possible, of course, for scripts to download the content of *trans* URLs themselves, and process document or image content in whatever way they want. In JavaScript code, the *trans* service can be used with a `callback` parameter just like the *mqlread* and *search* services can be. The sub-sections that follow extend our *metaweb.js* module to add functions for invoking the *trans* service. The module code is followed by two examples. One uses *trans* with images, iframes and hyperlinks. The other uses the `callback` parameter to the *trans* service and actually downloads document content directly.

4.7.1. JavaScript Functions for Downloading Content

Example 4.11 is JavaScript code that extends our *metaweb.js* module to handle the *trans* family of services. Three simple functions, `Metaweb.contentURL()`, `Metaweb.blurbURL()`, and `Metaweb.thumbnailURL()` accept an object id and return the a URL for fetching the specified content. The `blurbURL` function accepts optional length and paragraph breaking arguments and the `thumbnailURL` function accepts optional width and height arguments. Both encode these arguments into the returned URL. These URL functions will be demonstrated in Example 4.12.

The `Metaweb.download()` function is more interesting. It uses the `callback` parameter and a dynamically generated script tag to asynchronously download document content (or a document blurb) and pass it to a specified function (or insert it into a specified document element). This allows document content to be inserted directly into a document rather than isolated in a separate `<iframe>`. `Metaweb.download()` is demonstrated in Example 4.13.

The code in Example 4.11 is an extension to the *metaweb.js* module of Example 4.8, and Example 4.9 and is intended to be appended to those previous examples. The code shown here assumes that `Metaweb`, `Metaweb.HOST` and `Metaweb.makeCallbackName()` are already defined.

Example 4.11. metaweb.js: an extension for the trans service

```
Metaweb.RAW = "/api/trans/raw";
Metaweb.BLURB = "/api/trans/blurb";
Metaweb.THUMB = "/api/trans/image_thumb";

// Return a URL for fetching the content specified by id.
// This id must identify a /type/content object, or a /common/document or
// /common/image.  The returned URL is suitable for use as the value of
// the src attribute of <iframe> or <img> or the href attribute of <a>.
Metaweb.contentURL = function(id) {
    return Metaweb.HOST + Metaweb.RAW + id;
};

// Return the URL of an excerpt or "blurb" of the document specified by id.
// The maxlen argument specifies the length of the blurb. If maxlen is
// omitted, the default is 200. If the document is an HTML document, then only
// content within <p> tags is returned.  Normally all HTML tags are stripped
// from the returned blurb, making it plain text.  For long blurbs, this can
// cause paragraphs to run together. Pass true as the third argument to retain
// <p> tags (but strip all others) in the returned blurb.
Metaweb.blurbURL = function(id, maxlen, paragraphs) {
    var url = Metaweb.HOST + Metaweb.BLURB + id;     // Base url
    if (maxlen) url += '?maxlength=' + maxlen;       // Specify blurb length
    if (paragraphs) url += '&break_paragraphs=true'; // Include <p> tags
    return url;
};

// Return the URL of a scaled-down version of the image specified by id.
// The thumbnail always preserves the aspect ratio of the original image.
// Specify the maximum width and height of the image with the maxwidth and
// maxheight arguments.  The defaults for both are 75, meaning that the
// thumbnail will have one dimension equal to 75 and the other less than or
// equal to 75.
Metaweb.thumbnailURL = function(id, maxwidth, maxheight) {
    var url = Metaweb.HOST + Metaweb.THUMB + id;
    if (maxwidth) url += '?maxwidth=' + maxwidth;
    if (maxheight) url += '&maxheight=' + maxheight;
    return url;
}

// Download the /common/document or /type/content with id specified by from.
// If the argument to is a function, pass the document content, content type
// and encoding to the function.  Otherwise, if to is a DOM element or a
// string that identifies a DOM element insert the content into that element.
// The third argument is optional. If specified, it should be the length
// of the desired excerpt to be downloaded with /api/trans/blurb.
Metaweb.download = function(from, to, maxlen) {
    // What service are we using?
    var service = maxlen ? "/api/trans/blurb" : "/api/trans/raw";

    // This is the URL we must request with a script tag.
    var url = Metaweb.HOST + service + from;

    // Obtain a unique name for the function to receive the download.
    var cb = Metaweb.makeCallbackName(url);

    // Add the JSONP callback parameter to the URL
    url += "?callback=Metaweb." + cb;

    // Add the maxlength argument for blurbs.
    if (maxlen && typeof maxlen == "number") url += "&maxlength=" + maxlen;

    // Create the script tag that will do the download for us.
    var script = document.createElement("script");

    // Define the uniquely-named function that receives the response.
    Metaweb[cb] = function(envelope) {
        // Clean up this function and the script tag.
        document.body.removeChild(script);   // Remove the <script> tag.
        delete Metaweb[cb];                  // Delete this function.

        // If there was an error, throw an error message
        if (envelope.code != "/api/status/ok") {
            var err = envelope.messages[0];
            throw "Metaweb.download: " + envelope.status + ": " +
              err.code + ": " + err.message;
        }

        // Otherwise, get the results
        var doc = envelope.result;

        // Now handle the content we've downloaded based on the type of to.
        switch(typeof to) {
        case "function":  // Pass content to a function.
            to(doc.body, doc.media_type, doc.text_encoding);
            break;
        case "string":    // Treat string as element id.
            document.getElementById(to).innerHTML = doc.body;
            break;
        case "object":    // Assume to is a DOM element.
            to.innerHTML = doc.body;
            break;
        }
    }

    // Now set the URL of the script tag and add that tag to the document.
    // This triggers the HTTP request and invokes the trans service.
    script.src = url;
    document.body.appendChild(script);
}
```

4.7.2. Example: What's New on freebase.com

Example 4.12 is a JavaScript-based example that displays recently-added content from freebase.com. It issues *mqlread* queries to find the five images and five documents most recently added to *freebase.com*. (It takes advantage of the fact that the `Metaweb.read()` function defined in Example 4.8 can issue multiple queries at the same time.) It then dynamically generates `<img>` and `<iframe>` tags to display image thumbnails and document blurbs, and uses the `Metaweb.thumbnailURL()` and `Metaweb.blurbURL()` functions defined in Example 4.11 to create URLs for the `src` attributes of those tags. It also creates `<a>` tags, and uses `Metaweb.contentURL()` to hyperlink to full-sized versions of the images and documents. (These links open new windows to display the image or document.)

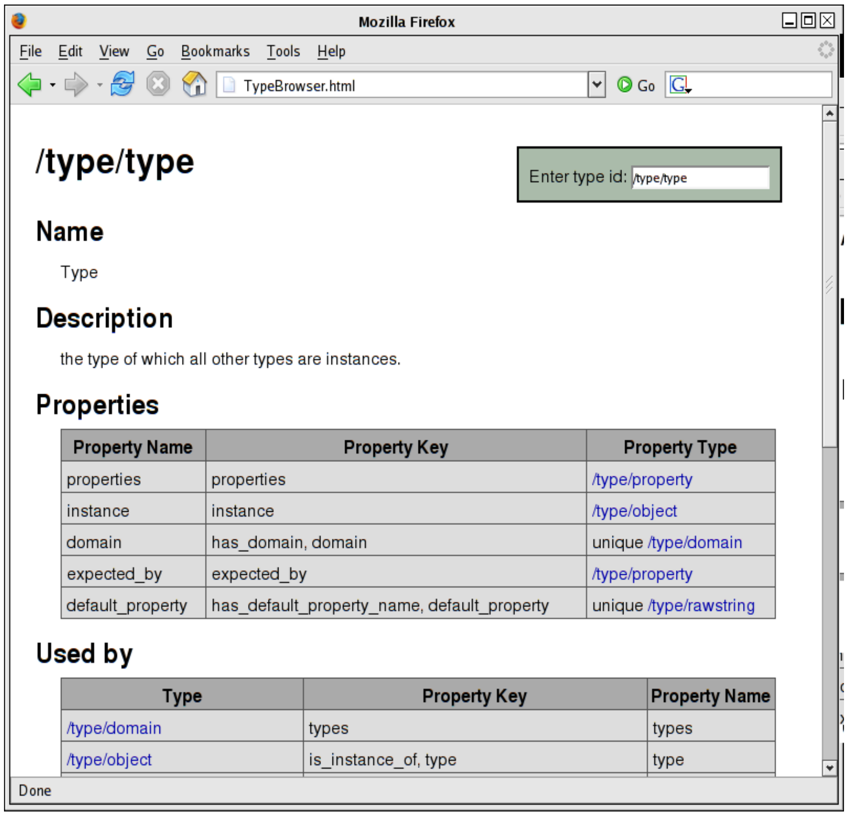
Example 4.12. WhatsNew.html: fetching new images and documents from freebase.com

```
<html>
<head>
<script src="json2.js"></script>    <!-- Required by metaweb.js -->
<script src="metaweb.js"></script>  <!-- Defines Metaweb.read(), etc. -->
<script>
// How many images and how many documents do we display?
var N = 5;                                     // This is the default
if (window.location.search.substring(0,3) == "?n=")  // URL argument overrides
    N = parseInt(window.location.search.substring(3));

// The query to find the N newest images
var imageQuery = [{
    type:"/common/image", id:null,         // Return image ids
    timestamp:null, sort:"-timestamp",     // Most recent first
    limit:N,                               // Only N of them
    "/type/content/media_type":null,       // Check image type, too
    "/type/content/media_type|=":[         // We only want images that are:
        "/media_type/image/gif",             // GIF or
        "/media_type/image/png",             // PNG or
        "/media_type/image/jpeg"             // JPEG
        ]
}];

// The query to find the N newest documents
var documentQuery = [{
    type:"/common/document", id:null,    // Return document ids
    timestamp:null, sort:"-timestamp",   // Most recent first
    limit:N                              // Only N of them
}];

// When the document has loaded, send the queries above to api.freebase.com.
// This will invoke the functions below when the results arrive.
window.onload = function() {
    Metaweb.read(imageQuery, displayImages,
                 documentQuery, displayDocuments);
};

// This function is invoked with the results of the image query.
function displayImages(images) {
    var container=document.getElementById("images"); // Get container element.
    for(var i = 0; i < images.length; i++) {         // Loop through images.
        var id = images[i].id;                       // Get the image id.
        var img = document.createElement("img");     // Create <img> tag
        img.src = Metaweb.thumbnailURL(id);          // ...for image thumbnail
        img.title = images[i].timestamp;             // ...timestamp tooltip.
        var link = document.createElement("a");      // Create hyperlink
        link.href = Metaweb.contentURL(id);          // ...to a full-size image
        link.target = "_new";                        // ...in new window.
        link.appendChild(img);                       // Put image in link.
        container.appendChild(link);                 // Put link in container.
    }
}

// This function is invoked with the results of the document query.
function displayDocuments(docs) {
    container = document.getElementById("docs");     // Get container element.
    for(var i = 0; i < docs.length; i++) {           // Loop through docs.
        var id = docs[i].id;                         // Get the document id.
        var blurb =document.createElement("iframe"); // Create an iframe
        blurb.src = Metaweb.blurbURL(id);            // ...to hold doc blurb.
        var link = document.createElement("a");      // Hyperlink
        link.href = Metaweb.contentURL(id);          // ...to full document
        link.target = "_new";                        // ...in a new window.
        link.innerHTML = docs[i].timestamp;          // Use timestamp as link.
        var listitem = document.createElement("li"); // Create list item.
        listitem.appendChild(blurb);                 // Put blurb in item.
        listitem.appendChild(link);                  // Put link in item.
        container.appendChild(listitem);             // Put item in container.
    }
}
</script>
</head>
<body><!--Static document body. Thumbnails and blurbs dynamically inserted-->
  <h2>The Newest Images</h2>                 <!-- Images heading -->
  <i>Click thumbnail for full-size image</i>  <!-- Instructions -->
  <div id="images"></div>                    <!-- Thumbnails will go here -->
  <h2>The Newest Documents</h2>              <!-- Documents heading -->
  <i>Click timestamp for full document</i>    <!-- Instructions -->
  <ol id="docs"></ol>                        <!-- Document blurbs go here -->
```

```
</body>
</html>
```

4.7.3. Example: A Metaweb Type Browser

Example 4.13 is a JavaScript-based web application that demonstrates the `Metaweb.download()` function defined in Example 4.11. This application is a Metaweb type browser that displays information about any Metaweb type. Figure 4.2 shows a sample page. Clicking on the id of another type (or typing a type id in the upper right) displays information about that type. You may actually find this type browser quite useful for exploring Metaweb system types and the types in other domains.



/type/type

Enter type id: /type/type

Name

Type

Description

the type of which all other types are instances.

Properties

Property Name	Property Key	Property Type
properties	properties	/type/property
instance	instance	/type/object
domain	has_domain, domain	unique /type/domain
expected_by	expected_by	/type/property
default_property	has_default_property_name, default_property	unique /type/rawstring

Used by

Type	Property Key	Property Name
/type/domain	types	types
/type/object	is_instance_of, type	type

Figure 4.2. A Metaweb type browser

In addition to demonstrating document download with `Metaweb.download()`, this example is notable because it uses a more complicated MQL query than the other examples in this chapter and because its HTML output is more complex than previous examples. The code is well-commented, and if you've understood previous JavaScript examples, you should not have trouble following this one. One feature to note is the use of a simple helper class, named `DOMStream`, for dynamically generated output.

Example 4.13. TypeBrowser.html: a Metaweb type browser

```
<html>
<head>
<!-- These are the modules we need -->
<script language="javascript" src="json2.js"></script>
<script language="javascript" src="metaweb.js"></script>
<script language="javascript">

// This is the query we use to get information about a type.
// Note that we have to fill in the id of the type we're interested in.
var query = {
    id:null,            // The type we're asking about. Filled in below.
    type:"/type/type",  // The type of our type is /type/type :-)

    // What is the human-readable type name?
    name:null,

    // Objects with documentation are co-typed /freebase/documented_object.
    "/freebase/documented_object/tip":null,       // Get short description
    "/freebase/documented_object/documentation":{ // And full documentation
         "optional":true,                         // ...if there is any
         "id":null                                // Return document id.
    },

    // Types are also co-typed /freebase/type_profile
    // This property gives us the status (published, private, etc.) of the type
    "/freebase/type_profile/published":null,   // Get publication status

    // What properties does this type have?
    properties:[{
        optional:true,                         // Okay if there are none
        name:null,                             // Property name for display
        key:[],                                // Property name for MQL
        expected_type: {name:null, id:null},   // Type of property value
        unique:null                            // Is it unique?
    }],

    // What other types have properties of this type?
    expected_by:[{
        limit:25,                              // Don't return too many
        optional:true,                         // Okay if there are none
        name:null,                             // The property name
        key:[],                                // The property key
        schema: {name:null, id:null}           // What type defines it?
    }],

    // What are some instances of this type?
    instance:[{
        limit:25,                              // Don't return too many
        optional:true,                         // Okay if there are none
        id:null,                                // So we can link to it
        name:null                               // Object name
    }]
};

// When we're first loaded, display /common/topic, or the type in the URL
window.onload = function() {
    var type = "/common/topic";                         // Assume /common/topic
    var search = window.location.search;
    if (search && search.indexOf("?t=") == 0)           // If URL specifes type
       type = decodeURIComponent(search.substring(3)); // Then use that one
    queryType(type);                                    // Display type info.
}

// Query the specified type.  Call displayType() when the results arrive
function queryType(type) {
    query.id = type;                 // Specify the type in the query above
    Metaweb.read(query,              // Issue the query
                 displayType);       // Pass result object to displayType
}

// Generate a page of information based on our query results.
function displayType(result) {
    // DOMStream is a helper class defined below.
    // We use it here to output HTML text to the placeholder element
    var out = new DOMStream("placeholder");
    out.clear()                              // Erase existing content

    // If we didn't get any results then the input was invalid
    if (!result) {
        out.write("No such type");           // Output failure message
        out.flush();                         // Flush output to placeholder
        return;                              // We're done.
    }

    // Now begin generating information about the type.
    // First, display the type id as the page title.
    out.write("<h1>", result.id, "</h1>");

    // If there is a short description, display it beneath the title.
    var tip = result["/freebase/documented_object/tip"];
    if (tip) out.write("<i>", tip, "</i>");

    // Display the human-readable name of the type
    out.write("<h2>Name</h2>", result.name);

    // Display publication status of the type, if available.
    var status = result["/freebase/type_profile/published"];
    if (status) out.write("<h2>Status</h2>" + status);

    // Display type documentation, if there is any
```

```
out.write("<h2>Documentation</h2>");       // Section header
var doc = result["/freebase/documented_object/documentation"];
if (doc && doc.id) {
    // If there was a document describing the type, output a placeholder
    // element for its content, and issue a request for the content to
    // be inserted into that element.
    out.write("<div id='docplaceholder'><i>Loading...</i></div>");
    Metaweb.download(doc.id, "docplaceholder"); // Asynchronous!
}
else out.write("No documentation available.");

// Display a table of properties
out.write("<h2>Properties</h2>")
if (result.properties.length == 0) out.write("No properties");
else {
    out.write('<table border="1"><tr>',
              '<th>Property Key</th>',
              '<th>Property Name</th>',
              '<th>Property Type</th></tr>');

    for(var i = 0; i < result.properties.length; i++) {
        out.write('<tr><td>', result.properties[i].key.join(", "),
                  '</td><td>', result.properties[i].name,
                  '</td><td>');
        if (result.properties[i].unique) out.write("unique ");
        displayTypeLink(out, result.properties[i].expected_type.id,
                        result.properties[i].expected_type.name);
        out.write('</td></tr>');
    }
    out.write("</table>");
}

// Display the properties of other types that use this type
out.write("<h2>Used by</h2>")
if (result.expected_by.length == 0)
    out.write("There are no Properties of this type.");
else {
    out.write('<table border="1"><tr>',
              '<th>Type</th>',
              '<th>Property Key</th>',
              '<th>Property Name</th>',
              '</tr>');

    for(var i = 0; i < result.expected_by.length; i++) {
        out.write('<tr><td>');
        displayTypeLink(out, result.expected_by[i].schema.id,
                        result.expected_by[i].schema.name);
        out.write('</td><td>', result.expected_by[i].key.join(", "),
                  '</td><td>', result.expected_by[i].name,
                  '</td><tr>');
    }
```

```
        out.write("</table>");
    }


    // Output a list of the names of instances of this type
    out.write("<h2>Instances</h2>");
    if (result.instance.length == 0) out.write("No instances");
    else {
        for(var i = 0; i < result.instance.length; i++) {
            var id = result.instance[i].id;
            var name = result.instance[i].name;
            if (!name) name = id;
            if (i != 0) out.write(", ");
            out.write("<a target='_new' href='http://freebase.com/view",
                      id, "'>", name, "</a>");
        }
    }

    // Calling flush makes the output visible on the page
    out.flush();
}

// Output a link to a type. Use the type id as the link text, and
// make the type name available as a tooltip
function displayTypeLink(out, id, name) {
    out.write('<a title="', name, '" onclick="queryType(\'', id, '\')">',
              id, '</a>');
}

// This little DOMStream class writes HTML into the element we specify
function DOMStream(elt) {                    // Constructor function
    if (typeof elt == "string")              // Expects a DOM element
        elt = document.getElementById(elt);  // or element id string
    this.elt = elt;                          // Remember the element
    this.buffer = [];                        // Array to buffer output
}
DOMStream.prototype.clear = function() {     // Erase element content
    this.elt.innerHTML = "";
};
DOMStream.prototype.write = function() {     // Buffer up all arguments
    this.buffer.push.apply(this.buffer, arguments); // JavaScript voodoo
};
DOMStream.prototype.flush = function() {     // Output all text to the element
    this.elt.innerHTML += this.buffer.join(""); // Concatenate and display
    this.buffer.length = 0;                    // Empty the buffer
};
</script>

<style>
/* Some CSS styles to make everything look good */
body {
  font-family: Arial, Helvetica, sans-serif; /* We like sans-serif */
  margin-left: .5in;                         /* Indent everything... */
}
h1, h2 {  margin-left: -.25in; }             /* ...except headings */
h2 { margin-bottom: 5px; margin-top:10px; }
/* Make tables look nice */
table { border-collapse: collapse; width: 95%;}
th { background-color: #aaa;}
td { background-color: #ddd; padding: 1px 5px 1px 5px; }

/* Our <a> tags don't have hrefs, so we need to style them ourselves */
a { color: #00a; }
a:hover { text-decoration:underline; cursor:pointer;}

/* Make the input field look nice */
form.inputform {
  float:right; border: solid black 2px; background-color: #aba;
  margin: 15px 30px 0px 0px; padding: 10px;
}
</style>
</head>
<body>
<!-- A form in which the user can enter a type id -->
<form class='inputform' onsubmit="queryType(this.t.value); return false;">
Enter type id: <input name="t"></form>
<!-- Generated content goes here -->
<div id="placeholder"></div>
</body>
</html>
```

4.8. Metaweb Services with Python

As a final, advanced example of the use of the *mqlread*, *search*, and *trans* services, Example 4.15 presents a Python module for working with Metaweb services. This module defines a `metaweb.Session` object that represents the host name of a Metaweb server, and also encapsulates a set of options (such as the `lang` option to *mqlread* and the `maxwidth` option to */api/trans/image_thumb*). Each `Session` object also maintains a "cookie jar" for storing any HTTP cookies returned by Metaweb services, and uses those cookies in any subsequent requests made. (Cookies are used for authentication and caching, and are discussed in Chapter 6.)

One feature of Example 4.15 is of particular note. The `results()` method of the `metaweb.Session` class is a generator that returns the results of a MQL query one at a time, and uses the `cursor` envelope parameter to submit a MQL query as many times as necessary to retrieve all available results. You might use it in code like this:

```
import metaweb                                 # Use the metaweb module
freebase = metaweb.Session("api.freebase.com") # Create a Session object
albums_by_bob = [{'type':'/music/album',       # This is our MQL query
                  'artist':'Bob Dylan',
                  'name':None }]
for album in freebase.results(albums_by_bob):  # Loop through query results
    print album["name"]                        # Print album names
```

Example 4.14 is a Python version of Example 4.10: it lists albums by a specified band or searches for bands whose name is like a specified string. It demonstrates the metaweb.py module in more detail, showing how to use the `read()` method to invoke *mqlread*, the `search()` method to invoke the *search* service, and the `blurb()` method to invoke the */api/trans/blurb* service.

Example 4.14. Using the metaweb.py module to read, search and download

```
import sys           # Command-line arguments, etc.
import metaweb       # Metaweb services

band = sys.argv[1]                       # The band the user is asking about
query = { 'type': '/music/artist',       # Our MQL query in Python.
          'name': band,                  # Place the band in the query.
          'album': [{ 'name': None,
                      'release_date': None,
                      'sort': 'release_date' }]}

freebase = metaweb.Session("api.freebase.com") # Create a session object
result = freebase.read(query)                  # Submit query, get results
if result:                                     # If we got a result
    print("Albums by %s:" % result['name'])    # print the album names
    print("\n".join([album['name'] for album in result['album']]))
else:                                          # Otherwise: no result
    matches=freebase.search(band + "*",        # Start a search
                        type="/music/artist",  # Only for bands
                        limit=5)               # We only want 5
    if (len(matches) == 0):                    # If no search results
```

```
        print "Unknown band."                    # Give up.
    else:                                        # If we got some search results
        print "Did you mean one of these?:"
        for match in matches:                    # Loop through the matches
            print
            print match['name']                  # Print the name of the match
            article = match['article']           # Get associated article
            if article:                          # If there is an article
                text,type = freebase.blurb(article['id'],  # Download a blurb
                                           maxlength=100)  # 100 chars long
                print text;                                # And print it
```

With those usage examples behind us, we end this chapter with the *metaweb.py* code. Note that only read methods are shown here. We'll add methods for making MQL write queries and for uploading content in Chapter 6.

Example 4.15. metaweb.py: a Python module for Metaweb

```
#
# metaweb.py: A python module for writing Metaweb-enabled applications
#
"""
This module defines classes for working with Metaweb databases.

  metaweb.Session: represents a connection to a database
  metaweb.ServiceError: exception raised by Session methods

Typical usage:

    import metaweb
    freebase = metaweb.Session("api.freebase.com")
    q1 = [{ 'type':'/music/album', 'artist':'Bob Dylan', 'name':None }]
    q2 = [{ 'type':'/music/album', 'artist':'Bruce Springsteen', 'name':None }]
    bob,bruce = freebase.read(q1, q2)  # Submit two queries, get two results
    for album in bob: print album['name']

    # Get query results with a generator method instead
    albums = freebase.results(q2)
    albumnames = (album['name'] for album in albums)
    for name in albumnames: print name

    # Download an image of U2
    result = freebase.read({"id":"/en/u2","/common/topic/image":[{"id":None}]})
    imageid = result["/common/topic/image"][0]["id"]
    data,type = freebase.download(imageid)
    print "%s image, %d bytes long" % (type, len(data))

"""

import urllib        # URL encoding
import simplejson    # JSON serialization and parsing
import urllib2       # URL content fetching
import cookielib     # HTTP Cookie handling

# Metaweb read services
READ = '/api/service/mqlread'    # Path to mqlread service
SEARCH = '/api/service/search'   # Path to search service
DOWNLOAD = '/api/trans/raw'      # Path to download service
BLURB = '/api/trans/blurb'       # Path to document blurb service
THUMB = '/api/trans/image_thumb' # Path to image thumbnail service

# Metaweb write services
LOGIN = '/api/account/login'     # Path to login service
WRITE = '/api/service/mqlwrite'  # Path to mqlwrite service
UPLOAD = '/api/service/upload'   # Path to upload service
TOUCH = '/api/service/touch'     # Path to touch service

# Metaweb services return this code on success
OK = '/api/status/ok'

class Session(object):
    """
    This class represents a connection to a Metaweb database.

    It defines methods for submitting read, write and search queries to the
    database and methods for uploading and downloading binary data.
    It encapsulates the database URL (hostname and port), read and write
    options, and maintains authentication and cache-related cookies.

    The Session class defines these methods:

      read(): issue one or more MQL queries to mqlread
      results(): a generator that performs a MQL query using a cursor
      search(): invoke the search service
      download(): retrieve content with trans/raw
      contenURL(): like download(), but just return the URL
      blurb(): retrieve a document blurb with trans/blurb
      blurbURL(): like blurb(), but just return the URL
      thumbnail(): retrieve an image thumbnail with trans/image_thumb
      thumbnailURL(): like thumbnail(), but just return the URL
      login(): establish credentials (as a cookie) for writes
      write(): invoke mqlwrite
      upload(): upload content
      touch(): get a fresh mwLastWriteTime cookie to defeat caching

    Each Session instance has these read/write attributes:

      host: the hostname (and optional port) of the Metaweb server as a string.
         The default is sandbox.freebase.com.  Every Monday, the sandbox is
         erased and it is updated with a fresh copy of data from
         www.freebase.com.  This makes it an ideal place to experiment.
      cookiejar: a cookielib.CookieJar object for storing cookies.
         If none is passed when the class is created, a
         cookielib.FileCookieJar is automatically created.  Note that cookies
         are not automatically loaded into or saved from this cookie jar,
         however. Clients that want to maintain authentication or cache state
         across invocations must save and load cookies themselves.

      options: a dict mapping option names to option values. Key/value
         pairs in this dict are used as envelope or URL parameters by
         methods that need them.  The read() method looks for a lang
         option, for example and the image_thumb looks for a maxwidth
         option. Options may be passed as named parameters to the Session()
         constructor or to the various Session methods.
    """

    def __init__(self, host="sandbox.freebase.com", cookiejar=None, **options):
        """Session constructor method"""
        self.host = host
        self.cookiejar = cookiejar or cookielib.FileCookieJar()
        self.options = options

    def read(self, *queries, **options):
        """
        Submit one or more MQL queries to a Metaweb database, using any
        named options to override the option defaults. If there is
        a single query, return the results of that query. Otherwise, return
        an array of query results. Raises ServiceError if there were problems
        with any of the queries.
        """

        # How many queries are we handling?
        n = len(queries)

        # Gather options that apply to these queries
        opts = self._getopts("lang", "as_of_time", "escape",
                             "uniqueness_failure", **options)

        # Create an outer envelope object
        outer = {}

        # Build the inner envelope for each query and put it in the outer.
        for i in range(0, n):
            inner = {'query': queries[i]} # Inner envelope holds a query.
            inner.update(opts)            # Add envelope options.
            outer['q%d' % i] = inner      # Put inner in outer with name q(n).

        # Convert outer envelope to a string
        json = self._dumpjson(outer)

        # Encode the query string as a URL parameter and create a url
        urlparam = urllib.urlencode({'queries': json})
        url = 'http://%s%s?%s' % (self.host, READ, urlparam)

        # Fetch the URL contents, parse to a JSON object and check for errors.
        # From here on outer and inner refer to response, not query, envelopes.
        outer = self._check(self._fetch(url))

        # Extract results from the response envelope and return in an array.
        # If any individual query returned an error, raise a ServiceError.
        results = []
        for i in range(0, n):
            inner = outer["q%d" % i]         # Get inner envelope from outer
            self._check(inner)               # Check inner for errors
            results.append(inner['result'])  # Get query result from inner

        # If there was just one query, return its results.  Otherwise
        # return the array of results
        if n == 1:
            return results[0]
        else:
            return results

    def results(self, query, **options):
        """
        A generator version of the read() method. It accepts a single
        query, and yields query results one by one. It uses the envelope
        cursor parameter to return a full set of results even when more
        than one invocation of mqlread is required.
        """

        # Gather options that apply to this query
        opts = self._getopts("lang", "as_of_time", "escape",
                             "uniqueness_failure", **options)

        # Build the query envelope
        envelope = {'query': query}
        envelope.update(opts)

        # Start with cursor set to true
        cursor = True

        # Loop until cursor is no longer true
        while cursor:
            # Use the cursor as an envelope parameter
            envelope['cursor'] = cursor

            # JSON-encode the envelope and convert it to a URL parameter
            params=urllib.urlencode({'query': self._dumpjson(envelope)})

            # Build the URL
            url = 'http://%s%s?%s' % (self.host, READ, params)
            # Fetch and parse the URL contents, raising ServiceError on errors
            response = self._check(self._fetch(url))

            # Get the results array and yield one result at a time
            results = response['result']
            for r in results:
                yield r

            # Get the new value of the cursor for the next iteration
            cursor = response['cursor']


    def search(self, query, **options):
        """
        Invoke the search service for the specified query string.  If that
        string ends with an asterisk, perform a prefix search instead of a
        straight query.  type, domain, type_strict, and other search service
        options may be specified as Session options or may be passed as named
        parameters.
        """
        opts = self._getopts("domain", "type", # Build a dict of search options
                             "type_strict",    # from these session options
                             "limit", "start",
                             "escape", "mql_output",
                             **options)        # plus any passed to this method

        if query.endswith('*'):           # If search string ends with *
            opts["prefix"] = query[0:-1]  # then this is a prefix search
        else:                             # Otherwise...
            opts["query"] = query         # It is a regular query

        params = urllib.urlencode(opts)                  # Encode options
        url = "http://%s%s?%s" % (self.host,SEARCH,params) # Build URL
        envelope = self._fetch(url)                      # Fetch response
        self._check(envelope)                            # Check that its OK
        return envelope["result"]                        # Return result


    def download(self, id):
        """
        Return the content and type of the content object identified by id.

        Returns two values: the downloaded content (as a string of characters
        or bytes) and the type of that content (as a MIME-type string, from
        the Content-Type header returned by the Metaweb server). Raises
        ServiceError if the request fails with a useful message; otherwise
        raises urllib2.HTTPError.  See also the contentURL() method.
        """
        return self._trans(self.contentURL(id))
```

```
def blurb(self, id, **options):
    """
    Return the content and type of a document blurb.  See blurbURL().
    """
    return self._trans(self.blurbURL(id, **options))

def thumbnail(self, id, **options):
    """
    Return the content (as a binary string) and type of an image
    thumbnail.  See thumbnailURL().
    """
    return self._trans(self.thumbnailURL(id, **options))


def contentURL(self, id):
    """
    Return the /api/trans URL of the /type/content, /common/image,
    or /common/document content identified by the id argument.
    """
    return self._transURL(id, DOWNLOAD)

def blurbURL(self, id, **options):
    """
    Return the /api/trans URL of a blurb of the document identified by id.

    The id must refer to a /type/content or /common/document object.
    Blurb length and paragraph breaks are controlled by maxlength and
    break_paragraph options, which can be specified in the Session object
    or passed to this method.
    """
    return self._transURL(id, BLURB, ["maxlength", "break_paragraphs"],
                          options)

def thumbnailURL(self, id, **options):
    """
    Return the URL of a thumbnail of the image identified by id.

    The id must refer to a /type/content or /common/image object.
    Thumbnail width and height are controlled by the maxwidth and
    maxheight options, which can be specified on the Session object, or
    passed to this method.
    """
    return self._transURL(id, THUMBNAIL, ["maxwidth","maxheight"], options)


# A utility method that returns a dict of options.
# It first builds a dict containing only the specified keys and their
# values, and only if those keys exist in self.options.
# Then it augments this dict with the specified options.
def _getopts(self, *keys, **local_options):
    o = {}
```

```
        for k in keys:
            if k in self.options:
                o[k] = self.options[k]
        o.update(local_options)
        return o

    # Fetch the contents of the requested HTTP URL, handling cookies from
    # the cookie jar.  Return a tuple of http status, headers and
    # response body. This is the only method in this module that performs
    # HTTP or manages cookies.  This implementation uses the urllib2 library.
    # You can subclass and override this method if you want to use a
    # different implementation with different performance characteristics.
    def _http(self, url, headers={}, body=None):
        # Store the url in case we need it later for error reporting.
        # Note that this is not safe if multiple threads use the same Session.
        self.lasturl = url;

        # Build the request.  Will use POST if a body is supplied
        request = urllib2.Request(url, body, headers)

        # Add any cookies in the cookiejar to this request
        self.cookiejar.add_cookie_header(request)

        try:
            stream = urllib2.urlopen(request) # Try to open the URL, get stream
            self.cookiejar.extract_cookies(stream, request) # Remember cookies
            headers = stream.info()
            body = stream.read()
            return (stream.code, headers, body)
        except urllib2.HTTPError, e:          # If we get an HTTP error code
            return (e.code, e.info(), e.read())  # But return values as above


    # Parse a string of JSON text and return an object, or raise
    # InternalServiceError if the text is unparseable.
    # This implementation uses the simplejson library.
    # You can override it in a subclass if you want to use something else.
    def _parsejson(self, s):
        try:
            return simplejson.loads(s)
        except:
            # If we couldn't parse the response body, then we probably have an
            # low-level HTTP error with no JSON in its response. This should
            # not happen, but if it does, we createa a fake response object
            # so that we can raise a ServiceError as we do elsewhere.
            raise InternalServiceError(self.lasturl, s)

    # Encode the object o as JSON and return the encoded text.
    # If pretty is True, use line breaks and indentation to make the output
    # more human-readable.  Override this method if you want to use an
    # implementation other than simplejson.
```

```
def _dumpjson(self, o, pretty=False):
    if pretty:
        return simplejson.dumps(o, indent=4)
    else:
        return simplejson.dumps(o)

# An internal utility function to fetch the contents of a Metaweb service
# URL and parse the JSON results and return the resulting object.
#
# Metaweb services normally return JSON response bodies even when an HTTP
# error occurs, and this function parses and returns those error objects.
# It only raises an error on very low-level HTTP errors that do
# not include a JSON object as its body.
def _fetch(self, url, headers={}, body=None):
    # Fetch the URL contents
    (status, headers, body) = self._http(url, headers, body);
    # Parse the response body as JSON, and return the resulting object.
    return self._parsejson(body)

# This is a utility method used by download(), blurb() and thumbnail()
# to fetch the content and type of a specified URL, performing
# cookie management and error handling the way the _fetch function does.
# Unlike other Metaweb services, trans does not normally return a JSON
# object, so we cannot just use _fetch here.
def _trans(self, url):
    (status,headers,body) = self._http(url)   # Fetch url content
    if (status == 200):                       # If successful
        return body,headers['content-type']   # Return content and type
    else:                                     # HTTP status other than 200
        errobj = self._parsejson(body)        # Parse the body
        raise ServiceError(url, errobj)       # And raise ServiceError

# An internal utility function to check the status code of a Metaweb
# response envelope and raise a ServiceError if it is not okay.
# Returns the response if no error.
def _check(self, response):
    code = response['code']
    if code != OK:
        raise ServiceError(self.lasturl, response)
    else:
        return response

# This utility method returns a URL for the trans service
def _transURL(self, id, service, option_keys=[], options={}):
    url = "http://" + self.host + service + id    # Base URL.
    opts = self._getopts(*option_keys, **options) # Get request options.
    if len(opts) > 0:                             # If there are options...
        url += "?" + urllib.urlencode(opts)       # encode and add to url.
    return url
```

```
class ServiceError(Exception):
    """
    This exception class represents an error from a Metaweb service.

    When anything goes wrong with a Metaweb service, it returns a response
    object that includes an array of message objects.  When this occurs we
    wrap the entire response object in a ServiceError exception along
    with the URL that was requested.

    A ServiceError exception converts to a string that contains the
    requested URL (minus any URL parameters that contain the actual
    query details) plus the status code and message of the first (and
    usually only) message in the response.

    The details attribute provides direct access to the complete response
    object. The url attribute provides access to the full url.
    """

    # This constructor expects the URL requested and the parsed response.
    def __init__(self, url, details):
        self.url = url
        self.details = details

    # Convert to a string by printing url + the first error code and message
    def __str__(self):
        prefix = self.url.partition('?')[0]
        msg = self.details['messages'][0]
        return prefix + ": " + msg['code'] + ": " + msg['message']


class InternalServiceError(ServiceError):
    """
    A ServiceError with a fake response object. We raise one of these when
    we get an error so low-level that the HTTP response body is not a
    JSON object.  In this case we basically just report the HTTP error code.
    An exception of this type probably indicates a bug in this module.
    """
    def __init__(self, url, body):
        ServiceError.__init__(self, url,
                              {'code':'Internal service error',
                               'messages':[{'code':'Unparseable response',
                                            'message':body}]
                               })
```

5

The MQL Write Grammar

Insertions, deletions and updates to a Metaweb database are expressed in a variant of the Metaweb Query Language that was documented in Chapter 3. The variant used for writing to Metaweb is known as the MQL write grammar, and is the subject of this chapter. Write queries are submitted to Metaweb via the *mqlwrite* service, which is covered in Chapter 6.

MQL writes are represented as JSON objects, just as MQL reads are. A number of features of the MQL read grammar only make sense for reads and are not allowed in MQL writes. These include:

- the use of `[]` to query an array of values,
- the use of directives like `sort`, `limit` `return` and `optional`, and
- the use of operators like `~=`.

The MQL write grammar does allow property prefixes like the read grammar does (though these are not often useful in writes) and defines two write-specific directives that are not allowed for reads. The `create` directive is used to create a new object in the database, and the `connect` directive is used to create a link between two objects. (As we'll see in the tutorial, however, the `connect` directive is sometimes implicit and need not be specified explicitly).

Using this Tutorial

The best way to follow this tutorial is to try out the queries as you read about them. While you learn how to make MQL writes, please use the sandbox server at *http://sandbox.freebase.com*. This server is intended for experimentation. The sandbox hosts a replica of *freebase.com*, and this replica is re-created once a week. This means that any writes you perform (or mistakes you make!) on the sandbox will not persist longer than a week.

Anyone can read data from Freebase, but before you can execute write queries, you must register for an account and login. If you already have an account at *www.freebase.com*, it may already have been replicated on the sandbox server. If not, you can create a new account for yourself on the sandbox. Follow the links from the *http://sandbox.freebase.com/* homepage to register.

Once you are logged on to the sandbox, you can execute MQL write queries using the the Freebase query editor at *http://sandbox.freebase.com/tools/queryeditor*. Enter queries from this tutorial, click the **write** button, and view the results.

5.1. Creating a Type to Work With

Before we do any explicit MQL writes, let's begin by creating a simple type to work with. By creating and using your own type, you guarantee that the writes you try while working through this tutorial won't interact with writes being issued by other developers who may be working on the tutorial at the same time. As you know, Metaweb types are defined by regular Metaweb objects in the database. This means that types are created like any other objects, with MQL queries. Defining a type with raw MQL is difficult and error prone, however, so the *freebase.com* client provides an easier way to do it.

The type we're creating will represent musical notes, and we'll call it "note". Sign in to *sandbox.freebase.com* (creating an account if you have not already done so). From your homepage you can follow links to your default domain where you can create this new note type. Optionally, you may first want to create a new domain named "music" and put the note type into that domain. Don't add any properties to your new type yet. We'll do that later in this tutorial. This manual is focused on MQL and does not attempt to provide step-by-step descriptions of how to use the Freebase client to create domains, types, and properties. The Freebase UI is reasonably intuitive, however, and you can find detailed documentation at *http://www.freebase.com/help*.

The Freebase UI displays the names of types and their domains. To write MQL queries, however, we need to know type ids. Each Freebase user has a namespace of the form `/user/name`. Each user account initially has one domain, named `default_domain` under their namespace. If your username is "wanda", then the id of your default domain is `/user/wanda/default_domain`. If you used this default domain when creating the note type, the id of that type will be `/user/wanda/default_domain/note`. If you created a new domain named `music`, then the note type will have id `/user/wanda/music/note`. If you display details about your type in the Freebase client, you'll find the id of the type embedded in the URL displayed in your web browser's location bar. It might look like this:

```
http://sandbox.freebase.com/view/user/wanda/music/note
```

For the rest of this chapter, the queries will use types in the domain `/user/docs/music`. When you run the queries, replace "docs" with your own username, and, if you did not create your own music domain, replace "music" with "default_domain".

5.2. Creating Objects

Let's begin with a very simple write query:

Write	Result
`{`	`{`
`  "create":"unless_exists",`	`  "create":"created",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"A",`	`  "name":"A",`
`  "id":null`	`  "id":"/guid/1f8000000000037ffc"`
`}`	`}`

The first line of the query says that we want to create a new object, unless a matching object already exists. The second line specifies the type of the object we're creating (remember to substitute your own user name for "docs" here). The third line specifies a value for the `name` property of the new object. The fourth line of the write query is a request for the `id` of the newly created object. Asking for an id is the only way you are allowed to use `null` in a write query. You may not use `null` (or `[]`) as the value of any property other than `id` (or `guid`).

Now let's look at the response to the write query. The first line is the `create` property, but its value has changed from `unless_exists` to `created`. This tells us that the object we specified did not already exist, and Metaweb has created it for us. The second and third lines simply repeat the `type` and `name` properties that we passed in. They don't provide any new information, but maintain the MQL invariant that responses have the same properties as queries. Finally, the fourth line returns the id of the newly created object.

Metaweb IDs in this Tutorial

The object ids used in this tutorial were created on the sandbox server, and are no longer valid, so you should not try to query these objects directly. Instead, substitute your own user name into the write queries, and create your own objects, with their own ids, as you follow along with this tutorial. If you are reading a printed or PDF version of this chapter, note that ids in the `/guid` namespace have been shortened so that they fit more easily in two-column format.

Now let's see what happens if we run exactly the same query again:

Write	Result
`{`	`{`
`  "create":"unless_exists",`	`  "create":"existed",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"A",`	`  "name":"A",`
`  "id":null`	`  "id":"/guid/1f8000000000037ffc"`
`}`	`}`

We're asking that an object be created unless it already exists. And this time it does already exist. So Metaweb returns the `existed` as the value of the `create` property, and returns the `id` of the already existing object. Note that this id is the same as the one we've already seen.

Now let's force Metaweb to create another new test object for us:

Write	Result
`{`	`{`
`  "create":"unconditional",`	`  "create":"created",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"A",`	`  "name":"A",`
`  "id":null`	`  "id":"/guid/1f800000000003800f"`
`}`	`}`

In this query, we've changed the value of the `create` directive to `unconditional`. As its name implies, this value tells Metaweb to create a new object no matter what. Since a new object is created unconditionally, the value of the `create` property in the response will always be `created`. You can see that a new object was created by comparing the `id` returned by this query to those returned by the previous two queries.

When to use create:unconditional

Metaweb is an "identity database", intended to have only one copy of each object. So it is usually not necessary or correct to use `"create":"unconditional"` in a MQL write query. Most writes use `"create":"unless_exists"`, and many others use `"create":"unless_connected"`, which has not been introduced yet. Using "unconditional" leads to duplicate objects, and as we'll see below, this can get you into trouble!

We now have two note objects with the name "A". What happens if we run the original `unless_exists` write again?

```
{
  "code" : "/api/status/error",
  "messages" : [
    {
      "code" : "/api/status/error/mql/result",
      "info" : {
        "count" : 2,
        "guids" : [
          "#1f8000000000037ffc",
          "#1f800000000003800f"
        ]
      },
      "message" : "Need a unique result to attach here, not 2",
      "path" : "",
      "query" : {
        "create" : "unless_exists",
        "error_inside" : ".",
        "id" : null,
        "name" : "A",
        "type" : "/user/docs/music/note"
      }
    }
  ]
}
```

The query fails this time, and returns the JSON object shown above. The `"create":"unless_exists"` directive works only if there are 0 or 1 instances of the object. If there is no object that matches, it creates one. If there is one object that matches, it returns it. But if there are more than one, it has no way to choose which one to return, and fails with an error message. Note that the query fails even if we omit `"id":null`. The lesson here is that if you plan to use `unless_exists`, you should use it consistently so you never end up with more than one instance of an object.

5.3. Connecting Objects

So far we've created two distinct objects with identical types and names. Let's now rename one so we can tell them apart by name. Recall that an object is named by linking it to a primitive value of `/type/text`. We want to update the `name` link to refer to a different value:

Write	Result
`{`	`{`
`  "id":"/guid/1f800000000003800f",`	`  "id":"/guid/1f800000000003800f",`
`  "name":{`	`  "name":{`
`    "connect":"update",`	`    "connect":"updated",`
`    "value":"B",`	`    "value":"B",`
`    "lang":"/lang/en"`	`    "lang":"/lang/en"`
`  }`	`  }`
`}`	`}`

The first line of the query identifies, by id, the object we want to modify. The second and third lines specify that want to update the `name` property of that object so that it refers to the `/type/text` value specified by the 4th and 5th lines. (Recall that `/type/text` is a primitive value that consists of a string of text and a language identifier for that text. MQL write queries require you to specify both the `value` and `lang` properties when manipulating a name.)

The response looks just like the query except that the value of the `connect` property has changed to `updated`. This tells us that the update we requested has been performed.

What happens if we run exactly the same write query again?

Write	Result
`{`	`{`
`  "id":"/guid/1f800000000003800f",`	`  "id":"/guid/1f800000000003800f",`
`  "name":{`	`  "name":{`
`    "connect":"update",`	`    "connect":"present",`
`    "value":"B",`	`    "value":"B",`
`    "lang":"/lang/en"`	`    "lang":"/lang/en"`
`  }`	`  }`
`}`	`}`

We're asking to make a change that has already been made, and Metaweb lets us know this by setting the `connect` property of the response to `present`.

We now have two newly-created objects with the same type and different names. We changed the name of the second object by updating a `/type/text` value. `/type/text` is a primitive type in Metaweb, so this isn't quite the same thing as a link between two different objects in the database. Now, let's modify the first object (the note A) so that it is a `/common/topic` in addition to being a note:

```
Write                                Result
{                                    {
  "id":"/guid/1f8000000000037ffc",     "id":"/guid/1f8000000000037ffc",
  "type":{                             "type":{
    "connect":"insert",                  "connect":"inserted",
    "id":"/common/topic"                 "id":"/common/topic"
  }                                    }
}                                    }
```

The first line of the query specifies the object to be modified. The second and third lines specify that we want to insert a new connection between this object and another object, and that this new connection should use the `type` property. The fourth line specifies, by id, the object that is being connected to.

Note that the value of the `connect` directive is `insert` instead of `update`, which is what we used above. The difference between the two is simple. Use `"connect":"update"` for properties that have a unique value (and for the `name` property, which is unique on a per-language basis). Use `"connect":"insert"` for properties, such as `type`, that can have more than one value. You are also allowed to use `"connect":"insert"` with unique properties if there is not already a value for that property. In scripts that query links or use reflection (see §3.7) you may sometimes have to craft a MQL write query without knowing whether a particular property is unique or not. In that case, use "replace" instead of "update" or "insert". `"connect":"replace"` does an update for unique properties and does an insert for non-unique properties. `"connect":"replace"` is also useful if you have reason to believe that a property that is currently unique may in the future be relaxed to allow multiple values.

The response to the query above sets the value of the `connect` directive to `inserted`, telling us that the insertion was successful. Our note named "A" is now also a `/common/topic`, which means that we can associate images, documents and aliases with the object.

What happens if we run the same query again?

```
Write                                Result
{                                    {
  "id":"/guid/1f8000000000037ffc",     "id":"/guid/1f8000000000037ffc",
  "type":{                             "type":{
    "connect":"insert",                  "connect":"present",
    "id":"/common/topic"                 "id":"/common/topic"
  }                                    }
}                                    }
```

We're asking to insert `/common/topic` into a set of types that already includes `/common/topic`, and we get the response `present`. It tells us that this value is already in the set and that nothing has changed. (Non-unique properties in Metaweb are like sets: they do not allow duplicates.)

Idempotent Writes

One of the important features of writes in Metaweb (assuming you avoid unconditional node creation) is that they are *idempotent*: you can safely make a write request multiple times. A write request can be thought of as an assertion of the desired state of the database. If the database is already in that state, nothing is done. Otherwise, insertions and deletions are performed as necessary to make the database match the assertion.

Let's do a quick read query to confirm that our object is a member of two types:

Read	Result

`{`
`  "id":"/guid/1f8000000000037ffc",`
`  "type":[]`
`}`
	`{`
	`  "id":"/guid/1f8000000000037ffc",`
	`  "type":[`
	`    "/user/docs/music/note",`
	`    "/common/topic"`
	`  ]`
	`}`

So we see that our object is, in fact, a note and a topic.

5.4. Disconnecting Objects

We've seen that Metaweb allows us to connect objects with `"connect":"insert"` or `"connect":"update"`. To disconnect objects, use `"connect":"delete"`. Let's alter the object that represents the note A again, to remove `/common/topic` from its set of types:

Write	Result

`{`
`  "id":"/guid/1f8000000000037ffc",`
`  "type":{`
`    "connect":"delete",`
`    "id":"/common/topic"`
`  }`
`}`
	`{`
	`  "id":"/guid/1f8000000000037ffc",`
	`  "type":{`
	`    "connect":"deleted",`
	`    "id":"/common/topic"`
	`  },`
	`}`

This query looks just like the query we used to add the type, except that we've changed "insert" to "delete". And Metaweb's response looks just like the response to the insertion, except that "inserted" has changed to "deleted".

At this point, you probably have a pretty good idea what will happen if we re-run the query:

```
Write                                   Result
{                                       {
  "id":"/guid/1f8000000000037ffc",        "id":"/guid/1f8000000000037ffc",
  "type":{                                "type":{
    "connect":"delete",                     "connect":"absent",
    "id":"/common/topic"                    "id":"/common/topic"
  }                                       }
}                                       }
```

We asked Metaweb to remove `/common/topic` from a set that did not contain `/common/topic`, so it returned "absent" to indicate that nothing has been changed.

The MQL write grammar has no syntax for deleting objects themselves. The closest thing to deleting an object is to delete all connections from that object to others. If an object has no type, no name, and no other properties of interest, then it becomes effectively unreachable, and is almost as good as gone. [1]

When an object has had all its links deleted, it can still be queried by `id`, `guid` or `creator` (Metaweb does not allow these read-only properties to be deleted.) In practice, however, unreachable objects will only be found by determined searchers, and their continued existence is very unlikely to affect the results of future queries. Unreachable objects may at some point be purged from a Metaweb database, but their guids will *never* be reused.

Let's use this unlinking technique to "delete" the two note objects we've created:

```
Write                                 Result
[{                                    [{
  "id":"/guid/1f8000000000037ffc",      "id":"/guid/1f8000000000037ffc",
  "type":{                              "type":{
    "connect":"delete",                   "connect":"deleted",
    "id":"/user/docs/music/note"          "id":"/user/docs/music/note"
  },                                    },
  "name":{                              "name":{
    "connect":"delete",                   "connect":"deleted",
    "value":"A",                          "value":"A",
    "lang":"/lang/en"                     "lang":"/lang/en"
  }                                     }
},{                                   },{
  "id":"/guid/1f800000000003800f",      "id":"/guid/1f800000000003800f",
  "type":{                              "type":{
    "connect":"delete",                   "connect":"deleted",
    "id":"/user/docs/music/note"          "id":"/user/docs/music/note"
  },                                    },
  "name":{                              "name":{
    "connect":"delete",                   "connect":"deleted",
    "value":"B",                          "value":"B",
```

[1] Remember, however, that Metaweb maintains a modification history for each object. We learned how to query the history of an object and the historical state of an object in §3.7.4 and §4.2.4.4. The *freebase.com* client also makes object history available through links on the pages it displays.

```
Write                                Result
    "lang":"/lang/en"                    "lang":"/lang/en"
  }                                    }
}]                                   }]
```

Note that this write query is really two separate queries, included within square brackets. The *mqlwrite* service (the topic of Chapter 6) accepts submissions of multiple writes at once. Note that names are deleted with `"connect":"delete"`, even though they are unique and were originally created with `"connect":"update"`. You must specify the `lang` property explicitly when deleting a name.

As a final test, let's query the first of these objects and find out what little information it still carries:

```
Read                                   Result
{                                      {
  "id":"/guid/1f8000000000037ffc",       "id" : "/guid/1f8000000000037ffc",
  "*":null,                              "guid" : "#1f8000000000037ffc",
  "/type/reflect/any_master": [{         "type" : [],
      "id":null,                         "name" : null,
      "link":null,                       "key" : [],
      "optional":true                    "creator" : "/user/docs",
  }],                                    "permission" : "/boot/all_permission",
  "/type/reflect/any_reverse": [{        "timestamp" : "2008-08-29T23:42:08.0000Z",
      "id":null,                         "/type/reflect/any_master" : [{
      "link":null,                           "id" : "/boot/all_permission",
      "optional":true                        "link" : "/type/object/permission"
  }],                                    }],
  "/type/reflect/any_value": [{          "/type/reflect/any_reverse" : [],
      "link":null,                       "/type/reflect/any_value" : []
      "optional":true,                 }
      "value":null
  }]
}
```

As expected, the name and types of the object are gone. All that remains are its id, creator, timestamp, and permission.

Multiple Queries and Atomicity

When you submit multiple top-level write queries to Metaweb at the same time, it is natural to ask whether they are executed in order, and whether a query can depend on an object created by a previous query. The answer to both questions is no. The reason is a good one, however: when multiple queries are submitted at the same time, they are executed atomically: all are executed or none are executed.

In order to implement this atomic behavior, the Metaweb server first tests each query to determine whether it will succeed. It does this without actually executing the query. If all queries pass the test, then all are executed. Note, however, that this means that each query must be able to succeed before any other queries have been run. Therefore, the queries must be completely independent of each other. And since they are independent, there is really no way to tell what order Metaweb executes them in.

Although Metaweb allows multiple writes to be executed atomically, Metaweb engineers recommend keeping your writes simple and executing them one at a time whenever you can. This avoids server timeouts and makes debugging much easier if something goes wrong!

5.5. Writes and Default Properties

Take a look again at the MQL write queries we use to create and "delete" Note objects. First, the creation:

Write	Result
`{`	`{`
`  "create":"unless_exists",`	`  "create":"created",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"C#",`	`  "name":"C#",`
`  "id":null`	`  "id":"/guid/1f800000000104befe"`
`}`	`}`

Now contrast this with the query that "deletes" the object by unlinking its type and name:

Write	Result
`{`	`{`
`  "id":"/guid/1f800000000104befe",`	`  "id":"/guid/1f800000000104befe",`
`  "type":{`	`  "type":{`
`    "connect":"delete",`	`    "connect":"deleted",`
`    "id":"/user/docs/music/note"`	`    "id":"/user/docs/music/note"`
`  },`	`  },`
`  "name":{`	`  "name":{`
`    "connect":"delete",`	`    "connect":"deleted",`
`    "value":"C#",`	`    "value":"C#",`
`    "lang":"/lang/en"`	`    "lang":"/lang/en"`
`  }`	`  }`
`}`	`}`

The creation query is much more compact because we are able to specify the type as a single id and the name as a single string. In the deletion query, we must specify the expanded objects. There are three factors that interact to make the creation query shorter. First, recall from Chapter 3 that every type has a *default property*. For value types such as `/type/text` (the type of the `name` property) the default property is `value`. For core types in the `/type` domain, the default property is `id`. For all other types, the default property is `name`. So in the creation query, `"type":"/user/docs/music/note"` is shorthand (but see the caution below!) for:

```
"type": { "id":"/user/docs/music/note" }
```

The second factor that makes the creation query so compact is the fact that when you specify a default property rather than a full object in a MQL write query, Metaweb assumes an implicit `"connect":"insert"`. So writing `"type":"/user/docs/music/note"` is kind of (but not exactly: see the caution that follows) like writing:

```
"type": {
  "connect":"insert",
  "id":"/user/docs/music/note"
}
```

The third factor that makes the creation query compact is that the language of `/type/text` values is automatically set to the default of English.

All three factors come into play when we write `"name":"C#"`. "C#" becomes the value of the default property, which is the `value`. An implicit `"connect":"insert"` is added. And a `lang` property is added to specify `/lang/en`, or whatever language we are using. So `"name":"C#"` expands to (but see the caution!):

```
"name": {
  "connect":"insert",
  "value":"C#",
  "lang":"/lang/en"
}
```

5.5.1. Caution: unless_exists with Expanded Objects

From the explanation above, you might assume that the compact creation query with which we began this section could be equivalently (but less compactly written) as:

```
{
  "create":"unless_exists",
  "id":null,
  "name": "C#",
  "type": {
    "connect":"insert",
    "id":"/user/docs/music/note"
  }
}
```

If the queries used `"create":"unconditional"` then they would be the same. But the meaning of `unless_exists` is different for the two queries. The original compact query could be translated as *If you can find a Note object named "C#", return its id. Otherwise, create a new Note object, name it "C#", and return its id.*

But this variant that expands the `type` property is different in a subtle but important way. It tells Metaweb: *find or create an object named "C#", and then add Note to its set of types*. The difference between the two queries is critical if there is already an object (of type `/programming/language`, perhaps) with the name "C#".

Here's another way to think about this. When the type is specified by id, this is a constraint on the query. Metaweb must find an object that matches, or must construct one. When the type is specified in a sub-query with an explicit `connect` directive the sub-query is not a constraint, and does not affect the results of the `unless_exists` search.

5.6. Creating and Connecting More Objects

In order to try some more advanced MQL write queries, we need to add a property to our Note type. Like types themselves, properties are just Metaweb objects, and can be created with MQL writes. It is tricky to do this in practice, however, and the *freebase.com* client makes it easy to add properties. Start by viewing the *schema* page for your Note type. You can navigate to this page using links in the client, or enter its URL directly into your browser:

```
http://sandbox.freebase.com/type/schema/user/docs/music/note
```

On this schema page, you'll find a user interface for adding new properties. Create a property with name and key set to `next` and with its expected type set to the Note type. Also, make the property unique, so that it is restricted to one value. This newly created property links one Note object to another, and we'll use it to link notes to their perfect fifth – the note that is 7 semitones higher (usually, this is 5 white keys on a piano keyboard, which is probably why it is called a fifth.) If we start with the note C, we find that it's fifth is the note G. Let's create Note objects to represent the notes C and G. Note that the following query is two independent queries in an array:

Write	Result
`[{`	`[{`
`  "create":"unless_exists",`	`  "create":"created",`
`  "id":null,`	`  "id":"/guid/1f80000000000384b0",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"C"`	`  "name":"C"`
`},{`	`},{`
`  "create":"unless_exists",`	`  "create":"created",`
`  "id":null,`	`  "id":"/guid/1f80000000000384b4",`
`  "type":"/user/docs/music/note",`	`  "type":"/user/docs/music/note",`
`  "name":"G"`	`  "name":"G"`
`}]`	`}]`

We've asked Metaweb to create two Note objects, with names C and G, and to return their ids to us. Now, let's insert the link that indicates that G is the fifth of C:

```
Write                                    Result
{                                        {
  "id":"/guid/1f80000000000384b0",         "id":"/guid/1f80000000000384b0",
  "/user/docs/music/note/next":{           "/user/docs/music/note/next":{
    "connect":"update",                      "connect":"inserted",
    "id":"/guid/1f80000000000384b4"          "id":"/guid/1f80000000000384b4"
  }                                        }
}                                        }
```

This compact query identifies both note objects by id and connects them with a `connect` directive. Since we defined the `next` property to be unique, it uses `"connect":"update"` instead of `"connect":"insert"`. Note that since this query never specifies the type of the objects, we must use a fully-qualified property name for the `next` property. You can verify that this query did what we intended using the *freebase.com* client. Visit My Freebase on *sandbox.freebase.com*, and click on the Note type. On the page for the Note type, you should see a list of instances of that type. Click on the one named "C", and you'll see that it includes a hyperlink labeled "Next" to the note G.

The linking technique shown above is straightforward and easy to understand. It uses one query to create (or look up) the two objects to be linked. Then it uses a second simple query to connect the two objects. It is usually possible, however, to combine the creation and linking into a single query. The following query, for example, sets the `next` property of the note G to a newly-created note named D:

```
Write                                    Result
{                                        {
  "type":"/user/docs/music/note",          "type":"/user/docs/music/note",
  "name":"G",                              "name":"G",
  "next":{                                 "next":{
    "create":"unless_exists",                "create":"created",
    "type":"/user/docs/music/note",          "type":"/user/docs/music/note",
    "name":"D"                               "name":"D"
  }                                        }
}                                        }
```

Notice that there is no `connect` directive here. Since the `create` directive is nested in this query, the connection is implicit.

Here's a longer query of the same sort:

Write

```
{
  "create":"unless_exists",
  "type":"/user/docs/music/note",
  "name":"B flat",
  "next":{
    "create":"unless_exists",
    "type":"/user/docs/music/note",
    "name":"F",
    "next":{
      "create":"unless_exists",
      "type":"/user/docs/music/note",
      "name":"C"
    }
  }
}
```

Result

```
{
  "create":"created",
  "type":"/user/docs/music/note",
  "name":"B flat",
  "next":{
    "create":"created",
    "type":"/user/docs/music/note",
    "name":"F",
    "next":{
      "create":"connected",
      "type":"/user/docs/music/note",
      "name":"C"
    }
  }
}
```

This query creates a note F and links it to the existing note C, and then creates a note B flat and links it to the new note F. Note that the query uses `"create":"unless_exists"` three times. The response includes "created" twice for the newly created notes. But for the note C, which already exists, the response says `"create":"connected"`. This tells us that the note C already existed, but that a new connection has been made to it. If we rerun the query, we get `"create":"existed"` all three times, since the objects and links already exist.

The following query is like the one above, but shorter, and with one important tweak:

Write

```
{
  "create":"unless_exists",
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "create":"unless_connected",
    "type":"/user/docs/music/note",
    "name":"B flat"
  }
}
```

Result

```
{
  "create":"created",
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "create":"created",
    "type":"/user/docs/music/note",
    "name":"B flat"
  }
}
```

This query creates a new note E flat, and connects it to B flat. Notice, however, that in the nested clause of the query, we used a different form of the `create` directive: `"create":"unless_connected"`. And in the response we have a `"create":"created"`. If you examine the list of Note instances in the *freebase.com* client, you'll see that there are now two of them named "B flat". If you use `unless_connected`, then Metaweb looks for a matching object that is already connected. If it cannot find one, it creates a new one and connects it. In this case, there was an existing Note object named B flat, but it was not already connected, so the query created a new one. If we rerun the query, however, it simply returns `"create":"existed"` because the object and the connection exist.

Note that `unless_connected` only makes sense in nested clauses. If we change the outermost `unless_exists` in the query above to `unless_connected`, Metaweb complains: *Can't use 'create': 'unless_connected' at the root of the query*.

When to use unless_connected

`"create":"unless_connected"` directive is relatively infrequently used. Use it when objects must be unique within their "parent". One example of this is in the `/film` domain, where a `/film/performance` (an actor/character pair) can be linked to only one `/film/film`. Johnny Depp plays the character Jack Sparrow in more than one film, and there is a separate `/film/performance` object to represent each of those performances. If you were crafting a MQL query to create an object representing the film *Pirates of the Caribbean V: Jumping the Shark*, therefore, you would use `unless_connected` to link Depp's performance to the film. If you mistakenly used `unless_exists`, Metaweb would try (and fail because of the uniqueness requirement) to create a link to an existing Depp/Sparrow performance.

Let's clean up the extra B flat object we created:

Write

```
{
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "connect":"delete",
    "type":{
      "connect":"delete",
      "id":"/user/docs/music/note"
    },
    "name":{
      "connect":"delete",
      "value":"B flat",
      "lang":"/lang/en"
     }
  }
}
```

Result

```
{
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "connect":"deleted",
    "type":{
      "connect":"deleted",
      "id":"/user/docs/music/note"
    },
    "name":{
      "connect":"deleted",
      "value":"B flat",
      "lang":"/lang/en"
    }
  }
}
```

Note that the query above does two things. It disconnects the name and type of the extra B flat object, and also disconnects that object from E flat. Now all we have to do is connect E flat to the valid B flat object. This should be easy for you now:

Write

```
{
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "connect":"insert",
    "type":"/user/docs/music/note",
    "name":"B flat"
  }
}
```

Result

```
{
  "type":"/user/docs/music/note",
  "name":"E flat",
  "next":{
    "connect":"inserted",
    "type":"/user/docs/music/note",
    "name":"B flat"
  }
}
```

5.7. Review: Write Directives

At this stage of the tutorial, you've seen all the variations of the `create` and `connect` directives. Let's do a quick review before diving in to some more advanced examples.

The `create` directive comes in three forms:

`"create":"unless_exists"`	Look for the object in the database and create a new one if a match cannot be found.
`"create":"unless_connected"`	Look for a matching object that already exists and is already connected to the parent query. If no such object exists, create and connect a new one.
`"create":"unconditional"`	Always create the specified object. It is almost never necessary or appropriate to use this form of the `create` directive.

The possible responses to a `create` directive are the following:

`"create":"created"`	Indicates that a new object has been created. This is always the response for `unconditional` directives, but may also be returned by `unless_exists` and `unless_connected` directives.
`"create":"existed"`	Indicates that a pre-existing match was found and no object was created. This may be returned by `unless_exists` or `unless_connected` directives.
`"create":"connected"`	Indicates that the object already existed but a connection has been made. This response is only possible for `unless_exists` directives that are nested within a parent query.

The four forms of the `connect` directive are:

`"connect":"insert"`	Use this form to attach a value or object to a non-unique property. It can also be used to attach the first value or object to a unique property.

`"connect":"update"`	Use this form to attach a value or object to a unique property, replacing any value or object that was previously connected.
`"connect":"replace"`	This form does an update if the property is unique and does an insert otherwise. It is rarely necessary to use this type of connect.
`"connect":"delete"`	Use this form to detach a value or object from a property. It works for unique and non-unique properties.

There are five possible responses to a `connect` query:

`"connect":"inserted"`	Indicates that an `insert` directive was successful.
`"connect":"updated"`	Indicates that an `update` directive was successful.
`"connect":"deleted"`	Indicates that a `delete` directive was successful.
`"connect":"present"`	Indicates that an `insert` or `update` directive was unsuccessful because the specified connection was already present.
`"connect":"absent"`	Indicates that a `delete` directive was not successful because the connection to be deleted did not exist.

5.8. Working with Sets

The most interesting examples we've explored so far have used the `next` property of our Note type. We defined this property to be unique – so that it can have only one value. There are some features of the MQL write grammar that only become apparent when used on non-unique properties, however. Let's define a Chord type and give it a non-unique property named `note` which links to Note objects. (By convention, we use a singular property name, even though we expect each Chord object to refer to multiple Note objects.) Create this type and its property on *`sandbox.freebase.com`* by repeating the steps you followed to define the Note type and its `next` property. Just change the names to Chord and `note`, and don't check the "Restrict to one value" box. The examples that follow assume that the id of the new Chord type is `/user/docs/music/chord`. You'll need to substitute your own username into the queries as you follow along.

Once the new type is created, let's define a chord using the notes C, E, and G:

Write | **Result**

```
{                                   {
  "create":"unless_exists",           "create":"created",
  "name":"CEG",                       "name":"CEG",
  "type":[                            "type":[
    "/common/topic",                    "/common/topic",
    "/user/docs/music/chord"            "/user/docs/music/chord"
  ],                                  ],
  "note":[{                           "note":[{
    "create":"unless_exists",           "create":"connected",
    "type":"/user/docs/music/note",     "type":"/user/docs/music/note",
    "name":"C"                          "name":"C"
  },{                                 },{
    "create":"unless_exists",           "create":"connected",
    "type":"/user/docs/music/note",     "type":"/user/docs/music/note",
    "name":"G"                          "name":"G"
  },{                                 },{
    "create":"unless_exists",           "create":"created",
    "type":"/user/docs/music/note",     "type":"/user/docs/music/note",
    "name":"E"                          "name":"E"
  }]                                  }]
}                                   }
```

Several things immediately stand out about this query:

- It specifies the ids of two types within a JSON array. The created object will be both a Chord and a Topic. (We'll say more about arrays in write queries below).
- It specifies three notes, as expanded objects, within a JSON array. These are the set of values for the `note` property of the chord.
- Note objects C and G exist already, so the response to the `"create":"unless_exists"` directive for these two notes is `"create":"connected"` to indicate that an already-existing object was connected. (Since we knew ahead of time that these two notes already existed, we could have used `"connect":"insert"` instead.) Since note E did not exist, the response includes `"connect":"created"` indicating that the note object was created and connected.

So far in this chapter, we've only seen square brackets in write queries when we were bundling up multiple top-level queries to be submitted to Metaweb in a single batch. The MQL write grammar is actually more general than this: nested queries can also be collected into an array, and this allows us to connect more than one value to a property. In the case of the `type` property, our query specifies two types by their `id`. As we discussed earlier, types can be specified by id because `id` is the default property of `/type/type`. When types are specified this way, `"connect":"insert"` is assumed.

Multiple Types and Unqualified Property Names

When we specify more than one type for an object, we use a JSON array. But the Metaweb object model represents the types as an unordered set, so the order in which we specify them should not matter. In fact, however, it does. The last type in the array of types is used to qualify any unqualified property names that are not `/type/object` properties.

In the query above, if we had specified `/user/docs/music/chord` first, and `/common/topic` second, then Metaweb would have assumed that the unqualified `note` property meant `/common/topic/note`, and this would have caused an error since there is no such property. If you don't want to rely on the order of the types, you can just be explicit and use the fully-qualified names of all properties, such as `/user/docs/music/chord/note`.

5.9. Reciprocal Properties

One of the fundamental aspects of Metaweb is that all links between nodes are bi-directional. Our CEG Chord node has links to the nodes that represent the notes C, E, and G. Those links are bi-directional, which means that the C, E, and G nodes are linked to the CEG Chord node. The links are there, but our Note type doesn't define a appropriate property that exposes those links in the object-oriented view of the database.

The Freebase client makes it very easy to define such a property. View the schema of your note type by entering a URL like this:

```
http://sandbox.freebase.com/type/schema/user/docs/music/note
```

In addition to listing the properties defined by the Note type, this page also lists the "incoming properties" that have Note as their expected type. This list of incoming properties includes an option to create a reciprocal or "return property". A good name for the reciprocal of the `chord/note` property is, of course, `note/chord`. Since you now have a pair of properties, you can take advantage of the bi-directional nature of the links between chords and notes.

Let's experiment with this. First, we'll query the Chord CEG to find out what notes it contains:

Read	Result

```
{
  "type":"/user/docs/music/chord",
  "name":"CEG",
  "note":[]
}
```
	```
	{
	  "type":"/user/docs/music/chord",
	  "name":"CEG",
	  "note":["C","G","E"]
	}
	```

This result is unsurprising, given that the `/user/docs/music/chord/note` property is the one we defined originally. Now let's turn the query around and try out the reciprocal `/user/docs/music/note/chord` property we've just added. What chords is the note C a part of?

```
Read                              Result
{                                 {
  "type":"/user/docs/music/note",   "type":"/user/docs/music/note",
  "name":"C",                       "name":"C",
  "chord":[]                        "chord":["CEG"]
}                                 }
```

The note C "knows" that it is part of the chord CEG even though we never set its `chord` property. Setting a property automatically causes its reciprocal property to be set as well. Because links are bi-directional in Metaweb, this is all automatic.

Now let's create a new chord:

```
Write                                Result
{                                    {
  "create":"unless_exists",            "create":"created",
  "type":["/common/topic",             "type":["/common/topic",
          "/user/docs/music/chord"],           "/user/docs/music/chord"],
  "name":"BFG"                         "name":"BFG"
}                                    }
```

We've created a chord named BFG, but we haven't added the notes B, F and G to it. To further demonstrate reciprocal properties, we'll do the reverse, and add the chord to the notes:

```
Write                                Result
[{                                   [{
  "create":"unless_exists",            "create":"created",
  "type":"/user/docs/music/note",      "type":"/user/docs/music/note",
  "name":"B",                          "name":"B",
  "chord": {                           "chord":{
    "connect":"insert",                  "connect":"inserted",
    "type":"/user/docs/music/chord",     "type":"/user/docs/music/chord",
    "name":"BFG"                         "name":"BFG"
  }                                    }
},{                                  },{
  "create":"unless_exists",            "create":"existed",
  "type":"/user/docs/music/note",      "type":"/user/docs/music/note",
  "name":"F",                          "name":"F",
  "chord": {                           "chord":{
    "connect":"insert",                  "connect":"inserted",
    "type":"/user/docs/music/chord",     "type":"/user/docs/music/chord",
    "name":"BFG"                         "name":"BFG"
  }                                    }
},{                                  },{
  "create":"unless_exists",            "create":"existed",
  "type":"/user/docs/music/note",      "type":"/user/docs/music/note",
  "name":"G",                          "name":"G",
  "chord": {                           "chord":{
```

Write	Result
`    "connect":"insert",`	`    "connect":"inserted",`
`    "type":"/user/docs/music/chord",`	`    "type":"/user/docs/music/chord",`
`    "name":"BFG"`	`    "name":"BFG"`
`  }`	`  }`
`}]`	`}]`

This query connects the BFG chord to the `chord` property of the notes B, F, and G. (It also creates the note B, which didn't exist yet.) Now let's ask BFG what notes it contains:

Read	Result
`{`	`{`
`  "type":"/user/docs/music/chord",`	`  "type":"/user/docs/music/chord",`
`  "name":"BFG",`	`  "name":"BFG",`
`  "note":[]`	`  "note":["B","F","G"]`
`}`	`}`

Once again, we've demonstrated that we can set a property of an object by setting the reciprocal property to refer to that object.

We began this section by creating the `/user/docs/music/note/chord` property as the explicit reciprocal of `/user/docs/music/chord/note`. This step is not actually necessary, however. Metaweb can traverse a link in the reverse direction even if a property describing that direction does not exist. MQL also allows us to refer to the reciprocal of a property by prefixing the property id with an exclamation mark. So in the queries above, we could replace the property `chord` with `!/user/docs/music/chord/note`. See §3.4.4 for further discussion.

5.10. Writes and Ordered Collections

If a Metaweb property has not been declared a unique property, it may have a set of values. As we saw in Chapter 3, these sets may be *ordered*, and MQL read queries can access this order with the `index` directive. This section shows how to define an ordering with a MQL write query. Not surprisingly, this is also uses the `index` directive.

To demonstrate, we'll use our Chord type to represent arpeggios. An *arpeggio* (or "broken chord") is a set of notes played sequentially rather than simultaneously. Since there is a sequence, there is an order, and we'll use the `index` directive to specify the order in which the notes should be played. Here's how we might create a chord with ordered notes:

Write

```
{
  "create":"unless_exists",
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note": [{
    "index":0,
    "type":"/user/docs/music/note",
    "name":"C"
  },{
    "index":1,
    "type":"/user/docs/music/note",
    "name":"E"
  },{
    "index":2,
    "type":"/user/docs/music/note",
    "name":"G"
  }]
}
```

Result

```
{
  "create":"created",
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note":[{
    "index":0,
    "type":"/user/docs/music/note",
    "name":"C"
  },{
    "index":1,
    "type":"/user/docs/music/note",
    "name":"E"
  },{
    "index":2,
    "type":"/user/docs/music/note",
    "name":"G"
  }]
}
```

Two things stand out about this query: each note has an `index` associated with it, and there are no `connect` directives. The `index` directive specifies the ordering. Remember that this ordering is not a property of the Chord, nor of the Note objects that comprise it. Instead, the indexes are properties of the links between the chord and the notes. It is not surprising, then, that using the `index` directive in a write query implicitly specifies `"connect":"insert"` for that query. You can use the `index` directive even for object that have already been inserted: in this case, the `index` directive simply re-orders the object without attempting to re-insert it. If we were creating the Note objects at the same time as we were inserting them into this Chord, we would have to include both the `create` directive and the `index` directive.

There are some strict rules that govern the use of the `index` directive in write queries:

- The `index` directive may not appear within a top-level query. Indexes don't apply to objects but to the links between objects. The `index` directive is used in sub-queries to specify the order of the links between the parent object and the children.

- If there are *n* sibling sub-queries that specify an index, the values specified must include every integer from 0 to *n*-1. You must always start with zero. You may not include duplicate indexes, and you may not skip an index. It is not required that every element of a sub-query array have an index. Metaweb collections can be partially ordered and partially unordered.

This second rule may seem surprisingly strict, but remember that despite the name "index", the values we specify with the `index` directive are not array indexes. The numbers are merely a simple way to specify a series of less than and greater than relationships. The requirement that indexes always run from 0 through *n*-1 means that there is no way to insert an element at a given location with an ordered collection. All we can do is move elements to (or insert new elements at) the beginning of the list. Here's how we would change our CEG arpeggio into a GCE arpeggio, for example:

Write

```
{
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note": [{
    "index":0,
    "type":"/user/docs/music/note",
    "name":"G"
  }]
}
```

Result

```
{
  "type" : "/user/docs/music/chord",
  "name" : "broken CEG",
  "note" : [{
      "index" : 0,
      "type" : "/user/docs/music/note"
      "name" : "G",
   }]
}
```

In this query, the response is identical to the query: there is no `"index":"reordered"` property in the response to let us know that our query succeeded. But we can check with a simple read:

Read

```
{
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note":[{
    "index":null,
    "name":null,
    "sort":"index"
  }]
}
```

Result

```
{
  "type" : "/user/docs/music/chord",
  "name" : "broken CEG",
  "note" : [
    {"index" : 0, "name" : "G"},
    {"index" : 1, "name" : "C"},
    {"index" : 2, "name" : "E"}
  ]
}
```

Now, let's add two more notes to beginning of the arpeggio. This query demonstrates that the `index` property can be used along with a `create` directive. The notes already exist, but are not connected so we get `"create":"connected"` in the response:

Write

```
{
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note": [{
    "create":"unless_exists",
    "index":0,
    "type":"/user/docs/music/note",
    "name":"B"
  },{
    "create":"unless_exists",
    "index":1,
    "type":"/user/docs/music/note",
    "name":"F"
  }]
}
```

Result

```
{
  "type":"/user/docs/music/chord",
  "name":"broken CEG",
  "note":[{
    "create":"connected",
    "index":0,
    "type":"/user/docs/music/note",
    "name":"B"
  },{
    "create":"connected",
    "index":1,
    "type":"/user/docs/music/note",
    "name":"F"
  }]
}
```

If you repeat the read query from above, you'll see that the sequence of notes in our arpeggio is now BFGCE.

Metaweb's ordered collections are not random-access arrays and do not behave that way. In read queries, you cannot ask for the object with a specific index, you can only sort by index (and optionally limit the number of results.) And in writes, you cannot insert an element at a specified index unless you specify the index of all elements that come before it.

Arpeggios and Duplicate Notes

If you are a musician, you probably know that broken chords often repeat a note. In practice, we'd want to represent arpeggios like EGCE, where the note E appears twice. In Metaweb, the value of a property is a set, and sets do not allow duplicates, even when they are ordered. That is, ordered collections in Metaweb are still sets, not lists, and they do not allow duplicates. In order to represent an arpeggio EGCE, therefore, we'd have to create two separate note objects named E. But having two objects that both represent the note E is problematic: the `next` and `chord` properties of the Note type are premised on the assumption that there will only be one Note instance for each note.

There are two lessons to be learned here:

- Ordered collections are still sets, and do not allow duplicates.
- Designing good Metaweb schemas for knowledge representation is hard to do.

5.11. Namespaces and Enumerations

By placing an object in a namespace we define a fully-qualified name for it, and that name can be used as the value of the `id` property to uniquely identify the object. In this section we'll demonstrate how to do this and explore namespaces and enumerations in more detail.

We begin with a review of material from Chapter 2. First, remember that fully-qualified names and namespaces don't have anything to do with the `name` property of an object. The `name` property defines a human-readable display name for an object.

Fully-qualified names are defined by the value type `/type/key`. Every object has a `key` property that holds a set of `/type/key` values. If you want an object to have a fully-qualified name, insert a key into its `key` property. The `value` property of the key specifies the object's unqualified or local name. And the `namespace` property of the key specifies the object that defines the namespace. Any object can be a namespace: the only requirement is that the object must itself have a key. In this way we get a chain of `/type/key/value` properties that continues until we find a `/type/key/namespace` property that refers to the special root namespace object.

The type `/type/namespace` defines the property `/type/namespace/keys`, which is the reciprocal of `/type/key/namespace`. Namespaces also have a `unique` property. If `true`, the namespace may not contain two names for the same object. Objects that are used as namespaces are usually given the type `/type/namespace`, but this is not required.

The reason that namespaces are useful is that namespaces allow us to use fully-qualified names to uniquely identify objects. If an object is given a key, then we can use its unique fully-qualified name as the value of the `id` property. Identifying objects with a meaningful id is simpler than using a long string of hexadecimal digits in the `/guid` pseudo-namespace.

Now that we've reviewed namespaces, let's create a namespace in which we can define names for our Note objects. Since notes are of type `/user/docs/music/note`, let's use the plural form `/user/docs/music/notes` as the id of the namespace. Here's how we create the new namespace object and insert it into `/user/docs/music` (using our domain object as a namespace):

Write | **Result**

```
{                                       {
  "id":"/user/docs/music",                "id" : "/user/docs/music",
  "/type/namespace/keys": {               "/type/namespace/keys" : {
    "value":"notes",                        "value" : "notes",
    "namespace": {                          "namespace" : {
      "create":"unless_connected",            "create" : "created",
      "type":"/type/namespace",               "type" : "/type/namespace",
      "unique":false                          "unique" : false
    }                                       }
  }                                       }
}                                       }
```

Now let's put some of the note objects we've created into our new namespace:

Write | **Result**

```
[{                                            [{
  "type":"/user/docs/music/note",               "type":"/user/docs/music/note",
  "name":"C",                                   "name":"C",
  "key":{                                       "key":{
    "connect":"insert",                           "connect":"inserted",
    "namespace":"/user/docs/music/notes",         "namespace":"/user/docs/music/notes",
    "value":"C"                                   "value":"C"
  }                                             }
},{                                           },{
  "type":"/user/docs/music/note",               "type":"/user/docs/music/note",
  "name":"E",                                   "name":"E",
  "key":{                                       "key":{
    "connect":"insert",                           "connect":"inserted",
    "namespace":"/user/docs/music/notes",         "namespace":"/user/docs/music/notes",
    "value":"E"                                   "value":"E"
  }                                             }
},{                                           },{
  "type":"/user/docs/music/note",               "type":"/user/docs/music/note",
  "name":"G",                                   "name":"G",
  "key":{                                       "key":{
    "connect":"insert",                           "connect":"inserted",
    "namespace":"/user/docs/music/notes",         "namespace":"/user/docs/music/notes",
    "value":"G"                                   "value":"G"
Write                                        Result
  }                                            }
}]                                           }]
```

This query gives the notes C, E, and G keys named "C", "E", and "G" within the namespace `/user/docs/music/notes`. That is, it defines fully-qualified names for these notes `/user/docs/music/notes/C`, `/user/docs/music/notes/E`, and `/user/docs/music/notes/G`. Now that these notes have unique ids, it becomes (somewhat) easier to use them in queries. Here's how we might create a chord:

```
Write                                   Result
{                                       {
  "create":"unless_exists",               "create":"existed",
  "type":"/user/docs/music/chord",        "type":"/user/docs/music/chord",
  "name":"CEG",                           "name":"CEG",
  "note":[{                               "note":[{
    "connect":"insert",                     "connect":"present",
    "id":"/user/docs/music/notes/C"         "id":"/user/docs/music/notes/C"
  },{                                     },{
    "connect":"insert",                     "connect":"present",
    "id":"/user/docs/music/notes/E"         "id":"/user/docs/music/notes/E"
  },{                                     },{
    "connect":"insert",                     "connect":"present",
    "id":"/user/docs/music/notes/G"         "id":"/user/docs/music/notes/G"
  }]                                      }]
}                                       }
```

This query replaces the `name` and `type` properties of each note with a single `id` property. It doesn't actually do anything, since we have already created the CEG chord. We've seen that we can use a note's fully-qualified name as the value of its `id` property. What if we query the `id` of a note?

```
Read                                  Result
{                                     {
  "type":"/user/docs/music/chord",      "type" : "/user/docs/music/chord",
  "name":"CEG",                         "name" : "CEG",
  "note":[{"id":null}]                  "note" : [
}                                         {"id" : "/user/docs/music/notes/C"},
                                          {"id" : "/user/docs/music/notes/G"},
                                          {"id" : "/user/docs/music/notes/E"}
                                        ]
                                      }
```

Our query now returns the ids we just defined rather returning an id from the `/guid` namespace as it would have done before these ids were defined.

5.11.1. The /type/namespace/keys Property

We've seen that we can put objects into a namespace by setting the `key` property of the object. It is also possible to work with namespaces using the reciprocal property `/type/namespace/keys`. We've been using `/user/docs/music/note` as a namespace. This next query asks what keys it holds:

```
Read                                Result
{                                   {
  "id":"/user/docs/music/notes",      "id":"/user/docs/music/notes",
  "/type/namespace/keys":[]           "/type/namespace/keys":[
}                                       "C","E","G"
                                      ]
                                    }
```

The namespace holds the local names of the three notes we added. Let's repeat the query and ask for more detail:

```
Read                                Result
{                                   {
  "id":"/user/docs/music/notes",      "id" : "/user/docs/music/note",
  "/type/namespace/keys":[{}]         "/type/namespace/keys" : [{
}                                       "type" : "/type/key",
                                        "namespace" : "/user/docs/music/notes/C",
                                        "value" : "C"
                                      },{
                                        "type" : "/type/key",
                                        "namespace" : "/user/docs/music/notes/E",
                                        "value" : "E"
                                      },{
                                        "type" : "/type/key",
                                        "namespace" : "/user/docs/music/notes/G",
                                        "value" : "G"
                                      }]
                                    }
```

The values of the `/type/namespace/keys` property are `/type/key` values that have `value` and `namespace` properties.

There is one very important point to notice about these query results. When a key value is used with `/type/object/key`, the `namespace` property is the id of the namespace object (such as `/user/docs/music/notes`) that holds the key. But when a key value is used with `/type/namespace/keys`, the `namespace` property is the id of the object (such as `/user/docs/music/notes/C`) contained by the namespace. This is important to understand, so we'll state it another way: suppose that an object `o` has a fully-qualified name in the namespace `n`. If we query the `key` property of `o`, we'll find a `/type/key` object whose `namespace` property refers to `n`. And if we query the `/type/namespace/keys` property of `n`, we'll find a `/type/key` object whose `namespace` property refers to `o`.

If you wanted to create a Metaweb namespace browser application, you could repeat the query above, starting with the id of the root namespace `"/"`. The `namespace` properties of each of the returned keys specify the ids of all objects in the root namespace. If you recursively query each of these ids, you'll find the complete set of Metaweb objects with fully-qualified names.

It is also possible to add objects to namespaces using the `/type/namespace/keys` property instead of `/type/object/key`. The following query creates a new Note object named "G flat" and assigns it the fully-qualified name `/user/docs/music/notes/G_flat`:

Write

```
{
  "id":"/user/docs/music/notes",
  "/type/namespace/keys":{
    "connect":"insert",
    "value":"G_flat",
    "namespace":{
      "create":"unless_exists",
      "name":"G flat",
      "type":"/user/docs/music/note"
    }
  }
}
```

Result

```
{
  "id":"/user/docs/music/notes",
  "/type/namespace/keys":{
    "connect":"inserted",
    "value":"G_flat",
    "namespace":{
      "create":"created",
      "name":"G flat",
      "type":"/user/docs/music/note"
    }
  }
}
```

5.11.2. Fully-Qualified Names and Uniqueness

The key feature of fully-qualified names is their uniqueness: two objects simply cannot share the same id. In this section we experiment with uniqueness and demonstrate how to change the object to which an id refers and how to change the id of an object.

First, let's try to give the note F the same key that we assigned to G:

```
{
  "type":"/user/docs/music/note",
  "name":"F",
  "key":{
    "connect":"insert",
    "namespace":"/user/docs/music/notes",
    "value":"G"
  }
}
```

This query is syntactically valid JSON, and semantically valid MQL, but it fails with the error message "This value is already in use. Please delete it first": Metaweb simply will not allow the fully-qualified name `/user/docs/music/notes/G` to refer to two different note objects. If you want to make `/user/docs/music/notes/G` refer to the note F, you must first make sure that that note does not refer to the note G. This takes two queries. First, we must remove the fully-qualified name for the note G:

Write

```
{
  "id":"/user/docs/music/notes/G",
  "key":{
    "connect":"delete",
    "namespace":"/user/docs/music/notes",
    "value":"G"
  }
}
```

Result

```
{
  "id":"/user/docs/music/notes/G",
  "key":{
    "connect":"deleted",
    "namespace":"/user/docs/music/notes",
    "value":"G"
  }
}
```

And then we can assign that fully-qualified name to the note F:

Write

```
{
  "type":"/user/docs/music/note",
  "name":"F",
  "key":{
    "connect":"insert",
    "namespace":"/user/docs/music/notes",
    "value":"G"
  }
}
```

Result

```
{
  "type":"/user/docs/music/note",
  "name":"F",
  "key":{
    "connect":"inserted",
    "namespace":"/user/docs/music/notes",
    "value":"G"
  }
}
```

Now if we were to ask for the name of the note `/user/docs/music/notes/G`, we'd get "F". Making a fully-qualified name refer to another object is simpler if we use the `/type/namespace/keys` property instead. Here's how we could make `/user/docs/music/note/G` refer to the note G again. Note that only one query is required if we do it this way:

Write

```
{
  "id":"/user/docs/music/notes",
  "/type/namespace/keys": {
    "value":"G",
    "namespace":{
       "connect":"update",
       "type":"/user/docs/music/note",
       "name":"G"
    }
  }
}
```

Result

```
{
  "id":"/user/docs/music/notes",
  "/type/namespace/keys":{
    "value":"G",
    "namespace":{
      "connect":"updated",
      "type":"/user/docs/music/note",
      "name":"G"
    }
  }
}
```

This query locates the `/type/key` object that defines the name `/user/docs/music/notes/G`, and updates the `namespace` property of that key, so that the name points to a different object. Note that you should not typically have to alter namespaces like this. Objects that have fully-qualified names should typically be constants.

Finally, notice that changing the object to which a fully-qualified name refers (as we did above) is a completely different operation than changing the fully-qualified name of an object. If we wanted to refer to the note G by the name /user/docs/music/note/Gnatural instead of /user/docs/music/note/G, we could do this:

```
Write                                      Result
{                                          {
  "name":"G",                                "name":"G",
  "type":"/user/docs/music/note",            "type":"/user/docs/music/note",
  "key":[{                                   "key":[{
    "connect":"delete",                        "connect":"deleted",
    "namespace":"/user/docs/music/notes",      "namespace":"/user/docs/music/notes",
    "value":"G"                                "value":"G"
  },{                                        },{
    "connect":"insert",                        "connect":"inserted",
    "namespace":"/user/docs/music/notes",      "namespace":"/user/docs/music/notes",
    "value":"Gnatural"                         "value":"Gnatural"
  }]                                         }]
}                                          }
```

Since we did not set the unique property of our /user/docs/music/notes namespace to true, it is perfectly legal for one note to have two names (such as "G" and "Gnatural") in the namespace.

5.11.3. Enumerations

Suppose we want *all* of our Note objects to have a fully-qualified name in the /user/docs/music/notes namespace. We can simplify the process of defining these fully-qualified names by giving the Note type a property of /type/enumeration. (See §3.3.5 for a review of enumerations).

Begin by adding a new property to the Note type using the freebase.com client on the sandbox. Name the property "lname" (for "local name") and set its expected type to /type/enumeration. Since our /user/docs/music/notes namespace is not unique (it allows multiple keys to refer to the same object) don't make this new lname property be unique.

At the time of this writing, the Freebase client does not allow you to specify the namespace associated with this lname property, and we have to make our own write query to specify that. When you try this yourself, you may find that you are able to enter the /user/docs/music/notes namespace directly into the client. If so that is all you need to do. If not, execute this query with the query editor on the sandbox:

```
Write                                      Result
{                                          {
  "id":"/user/docs/music/note/lname",        "id" : "/user/docs/music/note/lname",
  "/type/property/enumeration": {            "/type/property/enumeration" : {
    "connect":"update",                        "connect" : "inserted",
    "id:"/user/docs/music/notes"               "id" : "/user/docs/music/notes"
  }                                          }
}                                          }
```

With this new property defined and linked to our namespace, we can query the names, ids, and lnames of our notes. Note objects that have already been given fully-qualified names in our namespace automatically have their `lname` property defined. The following read query (and partial set of results) demonstrate:

Read

```
[{
    "type" : "/user/docs/music/note",
    "name" : null,
    "id" : null,
    "lname" : null
}]
```

Result

```
[{
  "type" : "/user/docs/music/note",
  "name" : "C",
  "id" : "/user/docs/music/notes/C",
  "lname" : "C"
},{
  "type" : "/user/docs/music/note",
  "name" : "G",
  "id" : "/user/docs/music/notes/Gnatural",
  "lname" : "Gnatural"
},{
  "type" : "/user/docs/music/note",
  "name" : "F",
  "id" : "/guid/1f800000000901ca07",
  "lname" : null
}]
```

With the `lname` enumeration defined, it becomes simple to create a new Note object and give it a fully-qualified name at the same time:

Write

```
{
  "create":"unless_exists",
  "type":"/user/docs/music/note",
  "name":"F sharp",
  "lname":"Fsharp",
  "id":null
}
```

Result

```
{
  "create" : "created",
  "type" : "/user/docs/music/note",
  "name" : "F sharp",
  "lname" : "Fsharp",
  "id" : "/user/docs/music/notes/Fsharp"
}
```

Specifying a value for `lname` is all we need to do to define a fully-qualified name for the new object. Note that the `id` query we included in the write returns the fully-qualified name we defined rather than the guid of the object.

To define a lname for an already existing object, you can use a query like this one, which creates a second fully-qualified name for the Note object we just defined:

```
Write                                       Result
{                                           {
  "id":"/user/docs/music/notes/Fsharp",       "id" : "/user/docs/music/notes/Fsharp",
  "type":"/user/docs/music/note",             "type" : "/user/docs/music/note",
  "lname":{                                   "lname" : {
    "connect":"insert",                         "connect" : "inserted",
    "value":"sharpf"                            "value" : "sharpf"
  }                                           },
}                                           }
```

You can check that this worked with a simple read query:

```
Read                                          Result
{                                             {
  "id" : "/user/docs/music/notes/sharpf",       "id" : "/user/docs/music/notes/sharpf",
  "/user/docs/music/note/lname" : [],           "/user/docs/music/note/lname" : [
}                                                 "Fsharp",
                                                  "sharpf"
                                                ]
                                              }
```

5.12. Access Control and Permissions

As you know, Metaweb databases are completely open for reading: no authentication is required, and no access control is performed for reads. That is not the case for writes, but so far in this chapter we've ignored access control issues. You've been creating instances of types you've defined yourself, and assuming that you've logged into the sandbox correctly, you've always been able to perform write queries. In practice, however, you may often need to work with domains, types and objects created by someone else, and you need to understand Metaweb's access control mechanism in order to know which writes are allowed and which are forbidden.

5.12.1. Users, Usergroups and Permissions

The `/type/user` type represents a single Metaweb user. The `/type/usergroup` type represents a set or group of users. And the `/type/permission` type represents a set of usergroups. A `/type/permission` object can be called a "permission group". Every object has a `permission` property that specifies its permission group. Only users in that permission group are allowed to modify the object. (This over-simplifies Metaweb's access control model a bit: details on per-object access control and per-property access control appear below.)

Your `/type/user` object has an associated usergroup and permission group that grants you permission to modify the object, and prevents most other users from modifying your object. You can look up the name and ids of your own usergroup and permission group with a read query like this one:

Read **Result**

```
{                    {
  "id":"/user/docs",   "id" : "/user/docs",
  "permission": {      "permission" : {
    "id":null,           "id" : "/guid/1f800000000120900f",
    "name":null,         "name" : null,
    "permits":[{         "permits" : [{
      "id":null,           "id" : "/boot/user_administrator_group",
      "name":null,         "name" : null,
      "member":[]          "member" : ["/user/user_administrator"]
    }]                   },{
  }                        "id" : "/guid/1f800000000120901b",
}                          "name" : "docs's private user group",
                           "member" : ["/user/docs"]
                         }]
                       }
                     }
```

Furthermore, when you create a new domain using the *freebase.com* client, a new permission group and two new user groups are created to control access to the domain. We can explore this by re-running the query above for the `/user/docs/music` domain object. The results shown here omit some of the administrative members of the `/boot/schema_group` usergroup:

Read **Result**

```
{                          {
  "id":"/user/docs/music",   "id" : "/user/docs/music",
  "permission": {            "permission" : {
    "id":null,                 "id" : "/guid/1f800000000903db10",
    "name":null,               "name" : null,
    "permits":[{               "permits" : [{
      "id":null,                 "id" : "/guid/1f800000000903db14",
      "name":null,               "name" : "Owners of music domain",
      "member":[]                "member" : ["/user/docs"]
    }]                         },{
  }                              "id" : "/guid/1f800000000903db1a",
}                                "name" : "Music Experts",
                                 "member" : []
                               },{
                                 "id" : "/boot/schema_group",
                                 "name" : null,
                                 "member" : [
                                   "/user/domain_administrator",
                                   "/user/typelibrarian",
                                   "/user/delete_bot",
                                   "/user/merge_bot"
                                 ]
                               }]
                             }
                           }
```

You might also be interested to know what usergroups you are part of and what permissions groups those usergroups are part of:

Read

```
{
  "id":"/user/docs",
  "/type/user/usergroup":[{
    "id":null,
    "name":null,
    "/type/usergroup/permitted":[{
      "id":null,
      "name":null
    }]
  }]
}
```

Result

```
{
  "id" : "/user/docs",
  "/type/user/usergroup" : [{
    "id" : "/boot/all_group",
    "name" : "Global User Group",
    "/type/usergroup/permitted" : [{
      "id" : "/boot/all_permission",
      "name" : "Global Write Permission"
    }]
  },{
    "id" : "/guid/1f800000000120901b",
    "name" : "docs's private user group",
    "/type/usergroup/permitted" : [{
      "id" : "/guid/1f800000000120900f",
      "name" : null
    },{
      "id" : "/guid/1f8000000001209021",
      "name" : null
    }]
  },{
    "id" : "/guid/1f8000000001209025",
    "name" : "Owners of docs's default types",
    "/type/usergroup/permitted" : [{
      "id" : "/guid/1f8000000001209021",
      "name" : null
    }]
  },{
    "id" : "/guid/1f800000000903db14",
    "name" : "Owners of music domain",
    "/type/usergroup/permitted" : [{
      "id" : "/guid/1f800000000903db10",
      "name" : null
    }]
  }]
}
```

All Metaweb users (in good standing) are part of the `/boot/all_group` usergroup and therefore part of the `/boot/all_permission` permission group. By default, new objects created with MQL write queries are given a `permission` of `/boot/all_permission`. We'll learn how to alter this default and specify a more restrictive permission in Chapter 6. MQL write queries are not allowed to alter the `permission` property of any object, so once an object is created, its permission is fixed. It is possible, of course, to alter the membership of a permission group or a usergroup, and the *freebase.com* client defines a UI for adding users to domain-related usergroups.

5.12.2. Per-Object Access Control

Every Metaweb object has a `permission` property that specifies the `/type/permission` object that controls access to it. The fundamental Metaweb access control rule is surprisingly simple: *to create an outgoing link from an object o you must be a member of the permission group of o.* For the purposes of this rule, setting a primitive value on o is considered creating an outgoing link.

Another way to say this is that creating a link (between two objects or between an object and a primitive value) requires membership in the permission group of the object from which the link originates. Remember that Metaweb properties can be master properties, which represent outgoing links (and primitive values), or reverse properties, which represent incoming links. So we can also state the access control rule in terms of properties:

- In order to set a master property of o to some other object p, the user must be a member of the permission group of o.
- In order to set a reverse property of o to some other object p, the user must be a member of the permission group of p.

The basic rule is very simple. But many important aspects of the Metaweb access control model flow from it. For example, `/type/object/type` is a master property (with `/type/type/instance` its reverse). This means that any user can link an object they create to any type, regardless of the permissions associated with the type. It is simply not possible to define a Metaweb type that other users cannot use.

Although the use of types is not subject to access control, the use of namespaces is tightly controlled. The `/type/object/key` property is a reverse property: `/type/namespace/keys` is the master property. This means that adding or modifying names in a namespace requires membership in the the permission group of the namespace. This means, for example, that users can't add types to other user's domains (unless the owner of that domain has added them to an appropriate usergroup), can't modify the system types in the `/type` domain, and can't define new languages in the `/lang` namespace.

5.12.3. Per-Property Access Control

Metaweb allows one additional layer of access control on top of the basic access control rule. It is possible to define properties that require special permission to set. Per-property access control is not based on the permission groups of the objects being linked, but on the permission group of the type that defines the property. This feature is important to owners of authoritative datasets who want to make that data available on a read-only basis through properties on existing objects.

At the time of this writing [2], the *freebase.com* client does not allow you to define properties of this kind. Instead, you must use a MQL write query to manually set the `/type/property/requires_permission` to `true` for the property. As an example, let's make the `next` property of our Note type require per-property permissions:

[2]September, 2008

Write

```
{
  "id":"/user/docs/music/note/next",
  "/type/property/requires_permission":{
    "connect":"update",
    "value":true
  }
}
```

Result

```
{
  "id" : "/user/docs/music/note/next",
  "/type/property/requires_permission" : {
    "connect" : "inserted",
    "value" : true
  }
}
```

With this change made, setting the `next` property requires membership in the permission group of the Note type (which, by default, is the same as the permission group of the `/user/docs/music` domain). Anyone can create Note instances, but only owners of the type can set the `next` property. Note that per-property access control is in addition to, not instead of, per-object access control. So to set the `next` property of a Note object, the user must be a member of the permission group of the object itself, and also be a member of the permission group of the Note type.

6

Metaweb Write Services

This chapter explains, and demonstrates with examples, how to deliver MQL write queries to Metaweb's *mqlwrite* service. As a necessary prerequisite, it shows how to log in to Metaweb to obtain authentication credentials. The chapter also demonstrates how to use the *upload* service to upload content, such as images and HTML documents to the Metaweb content store.

The examples in this chapter are written in Python. Some are extensions to the *metaweb.py* module of Example 4.15, adding support for the various write services. Other examples are command-line scripts that use *metaweb.py* to create Metaweb objects and upload content.

6.1. Logging in to Metaweb

No registration or authentication is required to use the *mqlread* service, but writing to Metaweb requires that you login first. Before we can cover *mqlwrite*, therefore, we must explain the *login* service.

Since Metaweb services are HTTP-based, Metaweb authentication is cookie-based. You log in to the sandbox server by making an HTTP request (either a GET or a POST) to the URL `ht-tp://sandbox.freebase.com/api/account/login`, passing your username and password as URL-encoded form parameters. If login is successful, Metaweb returns one or more HTTP cookies in the response headers. These cookies contain your authentication credentials, and you must pass these back to Metaweb in the HTTP request headers of all subsequent write and upload requests. To log into the main *freebase.com* server instead of the sandbox, use *api.freebase.com* instead of *sandbox.freebase.com*.

The names and values of the authentication cookies are an implementation detail rather than a specification detail, and are subject to change. To ensure success, your code should accept all cookies returned by the *login* service, and must present all of them to subsequent calls to the *mqlwrite* and *upload* services.

If you write your applications using a suitably high-level HTTP library (or run them in a browser), cookie handling may be performed automatically for you. In this case, you may want to add the parameter `rememberme=1` to the *login* request; doing this will cause Metaweb to return persistent cookies rather than session cookies. In the *metaweb.py* module shown in Example 4.15, the `Session` class maintains a "cookie jar" for storing cookies, and the `_http()` utility method adds cookies to each HTTP request and retrieves new cookies values from each HTTP response. We can therefore add support for write services to the module without doing any explicit cookie management.

The cookies returned by the login service have a long lifetime and it is possible for a script to login once, and then save the contents of its cookie jar to a file. On subsequent runs, it can load the cookie jar from the file rather than logging in again. The examples in this chapter, however, simply call the login service each time the script is executed.

Metaweb Credentials and Security

If you use the normal HTTP protocol, the password you supply to the *login* service, and the authentication cookies you pass to *mqlwrite* and *upload* are transferred across the network unencrypted and could be observed in transit. If you are concerned about this, use HTTPS instead of HTTP for the *login*, *write* and *upload* services. To protect your credentials you must also use HTTPS any time you log in to the *freebase.com* client. Note that the examples in this chapter do not use HTTPS.

Example 6.1 shows the Python code for a Metaweb `login()` method. This method is written to be part of the `metaweb.Session` class of Example 4.15, and it depends on constants and utility methods defined in that example. If login fails, the `login()` method raises a `metaweb.ServiceError` exception. Otherwise it stores authentication credentials in the cookie jar of the Session object and returns silently.

Example 6.1. metaweb.py: the login service

```
def login(self, username, password):
    """
    Submit the username and password to the Metaweb login service.

    This causes Metaweb to return an authentication cookie which will
    be passed back to the server with all subsequent requests.
    Returns nothing, but raises ServiceError on failure.
    """

    # This is the URL that we POST our login request to
    url = "http://%s%s" % (self.host, LOGIN)
    # This header specifies how the request body is encoded
    headers = {'Content-Type': 'application/x-www-form-urlencoded'}
    # This is the body of the request
    body = urllib.urlencode({'username':username, 'password':password})

    # POST the request body to the url.  Store authentication cookies
    # returned with the response. Parse the response to test for success
    # and raise an error if login failed.
    self._check(self._fetch(url, headers, body))
```

The fact that this `login()` method can use the same `_fetch()` and `_check()` utility methods that the read services of Chapter 4 do indicates that the Metaweb login service behaves very much like a read service. The HTTP response body is a JSON object that includes a `code` property. If the value of this property is "/api/status/ok", then the login was successful, and the Cookie header of the response contains the authentication credentials. Otherwise, the login failed, and the `messages` property contains an error message and other details.

6.2. Making Write Queries

Chapter 5 explained, in great detail, how to express write queries in MQL. Now we explain how to send those queries to Metaweb and retrieve the result. The Metaweb write service is *mqlwrite*. The path to this service is `/api/service/mqlwrite`, and using it is much like using *mqlread*. Follow these steps to perform a write:

1. Place your write query into a request envelope object, as the value of the `query` property.

2. Add any necessary envelope parameters to the envelope object.

3. Serialize the envelope object to a JSON string, and URL encode the string.

4. Make an HTTP POST request to `/api/service/mqlwrite` on the Metaweb server, configured as follows:

   - The body of the request should be the string `query=` followed by the URL-encoded and JSON serialized query envelope

   - The request must have a `Cookie` header that contains the authentication cookies returned by the login service.

   - For security reasons, the request must also include an `X-Metaweb-Request` header. The value doesn't matter; this header must simply be present.

5. When the response arrives, parse the `Set-Cookie` header to extract the `mwLastWriteTime` cookie and save it for use with subsequent read requests.

6. The body of the response will be a JSON string. Parse this to obtain the response envelope.

7. Check the `code` property of the response envelope. If it is "/api/status/ok", then the write query succeeded and the `result` property of the response envelope contains the query results. If the `code` property has any other value, then the write failed, and the `messages` property is an array of message objects that contain details.

In addition to special cookie and HTTP header requirements, there are two important differences between *mqlwrite* and *mqlread*. First, `mqlwrite` responds only to POST requests, not GET requests. Second, it does not support a `callback` URL parameter. These differences mean that it is impossible to invoke the *mqlwrite* service from client-side JavaScript code using a `<script>` tag.

The sub-sections that follow demonstrate the use of *mqlwrite* with Python code, explain how to submit multiple named queries in a single *mqlwrite* invocation, and provide details about *mqlwrite* envelope parameters, the `X-Metaweb-Request` header, the `mwLastWriteTime` cookie.

6.2.1. Using mqlwrite With Python

Example 6.2 is a `write()` method intended to be inserted into the `metaweb.Session` class of Example 4.15. It uses a number of the utility methods defined in that previous example, and the fact that these utilities can be shared between *mqlread* and *mqlwrite* code demonstrates how similar these two services are. In particular, recall that the _fetch utility method invokes the _http method, which does automatic cookie management. This means that our code does not have to explicitly handle authentication credentials or the mwLastWriteTime cookie. Also, remember that: _http automatically does an HTTP POST when a body is supplied for the request, _fetch does JSON parsing of the HTTP result, and _check raises a ServiceError if the code property of the response is not /api/status/ok.

Example 6.2. metaweb.py: invoking the mqlwrite service

```
def write(self, q, **options):
    """
    Submit the MQL write q and return the result as a Python object.
    Options specify envelope parameters. Authentication credentials,
    from a previous call to login() must exist in the cookie jar.
    Raises ServiceError if the query fails.
    """

    # We're requesting this URL
    url = 'http://%s%s' % (self.host, WRITE)

    # These headers identify how the query is encoded in the request body
    # and guard against XSS attacks.
    headers = {
        'Content-Type': 'application/x-www-form-urlencoded',
        'X-Metaweb-Request': 'True'
        }

    # Gather options that apply to this query
    opts = self._getopts("use_permission_of", **options)

    # Build the query envelope, adding options to it
    envelope = {'query': q}
    envelope.update(opts)

    # JSON encode the envelope
    encoded = self._dumpjson(envelope)

    # Use the encoded envelope as the value of the query parameter in
    # the body of the request.
    body = urllib.urlencode({'query':encoded})

    # Now do the POST and parse and check the response
    response = self._check(self._fetch(url, headers, body))

    # Return the result from the response envelope
    return response['result']
```

6.2.2. Example: US State Quarters

Example 6.1 and Example 6.2 are helpful methods, but they are just utilities, not real-world examples of how you might write to Metaweb. For the sake of example, let's suppose that you are a coin collector and you think that *freebase.com* should include information about each of the coins issued by the US Mint under its 50 State Quarters program. First, visit the US Mint's website[1] to find out when each state quarter was released, how many were minted for each state, and when each state became a state.

Next, create a Metaweb type to model this data. Login to *sandbox.freebase.com* and create a new type named "US State Quarter" in your default domain. For our example, we'll use the type id `/user/docs/default_domain/us_state_quarter`.

Next, give your type four properties:

- A property named "State", of `/location/us_state` to specify the state with which the quarter is associated.
- A property named "Release", of `/type/datetime`, to specify the date on which the quarter was released into circulation.
- A property named "Statehood", of `/type/datetime` to specify when the state gained statehood.
- A property named "Mintage", of `/type/int`, to specify how many quarters were minted.

Optionally, make each of these properties unique by clicking the "Restrict to one value" checkbox.

Next, you need to get your data into manageable form. Extract data from the US Mint site, and arrange it in a plain text file named `quarters.txt` that looks like the following:

```
Delaware,1999-01-04,1787-12-07,774824000
Pennsylvania,1999-03-08,1787-12-12,707332000
New Jersey,1999-05-17,1787-12-18,662228000
Georgia,1999-07-19,1788-01-02,939932000
Connecticut,1999-10-12,1788-01-09,1346624000
Massachusetts,2000-01-03,1788-02-06,1163784000
```

Each line in this file is the data for a single quarter. Fields are separated by commas. The first field is the name of the state. The second and third fields are the release date and statehood date for that state. And the fourth field is the mintage for that state quarter.

With our type created, and the data in this format, we can now write a simple script to upload the data to *sandbox.freebase.com*. Example 6.3 shows how to use the *metaweb.py* module to do this. Note that you need to insert your own Freebase username and password into the script to make it work for you.

[1] http://www.usmint.gov/mint_programs/50sq_program/index.cfm?action=schedule

Example 6.3. quarters.py: writing a data set to Metaweb

```
import metaweb         # Use the metaweb module

USERNAME = 'username'  # Put your Freebase username and password here
PASSWORD = 'secret'

# The ID for our US State Quarter type depends on our username
TYPEID = '/user/' + USERNAME + '/default_domain/us_state_quarter'

# Create a metaweb.Session object to interact with the sandbox server
sandbox = metaweb.Session("sandbox.freebase.com")

# Make sure we can log in before we go any further
sandbox.login(USERNAME, PASSWORD)

# We will be creating multiple quarters in a single MQL query.
# We start with an empty array and add MQL writes to it in the loop below
query = []

f = open("quarters.txt", "r")                  # Open our file of quarter data
for line in f:                                 # Loop through lines of the file
    # Break each line into fields
    fields = line.strip().split(',')

    # This query creates a single quarter
    q = {'create':'unless_exists',             # Create a new object
         'id':None,                            # And return its id
         'type':["/common/topic", TYPEID],     # Make it a topic and a quarter
         'name': fields[0] + ' State Quarter', # The object's name
         'state': {'connect':'update',         # Connect to...
                   'type':'/location/us_state',  # a US State object...
                   'name':fields[0]},            # with this name.
         'release': fields[1],                 # Release date
         'statehood': fields[2],               # Statehood date
         'mintage': int(fields[3])}            # How many minted

    # Add this write to the array of writes
    query.append(q);

f.close()                                      # Close the data file

# Now send our one big query to Metaweb and get the result
result = sandbox.write(query)

# Display the id of the Metaweb object for each state
for r in result:
    print "%s %s: %s" % (r['create'], r['state']['name'], r['id'])
```

6.2.3. Sending Multiple Queries to mqlwrite

As we saw in Chapter 5, the MQL write grammar allows multiple independent writes to be specified in a single query as elements of a JSON array, and we took advantage of this fact in Example 6.3 to create all of our US State Quarter objects in a single invocation of *mqlwrite*.

Like *mqlread*, the *mqlwrite* service also allows multiple queries to be submitted using the `queries` parameter instead of the `query` parameter. To use the `queries` parameter, place one or more regular query envelope objects inside a new "outer envelope" object. The property names used in the outer envelope are arbitrary and are re-used in the response envelope. This is exactly the same for *mqlwrite* as it is for *mqlread*.

Suppose we want to create two objects in a single call to *mqlwrite*. Here are two envelopes that can accomplish that:

2 Writes in 1 Query	**2 Queries in 1 Envelope**
`{`	`{`
`  "query":[{`	`  "q1":{`
`    "create":"unless_exists",`	`    "query":{`
`    "name":"my test object #1"`	`      "create":"unless_exists",`
`  },{`	`      "name":"my test object #3"`
`    "create":"unless_exists",`	`    }`
`    "name":"my test object #2"`	`  },`
`  }]`	`  "q2":{`
`}`	`    "query":{`
	`      "create":"unless_exists",`
	`      "name":"my test object #4"`
	`    }`
	`  }`
	`}`

When you include multiple writes in a single query, the writes are executed atomically: they all succeed or they all fail. As a result, they are not allowed to depend on each other, and there is no way to tell what order they are executed in.

If you submit multiple queries using the `queries` parameter, they are not atomic. Each one succeeds or fails on its own – the outer response envelope includes separate response envelopes for each query, and each of these inner response envelopes has its own `code` property to indicate the success or failure of the query. Note, however, that queries are unordered, and there is no guarantee that they will be executed in the order in which they are written. Since execution order cannot be predicted, queries passed to in the same invocation of *mqlwrite* should not depend on each other.

Note that the `read()` method of Example 4.15 accepts multiple read queries and uses the `queries` parameter to *mqlread*. The `write()` method of Example 6.2, allows only a single write query and uses the `query` parameter instead. It would not be difficult, however, to modify the `write()` method to use the `queries` parameter instead.

Avoid Large Multiple Writes

Creating 50 objects in one request, as we did in the state quarters example is a reasonable, if uncommon, thing to do. But attempting 500 writes in one *mqlwrite* invocation (either 500 writes in one MQL query or 500 queries in one envelope) exposes your application to a higher risk of service timeouts and is not recommended. In practice, it is more common to make one write per request, even when working with large datasets. In general, keeping your write requests as small and simple as possible will make it easier to debug and diagnose any problems that arise.

6.2.4. Mqlwrite Envelope Parameters

Each query envelope passed to *mqlwrite* includes a property named `query` to specify the MQL query to be executed. Any other properties within the envelope object are known as envelope parameters and provide additional input to *mqlwrite*. At the time of this writing [2], the only supported envelope parameter for *mqlwrite* is `use_permission_of`.

6.2.4.1. Setting Write Permission for New Objects

By default, new objects created with MQL write queries are given a `permission` property of `/boot/all_permission`, which makes them modifiable by any Metaweb user. Once an object has been created, it is not possible to alter its `permission` property, so if you want to create an object with restricted write permissions, you must do so when the object is created.

The `use_permission_of` envelope parameter does exactly this. The value of this parameter should be an object id. The permission of the specified object is reused and becomes the permission of any objects created by the query. Note that this parameter is *not* named `use_permission` and its value should *not* be the id of a `/type/permission` object. Instead, specify the id of an object (typically a user, domain or type) whose permission you want to copy.

As an example, recall from Chapter 5 that we created a namespace `/user/docs/music/notes` to hold names for instances of our `/user/docs/music/note` type. We created that namespace with a MQL write query, and submitted the query using the query editor on the sandbox server. The query editor didn't set the `use_permission_of` parameter, and we ended up with a globally writeable namespace. It is more likely that we would want that namespace to be restricted to the same set of users that have permission to modify the `/user/docs/music` domain, so we should have submitted the query using this envelope:

```
{
  "query": {
    "id":"/user/docs/music",
    "/type/namespace/keys": {
      "value":"notes",
      "namespace": {
        "create":"unless_connected",
        "type":"/type/namespace",
```

[2]September, 2008

```
        "unique":false
      }
    }
  },
  "use_permission_of":"/user/docs/music"
}
```

The *metaweb.py* module uses Python's named arguments for specifying envelope parameters, and we can execute the write above with code like this:

```
import metaweb

username = "username"
password = "secret"
domain = "/user/" + username + "/music"

sandbox = metaweb.Session("sandbox.freebase.com", use_permission_of=domain)
sandbox.login(username, password)
sandbox.write({"id":domain,
               "/type/namespace/keys": {
                 "value":"notes",
                 "namespace": { "create":"unless_connected",
                                "type":"/type/namespace",
                                "unique":False }}})
```

6.2.5. The X-Metaweb-Request Header

The Metaweb *mqlwrite* service (and also the *upload* service documented later in this chapter) require a custom HTTP request header, named `X-Metaweb-Request` to be present in all requests. The value of the header is ignored, but if the header is not present in the request, the request will not be processed.

The requirement that the custom header be present is a security measure to prevent cross-site scripting (XSS) attacks. As a practical matter, it means that the *mqlwrite* and *upload* services cannot be invoked via HTML form submission, since there is no way to tell a web browser to add a custom header like this when POSTing a form.

6.2.6. Caching: mwLastWriteTime and Touch

For efficiency, Metaweb servers cache query results. Suppose you issue a read query and get a result. Then, another Metaweb user performs a write that alters the data you queried. If you re-issue your query, you are likely to get the same results (now cached) that you got the last time. After some time, the cached result will time out, and you'll see the new data written by the other user, but this will not happen right away.

If you're mixing reads and writes yourself, however, you always want your read queries to return the data you've just written, of course. Metaweb uses a cookie-based scheme to ensure that this happens. Any time you do a write with *mqlread* (or *upload*), the service returns a `mwLastWriteTime` cookie. Your web browser (or client code) then presents this cookie when it makes any subsequent read requests. This tells the *mqlread* service that it may not return any cached result older than the most recent write you've done. Your results may not arrive as quickly, but they will be consistent with the writes you've performed.

If you do your reads and writes using the *freebase.com* client, this cache management is invisible to you because your web browser handles cookies automatically. Similarly, the *metaweb.py* module developed in this chapter and Chapter 4 manages cookies automatically in the `_http` utility method, and the results returned by the `read()` method will be consistent with any previous writes made with the `write()` method.

If you develop your own library for working with *mqlread* and *mqlwrite*, you'll need to be aware of this caching issue and handle this `mwLastWriteTime` cookie appropriately.

Sometimes automatic cookie management is not enough to get proper caching behavior. Suppose you run one script to perform some writes, and then run another script to do some reads. Unless you saved the contents of cookie jar after doing the writes in the first script, and initialized the cookie jar of the second script from the saved state before doing the reads, you may get cached results that do not include the newly written data. Even trickier is the case where you do writes in the client (adding a property to a type, for example) and then do reads in a script. Unless you can make your script share the cookies of your web browser, you're likely to run into trouble. Another possible source of difficulty is when you perform reads in your web browser to verify the success of writes you've done in a script. If you check the state of an object with your web browser, and then run a script to modify that object, and then check the object again in your browser, you may not see the modification you've just made.

To solve these problems you need a way to obtain a current `mwLastWriteTime` cookie. The *touch* service does just that. If you make an HTTP GET request to */api/service/touch*, the Metaweb server will send you fresh `mwLastWriteTime` cookie that allows any subsequent reads to see the current state of the database. At the time of this writing [3], you can do this in the *freebase.com* client by typing **F8** to open the "Dev Tools" box at the bottom of the web page. Once you've done that, scroll to the bottom and click the "Refresh cache" link that has appeared. You can then type **F8** again to remove the Dev Tools.

Example 6.4 shows how to invoke the *touch* service in Python. It defines a simple `touch()` method to be added to the `metaweb.Session` class of Example 4.15:

Example 6.4. metaweb.py: invoking the touch service

```
def touch(self):
    """
    Defeat caching by requesting a current mwLastWriteTime cookie.
    This method returns nothing.
    """
    self._fetch("http://" + self.host + TOUCH)
```

Despite its name, the value of the `mwLastWriteTime` cookie is not just a timestamp: it is an opaque value that contains cache information other than just the last write time. For this reason, it is not

[3]September, 2008

possible to simply create your own value for the cookie – you have to obtain a valid value from *mqlwrite*, *upload*, or *touch*.

6.3. Uploading Data to Metaweb

The unique feature of Metaweb is the way that it stores relationships between objects. But, like any database, it can also store large chunks of data, such as long HTML documents or binary image files. In Chapter 4 we learned about the *trans* service for retrieving content. Here, we'll learn how to upload content to be stored in Metaweb. The service for uploading content is named *upload*, and has the URL path `/api/service/upload`. The *upload* service responds only to HTTP POST requests. The data to be uploaded is passed in the body of the request. The MIME type of the content, as well as the encoding of textual content is specified in the HTTP `Content-Type` request header.

6.3.1. An Upload Utility

Example 6.5 shows an `upload()` method for our *metaweb.py* module. It expects two arguments: a string of content (Python strings can be binary data or textual data) and the MIME type for the content. `upload()` creates a new `/type/content` object to hold your content and returns the id (in the `/guid` pseudo-namespace) of that object to you. This guid can be used to retrieve the content with the */api/trans/raw* service (see Chapter 4).

No Duplicates

The *upload* service does not always create a new `/type/content` object for the content you upload. All content is checksummed when it is uploaded, and these checksums are used to detect duplicate uploads. If the content you are uploading already exists in the Metaweb content store, the existing `/type/content` object is used. The `blob_id` property of `/type/content` holds the checksum value.

Example 6.5. metaweb.py: uploading content to Metaweb

```
def upload(self, content, type):
    """
    Upload the specified content (and give it the specified type).

    Return the guid of the /type/content object that represents it.
    The returned guid can be used to retrieve the content with
    /api/trans/raw.
    """

    # This is the URL we POST content to
    url = 'http://%s%s'%(self.host, UPLOAD)

    # These are the HTTP headers
    headers = {
        'Content-Type': type,
        'X-Metaweb-Request': 'True'
        }

    # POST the request, parse the response, check for success
    # This will raise an exception on failure
    response = self._check(self._fetch(url, headers, content))

    # Return the id of the uploaded conent
    return response['result']['id']
```

6.3.2. Examples: Uploading Images of State Quarters

In order to demonstrate the `upload()` method of Example 6.5 let's continue with our US State Quarter example. The US Mint has images of each of the state quarters on its website. Let's upload those images to *sandbox.freebase.com*, and make them visible through the `/common/topic/image` property of each quarter object. Example 6.6 shows how to do this.

In order to understand this code, you have to know that the *upload* service handles images specially. When you upload images, the `/type/content` object is also given the type `/common/image` with various properties filled out. Because your uploaded `/type/content` is *co-typed* as `/common/image`, you can link directly to image content from `/common/topic/image`.

Example 6.6. quarterpix.py: uploading images to Metaweb

```
import metaweb             # Our metaweb utilities
import urllib2             # For downloading images from the mint server

USERNAME = 'username'      # Put your Freebase username and password here
PASSWORD = 'secret'

# The ID for our US State Quarter type depends on our username
TYPEID = '/user/' + USERNAME + '/default_domain/us_state_quarter'

# Create a metaweb.Session object to interact with the sandbox server
sandbox = metaweb.Session("sandbox.freebase.com")

# Make sure we can log in before we go any further
sandbox.login(USERNAME, PASSWORD)

# All the images files are beneath this URL
imagedir = 'http://www.usmint.gov/images/mint_programs/50sq_program/states/'

# This dictionary maps state name to image name.
images = { 'Delaware': 'DE_winner.gif',
           'Pennsylvania': 'PA_winner.gif',
           'New Jersey': 'NJ_winner.gif',
           'Georgia': 'GA_winner.gif',
           'Connecticut': 'CT_winner.gif',
           'Massachusetts': 'MA_winner.gif'}

# Loop through the states
```

```
for state,filename in images.items():
    # First, download the image from the Mint's website
    image = urllib2.urlopen(imagedir + filename)
    type = image.info()['Content-Type']
    content = image.read()

    # Now upload it to Metaweb
    id = sandbox.upload(content, type)

    # Define a write query to link the quarter object to the uploaded image
    query = { 'type': TYPEID,
              'state':state,
              '/common/topic/image': { 'id':id, 'connect':'insert' }}

    # Submit the query and get the result
    result = sandbox.write(query)

    # Output the result
    print "%s: %s %s" %(state, result['/common/topic/image']['connect'], id)
```

Once you have run the code in Example 6.6, use the *freebase.com* client on the sandbox server to view your state quarter objects (refreshing the cache if necessary). You'll see that there are now images on the page.

6.3.3. Uploading Documents

It is fairly easy to upload an image and make it visible to users of *freebase.com*. It is a little trickier to do the same for textual content. When you upload a document, a `/type/content` object is created for that document. This allows the content to be retrieved with */api/trans/raw*, but it doesn't allow it to be viewed in any natural way in the client. To accomplish that, you must create a `/common/document` object to reference the content, and (optionally) a `/common/topic` object to reference the document. Example 6.7 shows how you can do this.

Example 6.7. uploaddoc.py: uploading HTML documents to Metaweb

```
import sys, re, metaweb

# Read the content of the file specified on the command line
# It must be an HTML file with a <title>
filename = sys.argv[1]
try:
    f = open(filename)
    doc = f.read()
    f.close()
except Exception, e:
    sys.exit(e)

# Search through the document for a title
try: title = re.search("(?i)<title>(.*)</title>", doc).group(1)
except: sys.exit("Document has no title")
```

```
# Log in to the sandbox server.
USERNAME = 'username'    # Put your own username and password here
PASSWORD = 'secret'
sandbox = metaweb.Session("sandbox.freebase.com");
sandbox.login(USERNAME, PASSWORD)

# Upload the document content to Metaweb.
# Note that we hardcode the text/html content type.
content_id = sandbox.upload(doc, "text/html")

# Submit a MQL write query to create a /common/topic and /common/document
# for the uploaded content
result = sandbox.write({'create':'unless_exists',
                        'type':'/common/topic',
                        'id':None,
                        'name':title,
                        'article' : { 'create':'unless_exists',
                                      'type':'/common/document',
                                      'id':None,
                                      'content':content_id }})

# Tell the user what we did
print "Uploaded %s: %s\n\tcontent: %s\n\tdocument: %s %s\n\ttopic: %s %s" % (
    filename, title, content_id,
    result['article']['create'], result['article']['id'],
    result['create'], result['id'])
```

This Python program expects the name of an HTML file as a command-line argument. It reads the file and determines the document title by searching (using a regular expression) for a `<title>` tag. It uploads the document text with the `upload()` method of Example 6.5. Then it submits a MQL write query to create a `/common/topic` that refers to a `/common/document` that refers to the uploaded content. It uses the document title as the name of `/common/topic` object.