This Document

This document discusses Morris Templating Language (MTL) Biz (which stands for "business") tags: their uses, usage and syntax.

It refers to concepts discussed in "Basic Concepts of MTL" and in "Common Tags." If you haven't already read these documents, you are encouraged to do so.

Conventions

- This document will show sample results as they would appear in a browser.
- In commentary text, data tags will be discussed in terms of what comes after the second dot. I.e. `<mtl biz.business.id>` will generally be referred to as "id" or "the id tag." In example code, the tags will be fully written out as they should appear in a template.

  Query qualifiers, on the other hand will always appear as they should in working code; i.e. there are no "other parts" to a query qualifier. A qualifier described as "id" in commentary should be written "`id=`" in an actual query.
- "Template," as it will be used in this text, can mean two things: [1)] a document in Template Manager (TM) and its contents and/or [2)] a section of MTL code that (should) stand alone and work without other supporting MTL code.
- Anything appearing in square brackets: [like this] is just a place holder. Do not insert the square brackets and text inside into a template; replace it with appropriate values.
- `Code to be typed into a template will look like this.`
- Result text (template output) will look like this.

Miscellaneous Notes

- This document is not intended to be an exhaustive tag reference. Some sections cover every query qualifier and data tag available in a tag set (at this writing,) others do not. For a full tag reference consult the "MTL Tag Quick Reference Guide."
- Line breaks may be inserted in the middle of MTL tags for clarity throughout this document, but line breaks are NOT LEGAL inside MTL tags[*]. When/if you copy the code in this document into templates, REMOVE ALL LINE BREAKS that appear between instances of "<mtl" and ">".

About the Author

G. Ann Campbell has been writing MTL since 1999 - when she started by timidly changing the background colors of table cells on onlineathens.com's classifieds templates.

In 2000 she joined MDW and began taking on larger MTL projects, such as the construction of the calendar template set deployed to most Morris Communications Corporation Web sites, and building out classifieds template sets from scratch.

She began teaching MTL in 2001 and started the odyssey of documentation shortly thereafter.

Since then her mission has been to explore strange new tag sets, to seek new uses and new output formats. To boldly go where no MTL-coder has gone before.

* Except in the values of variables. It's perfectly okay to put a line break inside a variable:

```
<mtl $okayLinebreak="The following line break is fine.
It ends up in the value of the variable.">
```

Table of contents

About Biz

Biz MTL is about businesses - where they are, what they do, how to find them & etc.

It's built and organized to give you, the template writer, as many angles on data retrieval as possible. You can select businesses based on geographical criteria, by type, by name or based on other features (such as whether they've ponied up for a prominent listing.)

Most biz data is entered through BizLink - the Java-based data entry client for business. If you don't have a BizLink account, contact your Support representative for download instructions, account setup and training in how to use the application.

Before we start into Biz MTL, you need to know a little bit about the underlying database BizLink puts data into and Forge pulls it back out of. It's a shared resource - shared among you and your sister papers.

And in it there's only one record for each business. Just like there's only one "Joe's Underground" at Broad Street and 8th in Augusta, Georgia, there is (or should be) only one record for it in the database.

So if you modify or delete a business in the database, keep in mind that you're not just affecting how that business shows up from your templates, but how it shows up from everyone else's too.

Getting Started

If there's not already a category named "business" or "biz" under your property in TM , create one. If there already is a biz or business category, create a new category anyway and name it whatever you want.

You should still name your category descriptively, but the actual category names don't really matter - none of the biz data will be stored here. The point is merely to have one location for all the business-related templates you'll be creating as you work through the exercises here. At the same time, you want a category that has only your templates, so when you're told to create a template at a specific level and sub-type you can do so without interfering with existing templates.

Biz.Business_Type

Biz.business_type is one of the simplest queries in Biz. It pulls back the elements in the user-defined list of business types. Business types are named groupings of Standard Industry Classifications (SIC's.) For instance codes that correspond to Veterinary, Pet Training and Animal Shelters are grouped together in the Animal Care business type.

Here's the query in its most basic form:

```
<mtl biz.business_type>
  <mtl biz.business_type.name><br>
</mtl biz.business_type>
```

At the time of this writing, it returns 61 records, but since business types are user-defined, that could change.

There are only two non-universal **data tags** for this query:

- id - the unique id of the business type
- name - the name of the business type

Of course, all the universal data tags, like index and the pagination tags, are available here, but we won't be covering them explicitly with each tag set. Just understand that they're generally available.

The **qualifiers** available for this query are pretty straight forward, too:

- `id=` the id or id's of the business type(s) you're looking for
- `name=` the name or names of the types you're looking for
- `business_id=` Plug in here the id of a business you'd like to find the type for. If your business has no SIC's, or if its SIC's are not included in business type(s) this will return 0 records - even for a valid business id.

Exercises

Ex. 1: Write a level one browse template to get a list of business types.

Biz.SIC

SIC codes are a way of very specifically defining what a business does. The differences between SIC's can be minute. For instance SIC 2040 is "Grain mill products" while 2041 is "Flour and other grain mill products." For obvious reasons, most businesses will probably fall into multiple SIC's.

You'll use the biz.sic query to get lists of SIC's. The lists might be of SIC's applied to a given business or of the SIC's in a business type. Here are the **qualifiers**:

- `business_id=` get SICs for specified business or businesses - specified by id
- `business_type=` get SIC's for specified business type
- `business_type_id=` return SIC's for the business type specified by id
- `sic=` return records for the specified SIC. (This is a one-to-one relationship. Feed one SIC in here and you'll get the single corresponding record.)
- `title=` return SIC's with the specified title. (This is *usually* a one-to-one relationship. Feed one SIC in here and you'll get the single corresponding record.)
- `title_contains=` return SIC's with titles that contain the specified string of text

Here are the **data tags** for biz.sic:

- business_type - the broad classification for this SIC. A given SIC may be in multiple business_types
- business_type_id - id(s) of the business type(s) the SIC is in
- sic - Code number of SIC. Also the unique numeric identifier of SIC record.
- title - title of SIC

You might have noticed at this point that the standard "id" and "name" qualifiers and data tags are missing. Almost every tag set in MTL has that pair - except this one. Why? Well, the id you'll find in every other table is supposed to make sure that there's at least one unique thing about every record. You can't rely on name to be unique (e.g. McDonald's) or on address (most towns have a Main Street, for instance.) So usually, there's a unique, database-assigned id for every record.

But with SIC's it's different. SIC's are by nature unique. If there were two SIC's with the same number (say "Cotton Candy manufacturing" and "Wood Pulping" both had SIC 2430) it would screw the whole system up. So we can rely on SIC's to be unique - and that makes having a database-assigned id unnecessary.

Exercises:

Ex. 1: Write a level two browse template to get a list of SIC's in a business type. Use a variable for the business type name, but make sure your template tests for a blank value in that variable and sets a default value if it's blank.

Ex. 2: Revise the template you made in the last chapter (Biz.Business_Type Ex. 1) and make it link to your template from Ex. 1 in this chapter - so that you can click through on the business type and get a list of the SIC's in the type.

Biz.Market

This is the other "simplest query" in Biz MTL. Like biz.business_type, it deals with user-defined groups. This time the groups are of cities. Typically, this is used for metro areas, and markets are named after the dominant city. For instance, the Jacksonville (Fla.) market contains Jacksonville, Fla.; Jacksonville Beach, Fla.; Keystone Heights, Fla.; Atlantic Beach, Fla.; etc.

This query only has two non-standard **qualifiers**:

- `name=` the name of the market you're looking for
- `id=` the id of the market you're looking for

It only has two non-standard **data tags**:

- name - the name of the market
- id - the market's unique id

Market is somewhat unusual in that each of its two pieces of data is unique. That is, in addition to having an unique id, a market's name must be unique - so there will be no confusing the Jacksonville, Fla. market with the Jacksonville, Ala. market.

Generally you'd use this query to get a list of markets or possibly to get a given market's id, although most of the queries that have a market_id qualifier also have a market_name qualifier.

Exercises:

Ex. 1: Write a level three browse template to get a list of markets.

Biz.City

Use this query to find out about cities.

Here are the **qualifiers** of note:

- `country=` get city records for this country or countries
- `id=` get the record for this id or list of ids
- `market_id=` get the cities this in market or markets
- `market_name=` get the cities in this market or markets
- `name=` Return only records for cities with this name. May be one or more city names.
- `state_abbr=` Return only city records in this state. May be one or more 2 letter state abbreviations.

Here are the **data tags** of note:

- country - the city's country
- id - unique numeric identifier of city
- name - name of the city
- state_abbr - abbreviation of state in which city exists

Notice that you can qualify by market to get the cities in a market, but because it's not a one-to-one relationship (a city can be in multiple markets) you can't get the markets a city is in from the biz.city query.

Exercises:

Ex. 1: Write a level four template to get a list of cities in a market. Use a variable for the market name, but make sure your template tests for a blank value in that variable & sets a default value if it's blank.

Ex. 2: Revise the template you made in the last chapter (Biz.Market Ex. 1) and make it link to your template from Ex. 1 in this chapter - so that you can click through on the market name and get a list of the cities in the market.

Biz.Business - geographic selection

Now we come to the heart of the matter. Biz is, of course, all about businesses, and this is the query you use to find out about them.

Here's a basic query to do just that:

```
<mtl biz.business city="aberdeen">
     <mtl biz.business.name><br>
     <mtl biz.business.street1><br>
     <mtl biz.business.street2><br>
     <mtl biz.business.city>, <mtl biz.business.state_abbr>
     <mtl biz.business.zip><p>
</mtl biz.business>
```

Notice that I've hard-coded a city, Aberdeen, into the query.

What I get back from it is the names and addresses of 23 businesses.

Of course, those 23 businesses are spread across six states and two countries… so this is maybe not what I wanted.

Let's try again:

```
<mtl biz.business city=$city state_abbr=$state zip=$zip>
     <mtl biz.business.name><br>
     <mtl biz.business.street1><br>
     <mtl biz.business.street2><br>
     <mtl biz.business.city>, <mtl biz.business.state_abbr>
     <mtl biz.business.zip><p>
</mtl biz.business>
```

Notice, I've switched from a hard-coded value for city to a variable. That makes my template more flexible. I've also added state abbreviation and zip code qualifiers. Now I can use a city/state combination to narrow my results down to a single city - since most states won't have two cities with the same name - or I can get the businesses for a single region in a city with the zip code qualifier.

I could also tackle the same-city-name-in-multiple-states issue using the city_id qualifier… if I know the city's id.

So now I can narrow down my geographic area. But what if I need to widen it?

Let's say I'm looking for businesses in Augusta, Ga. and Aiken, S.C.

When I break those two city/state combinations into parts, I get:

```
<mtl $city="Augusta,Aiken">
<mtl $state="Ga,SC">
```

The only problem with that is that if we use these two variables for the template above, it'll pull out businesses in four cities: Augusta, Ga; Augusta, SC; Aiken, Ga and Aiken, SC – not

just the two I'm expecting. Granted, two of these four cities probably don't exist, but we can't always count on that.

So how do we tackle that? We could get the city id's and use the city_id quaifier. Or we could use markets.

As discussed above, markets are groups of cities - usually cities that are geographically close to each other. And in fact, Augusta, Ga and Aiken S.C. are close enough to each other to be in the same market. So if a market containing the two doesn't already exist, we'll use BizLink to create one (and name it "Augusta.") Of course, the Augusta-Aiken area contains a number of cities, and they belong in the market too. When we're done creating this particular market, it ends up with about 28 cities of varying size.

You may be wondering where this is going. We set out to find businesses in Augusta, Ga. and Aiken, S.C., and now we have a market with 26 extra cities. But that's not a problem. Here's how we'll use our new market:

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
<mtl biz.business market=$market city=$city state_abbr=$state zip=$zip>
     <mtl biz.business.name><br>
     <mtl biz.business.street1><br>
     <mtl biz.business.street2><br>
     <mtl biz.business.city>, <mtl biz.business.state_abbr>
     <mtl biz.business.zip><p>
</mtl biz.business>
```

Notice the new qualifier: market=$market. We're setting a default value in case none is passed in, but we're using a variable here, and continue to use variables for the other qualifiers to keep our template flexible. We'll access the template like this:

```
http://stats.morris.com/forge?classification=[your
classification]&temp_type=[your type]&market=augusta&city=aiken,augusta
```

When we ask for businesses in Aiken and Augusta, we get all businesses in all of the Aikens and all of the Augustas in the database. Let's suppose that brings back 1,200 records. But then we further qualify (i.e. "narrow down") our search results to just businesses in our Augusta market - which will eliminate the businesses that are in *other* Augustas and Aikens.

Alternately, we could get the Augusta market's id and use that with the market_id qualifier.

I want to take one more whack at this before we leave the subject. We've been doing addresses the hard way, and there's the question of whether or not `<mtl biz.business.street2>` comes out blank, leaving an extra blank space in a listing. We could wrap an if statement around it to suppress the extra line break when it's blank, or we could use a shortcut tag, like this:

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
```

```
<mtl biz.business market=$market city=$city state_abbr=$state zip=$zip>
     <mtl biz.business.name><br>
     <mtl biz.business.street_city><p>
</mtl biz.business>
```

The `<mtl biz.business.street_city>` tag outputs all the non-blank street data, followed by the name of the city.

Qualifiers we've discussed in this chapter:

- `city=` city name or names you're looking for businesses in. Be aware, that city names are not unique. Searching by city name alone will likely return more than you expect.
- `city_id=` the unique id(s) of the cities you're looking for businesses in. Be aware that because data entered into BizLink is entered by humans, you can't count on it to be consistent. So the same city may appear in the database multiple times - with minute differences in spelling, spacing or punctuation. Each variation of a city name will get its own city id, i.e. "Auburndale FL" and "Auburndale FLA" would have two separate entries and therefore two separate id's in the database. So if you query by city id, you may need several id's to get all the businesses in a single city.
- `state_abbr=` the abbreviation of the state you'd like to find businesses in
- `zip=` the zip code you're looking for businesses in
- `market_name=` the wide geographic area you're looking for businesses in. Note, markets must be set up through BizLink before you can successfully query by them.
- `market_id=` the id of the market you're looking for businesses in. Since market names are unique, using the market's id has little advantage over using the market name itself.

Basic business info **data tags**:

- name - the name of a business
- street1 - the first line of the business' street address
- street2 - the second line, if there is one, of the business' street address
- city - the city the business is located in
- street_city - full street and city address of the business
- state_abbr - the abbreviation of the state the business is located in
- zip - the business' zip code
- country - the country the business is located in.

Exercises:

Ex. 1: Create a level 1 detail template to find businesses in a city

Ex. 2: Revisit your template from Biz.City, Ex. 1 and make it link to your template from Ex. 1 in this chapter - so that you can click through on the city name and get a list of the businesses there.

Business - topical selection

So now you can find the businesses in a given area - but you don't necessarily want to sort through all of them to find just the restaurants, for example.

That's where SIC's and business types come in:

```
<mtl biz.business business_type=$type sic=$sic [geographic qualifiers]
orderby=name>
     <b><mtl biz.business.name></b><br>
     <mtl biz.business.short_description><p>
</mtl biz.business>
```

Let's say we pass `market=Augusta&type=dining` in through the URL. Assuming all the restaurants in Augusta have appropriate SIC codes, (remember, this data entry function is not something that happens automatically) we should get a list of restaurants - nothing but restaurants - in Augusta.

We've also added `sic=$sic` to the query. Now we can find every business in a broad category, or narrow it down to a single SIC (or comma-delimited list of SIC's.)

Notice, we've changed the data tags in the template; dropped the address info and added `<mtl biz.business.short_description>`. There are actually two description fields in BizLink, and so two description tags; short description, which is supposed to be just that: a one-liner, and `<mtl biz.business.description>` with presumably more detail.

Depending on your application, you might want to show the name and brief description in a long list of businesses (use `next=name` in the close query tag so you only get one listing for McDonald's, for instance, in your list), then let users "drill down" to a details page with the business' addresses, full description and contact info.

Qualifiers from this chapter:

- `business_type=` Find businesses in these broad categories. These are SIC-dependent, so if a business that should show up under a given type doesn't, check its SIC codes.
- `sic=` find businesses with these specific SIC's applied to them

New **data tags** in this chapter:

- short_description - a one-line description of the business
- description - a fuller description of the business

Exercises:

Ex. 1: Modify your template from Biz.Business - geographic selection, Ex. 1 to add the ability to find businesses by business_type or SIC. Pare down the information it displays to just the name (one instance per chain) and the brief description.

Ex. 2: Revisit your template from Biz.SIC, Ex. 1 and make it link to your template from Ex. 1 in this chapter - so that you can click through on a SIC title and get a list of the businesses in the SIC.

Specific businesses

So far, we've looked at finding businesses in broad categories, basically browsing by topic or location. Sometimes though, that's too cumbersome.

Say you just want to find a McDonald's. Do you have to go through all the steps? Nope. Not with these **qualifiers**.

- `business_id=` An internal number that uniquely identifies a business. May be a list of numbers separated by commas. (You can't get much more specific than that.)
- `name=` The name of the business you want. (Case does not matter.) May be a list of names separated by commas.
- `name_contains=` Only return business whose names contain the specified characters. (Case does not matter.)
- `name_startswith=` Only return business whose names begin with the specified characters. (Case does not matter.)

To make these qualifiers worthwhile, you'll need to give the user a search interface. The `name=` qualifier requires an exact match - which most users won't hit. The `name_startswith=` qualifier… well, what it does is pretty obvious from the name. I wouldn't use it on my main interface for the simple reason that it'll probably narrow down my search matches too far. For example, if I'm looking for a business I know as "Muffler Shoppe" and search for "muffler" the business I'm looking for won't come back if it's actually "The Muffler Shoppe."

So for the main search interface, I'd rely on the `name_contains=` qualifier. That is, if my form input is this:

```
<input type=text name=namecontains>
```

I'd plug the variable that input is translated to (for more on this, see *Form-ing* in "MTL Common Tags") into the `name_contains=` qualifier:

```
<mtl biz.business [other qualifiers] name_contains=$namecontains>
```

I'd probably still have the other qualifiers in my results template - but only offer those options in an advanced search.

Now we've got it narrowed down to just a few businesses, and hopefully the list contains the one we were looking for. But all we've got on that business is a name and brief description - if it's not blank. And in the case of chains like McDonald's, we're only getting the name in the list once - even though there are probably several in the area.

So it's time to start pulling out more info.

This time, our query will use the `name=` qualifier because we have the exact name, and we want to get all instances of businesses with that name. Of course, we still want it narrowed down geographically:

```
<mtl biz.business name=$busname market=$market city=$city state=$state
zip=$zip orderby=street1>
```

In addition to the address tags, other **data tags** you may want to use on your "more info" page are:

- phone - If there is more than one phone number each number will be separated by a space. You can change the delimiter by specifying `separator=.` Normally this tag returns phone numbers of type "phone" other phone number types such as "toll-free", "pager", "cellular" and "voicemail" may be returned by specifying the types in parenthesis after the tag:
  `<biz.business.phone(voicemail,pager) separator="<br>">`
- fax - If there is more than one fax number each number will be separated by a space. You can change the delimiter by specifying `separator=.`
- email - the business' email address
- url - the business' URL

Exercises:

Ex. 1: Write a level 1 search template that submits to your level one detail template. Give the user inputs for business name, market or city and business type. Make sure your detail template honors the input from the search form.

Ex. 2: Write a level two detail template to give detailed information about businesses with a specific name. Revisit your level one detail to link business names to this template.

More business criteria

So far, we've looked at getting the basic information for every business in a certain area and/or industry. But depending on your business model, you may want to exclude some businesses (the ones that haven't paid to be listed) or make some businesses more prominent (the ones who've paid extra.)

First, the exclusion/inclusion issue. In BizLink you can give a business registration dates. Presumably, you'd use these to indicate the time span a business has "subscribed" to your listing service for.

And if a business has paid extra to show up more prominently, you'd use BizLink's "rank" input to indicate that.

So now you know where it goes in. How do you pull it back out? With these **qualifiers**:

- `reg_date=` accepts a single date in any of the standard formats MTL recognizes (for more see *Time, Time, Time* in "Common MTL tags.") If the value does not fall between a business' reg_start and reg_end, that business will be not be returned
- `rank=` accepts a single number, or a comma-delimited list of numbers. It returns business that have one of the specified ranks.

There are **data tags** for these things too:

- rank - Optional rank field.
- reg_start - The date the registration period begins.
- reg_stop - The date the registration period ends.

But it's hard to imagine that you'd want to output these pieces of data - unless you wanted to do something like "member since `<mtl biz.business.reg_start>`" or unless your template is for internal reporting purposes, rather than for public consumption.

On the other hand, you'd certainly want to order your results by rank, and possibly reg_start. Keep in mind, though, that both are optional inputs - so if you just do this:

```
<mtl biz.business [qualifiers here] orderby=rank,reg_start>
```

Unranked businesses will come out on top - even before the businesses ranked "1." So do it this way instead:

```
<mtl biz.business [qualifiers here]
orderby=rank(""):d,rank,reg_start(""):d,reg_start>
```

This orderby throws businesses without rank to the bottom of the list. For details on how/why, see *Advanced Sorting* in "MTL Basic Concepts."

Features

So we've established the basics about our businesses - including whether they've ponied up for prominent placement - but there will always be a lot more to tell about a business after the name, address and phone number. Just what though, will vary from industry to industry. For a restaurant, you may want the median entrée price, but that would be totally irrelevant for an auto parts store. And that's where features come in.

Features hold the extra data about a business that varies from industry to industry. They're grouped into named categories and associated with business types (named collections of associated SIC's.) Because it would be silly to even have the option to input the price of an oil change when you're dealing with a restaurant, feature categories are associated with business types and only show up for businesses with the relevant SIC's.

You can think of a category like a table. Each category will have a column for each feature in the category, and a row for each set of values.

For instance, let's say you're trying to profile the golf courses in your area. So under BizLink's Features tab, you create a category named "Golf Course Hole," and give it columns named hole number, par, yardage, handicap and course name (because some courses have more than one course.)

Now, for businesses with a SIC in the Golf business type, "Golf Course Hole" will show up in the Feature Category dropdown in the business' Features tab. (Note, there are two different tabs labeled "Features." There's a Features tab in the main window and a Features tab that shows up in the "Edit a Business" window.)

When you start filling in hole information for a golf course, you'll have one row for each hole in the course. So for the rows for Samples Country Club's nine-hole course, SampleWood, and the first four holes of the 18-hole course, SampleVistas, it might look something like this:

Golf Course Hole				
Hole Number	Par	Yardage	Handicap	Course Name
1	4	372	9	SampleWood
2	4	356	13	SampleWood
3	4	406	1	SampleWood
4	4	411	3	SampleWood
5	4	387	11	SampleWood
6	3	180	17	SampleWood
7	5	510	5	SampleWood
8	3	187	15	SampleWood
9	5	485	7	SampleWood
1	3	327	12	SampleVistas
2	4	420	3	SampleVistas
3	3	280	8	SampleVistas
4	5	502	4	SampleVistas

So that's how it gets set up and input. Now to pull it back out…

But before we go any further, take another look at the table above and make note of four significant parts. There's the **category** name, "Golf Course Hole;" the **feature names** - i.e. the column labels; and there are **rows** of **values**.

In the table above, the "Golf Course Hole" **category** has five **features**. They're named "Hole Number," "Par," "Yardage," &etc. The table shows 13 **rows**. The **value** of the Handicap **feature** in the third **row** is "1." The **value** "1" shows up twice in the "Hole Number" **feature**.

Not to beat a dead horse, but understanding the "anatomy" of this table will help you in the next few chapters as we discuss how to output this input.

Biz.Feature_Row

As you might guess from the name, this query deals with features a **row** at a time.

The query's **qualifiers** are what you'd expect:

- `business_id=` Return only feature rows for this business. May be one or more business ids.
- `category_id=` id of the single category you want values from
- `category_name=` name of the single category you want values from
- `city_id=` Return only feature values for businesses in this city. May be one or more city id numbers.
- `city_name=` return only feature values for businesses in this city.
- `market_id=` Return only feature values for businesses in this market. May be one or more market id numbers.
- `market_name=` return only feature values for businesses in this market

Notice that the category_id= and category_name= qualifiers can only take a single category at a time. Why? Well, it's part of what makes this particular query unique. Before we delve into it, though, let's take a look at the **data tags**:

- business_id - A number that uniquely identifies the business that this feature row is for.
- category_id - A number that uniquely identifies the category for this feature row.
- category_name - The name of the category that this feature row is in.
- NAME - Replace "NAME" with the feature name. This will pull out the value of that feature. Use underscores ('_') to replace any spaces in a feature name.

Hopefully you haven't gotten to the point where you're just skimming this documentation. If you have, maybe the fact that "name" is in upper case caught your eye. It's not a misprint. If you didn't already read it, go back and read the description for NAME.

If you're scratching your head now, that's okay. You've come to an unusual point in MTL: a roll-your-own-tag tag.

Here's the deal: Usually the tag name - what comes after the second dot in a tag - corresponds to the name of a column. (For a refresher on this concept, see *Almost There* and *Pseudo MTL* in "MTL Basic Concepts.") But business features (i.e. the column names) are user-defined. So there were really three choices. [1)] Write a tag for every word or combination of words that might get used for features. [2)] Let users define features at will, but make them wait in the engineering queue for the corresponding data tag(s) to be written. [3)] Let users roll-their-own tags, just like they're rolling-their-own features.

It was neck-and-neck between the first and third choices for a while, but after considering the size of just the abridged dictionary, calmer heads prevailed and we got option number three.

So let's take a look at our example golf feature category:

```
<mtl biz.feature_row category="golf course hole" business_id=$bus_id>
     <mtl biz.feature_row.course_name>
     <mtl biz.feature_row.hole_number>:
     par <mtl biz.feature_row.par>;
     <mtl biz.feature_row.yardage> yds.;
     <mtl biz.feature_row.handicap> handicap<br>
</mtl biz.feature_row>
```

Notice the data tags. You don't see "yardage" or "par," for instance, in the data tags I've listed, and you won't find them in the "MTL Tag Quick Reference Guide" either. They're instances of our roll-your-own tag, NAME.

Notice too that the column named "Course Name" becomes "course_name" and "Hole Number" becomes "hole_number" when used in a data tag - case insensitive and with underscores for spaces. Here's what we get:

SampleWood 1: par 3; 372 yds.; 9 handicap
SampleWood 2: par 4; 356 yds.; 13 handicap
SampleWood 3: par 4; 406 yds.; 1 handicap
SampleWood 4: par 4; 411 yds.; 3 handicap
SampleWood 5: par 4; 387 yds.; 11 handicap
SampleWood 6: par 3; 180 yds.; 17 handicap
SampleWood 7: par 5; 510 yds.; 5 handicap
SampleWood 8: par 3; 187 yds.; 15 handicap
SampleWood 9: par 5; 485 yds.; 7 handicap
SampleVistas 1: par 3; 327 yds.; 12 handicap
SampleVistas 2: par 4; 420 yds.; 3 handicap
SampleVistas 3: par 3; 280 yds.; 8 handicap
SampleVistas 4: par 5; 502 yds.; 4 handicap

You can orderby the NAME part of your data tags just like you can in most other queries:

```
<mtl biz.feature_row category="golf course hole" business_id=$bus_id
orderby=par,handicap>
     <mtl biz.feature_row.course_name>
     <mtl biz.feature_row.hole_number>:
     par <mtl biz.feature_row.par><br>
     <mtl biz.feature_row.yardage> yds.,
     <mtl biz.feature_row.handicap> handicap<p>
</mtl biz.feature_row>
```

Here's the output:

SampleVistas 1: par 3; 327 yds.; 12 handicap
SampleWood 8: par 3; 187 yds.; 15 handicap
SampleWood 6: par 3; 180 yds.; 17 handicap

SampleVistas 3: par 3; 280 yds.; 8 handicap
SampleWood 1: par 3; 372 yds.; 9 handicap
SampleWood 3: par 4; 406 yds.; 1 handicap
SampleWood 5: par 4; 387 yds.; 11 handicap
SampleWood 2: par 4; 356 yds.; 13 handicap
SampleVistas 2: par 4; 420 yds.; 3 handicap
SampleWood 4: par 4; 411 yds.; 3 handicap
SampleVistas 4: par 5; 502 yds.; 4 handicap
SampleWood 7: par 5; 510 yds.; 5 handicap
SampleWood 9: par 5; 485 yds.; 7 handicap

It's in order by par just fine. But look at what we sub-ordered by, the handicap values. They're not in order … are they?

Yes, they are. They're in ascii-betical order. When you sort alphabetically, any word that starts with "A" is going to come before any word that starts with "B." The same holds true with an ascii-betical sort, but in addition, any set of characters that starts with a "1" is going to come before any set of characters that starts with a "2," or "3." That's why of the par=3 records, the one with handicap=12 comes out before the one where handicap=8.

You and I can look at that and tell that SampleVistas hole 3 should come before hole 1; we see that the number 8 is less than the number 12. But Forge's not looking at those values like numbers; it sees them as words. And there's no way to tell Forge to look at some feature values like numbers and others like words. So it sees all feature values as words.

Take another look at that query, specifically `category="golf course hole."` Usually, I hammer on using variables in your queries for flexibility. So why did I hard-code the category value in the query above? Because you might as well. After all, the data tags correspond directly, and only, to that one category. So in this case I hard-code it and eliminate the possibility of a value other than the one I'm expecting being in my variable.

One more thing. It's tempting to try to cover all the bases, like so:

```
<mtl biz.feature_row category="golf course hole,golf_courses"
business_id=$bus_id>
     <mtl if {biz.feature_row.category_name="golf_courses">
          <mtl biz.feature_row.driving_range>
          <mtl biz.feature_row.chipping_area>
          <mtl biz.feature_row.practice_bunker>
     <mtl else>
          <mtl biz.feature_row.course_name>
          …
```

It's tempting, but it doesn't work. There are a number of reasons why. Here's a big one:

There's nothing that says feature names have to be unique. It's entirely possible to have a feature named "Name" in each feature category.

So let's say we do have two categories, "category1" and "category2" that both have a "name" feature. If we tried to do this:

```
<mtl biz.feature_row category="category1,category2" business_id=$bus_id>
     <mtl biz.feature_row.name>
```

How would Forge know which category's "name" to use? It wouldn't. So when Forge encounters a biz.feature_row query qualified by multiple categories, it just ignores the whole query.

Of course practically speaking, it's just as easy to write a query for each category as it is to try to combine them. Actually, it's probably easier.

Also, it works.

Exercises:

Ex. 1: Pick a business with feature data. Write a level 1, alt-detail template using biz.feature_row to display that data.

Biz.Feature_Category

You can use this query to find out which feature categories that are available to a business actually have values in them.

That may sound a little obvious at first, but consider: feature categories are associated with business types, and business types are collections of SIC's. You make a feature category available to a business by giving it a SIC that's in that business type. But not all the feature categories that apply to a business type will apply to *every* business in that type.

For instance, let's say you're dealing with a donut shop. It'll get lumped with the restaurants. And while you'll want to fill in the features that deal with the health inspection - because they apply - the feature category about reservation information just doesn't. So you'll leave it blank.

Now let's look at the **qualifiers**:

- `business_id=` only categories for this business or businesses will be returned
- `business_type=` only the categories that apply to this business type - for the businesses specified by `business_id=` (above) - will be returned
- `business_type_id=` only the categories that apply to this business type - for the businesses specified - will be returned
- `name=` return only categories with one of these names

Given the fact that you can qualify this query by business_type, it's tempting to think it'll give you a list of the feature categories pertinent to that type. But remember, to get anything back from this query, you must pass in the id of at least one business - preferably one with some feature data - because it's about which feature categories *available* to a business actually have something in them.

Let's go back to our donut shop example. Actually, it was started by a brilliant young entrepreneur who, in a flash of insight, realized that something about waiting to have their oil changed makes people hungry for donuts. Thus Dunk-n-Lube* was born.

Of course, this business falls into two business types: Automotive and Dining. So if you plug just Dunk-n-Lube's business id into the biz.feature_category query, you'll get all the categories that Dunk-n-Lube has info for: price of an oil change, health inspection report &etc.:

```
<mtl biz.feature_category business_id=$bus_id>
     <b><mtl biz.feature_category.name></b><br>
</mtl biz.feature_category>
```

Get's us:

Cuisine
Health_Inspection
Auto_Service

But we don't get listings for the "Reservations" feature category, or for any of the feature categories intended for use with auto dealerships – they don't apply, so they weren't filled in – so they don't show up here.

On the other hand, you won't usually want all of that data at one time. When you include Dunk-n-Lube in a dining guide, you'll just want the donut data. But when it shows up in Car Care, you don't necessarily want to call attention to the crème fills. So qualify your query by `business_type=` or `business_type_id=` and you'll only get feature categories relevant to the specified type (dining or automotive) that Dunk-n-Lube has data in.

Now let's look at the **data tags**:

- id - Unique numeric identifier of a feature category.
- name - Unique name of a feature category.

This is another instance of a data set where both the name and id are unique.

Exercises:

Ex. 1: Write a level 2, alt-detail template that returns a business's non-blank feature categories, given the business id and possibly a business type.

*Okay, okay. This business was made up. But you get the idea.

Biz.Feature_Name

So now you can find a business' categories of features. Let's see what the features themselves are.

We'll use these **qualifiers** to retrieve the feature names in a given category:

- `category_id=` Return the feature names for this category. There may be one or more category ids.
- `category_name=` Return the feature names for this category. There may be one or more category names.

Now that we've got our feature records, here are their **data tags**:

- id - A number that uniquely identifies this feature name. The name is not guaranteed to be unique as 2 different categories can have features with the same name.
- name - the name of this feature

Exercises:

Ex. 1: Add a biz.feature_name query to your level 1, alt-detail template to show the features in each of the categories.

Biz.Feature - Basic

So the feature template we're building has gotten to this point:

```
<mtl biz.feature_category business_id=$bus_id business_type=$type>
     <b><mtl biz.feature_category.name></b><br>
     <mtl biz.feature_name category_name=biz.feature_category.name>
          <mtl biz.feature_name.name><br>
     </mtl biz.feature_name>
     <p>
</mtl biz.feature_category>
```

Here's what it gets us for Samples Country Club:

Golf Course Hole
Hole Number
Par
Yardage
Handicap
Course Name

It tells us everything about the features for our business – everything except the actual feature values.

We can fill in that gap with biz.feature. Biz.feature is a new query, but it does something you've already seen. In *Biz.Feature_Row* you saw feature values retrieved, a row at a time, and learned how to pull values out of those rows using the feature names (`<mcc biz.feautre_row.NAME>`). That query is easy to use, but it has drawbacks: You have to write a separate query for each feature category you want values from, and you have to hard-code the features you want from that category.

Biz.feature retrieves those same values, but lets you query as many categories as you want. Once it gets them, it pulls the results back, not a row at a time - as you saw with biz.feature_row - but a value (cell in the table) at a time. Biz.feature is more flexible than biz.feature_row (you could let the user pick which categories to get values from, instead of deciding for her ahead of time) but slightly harder to use.

We'll start with these **qualifiers**:

- `name=` only return values for features with this name
- `id=` only return values for this specific feature
- business_id= only features for this business or businesses will be returned

And here are the **data tags** we're interested in at this point:

- name - the name of the feature for this feature value.
- value - value stored at this feature row and column.

Of course, we've already got the feature name from our biz.feature_name query, but it's nice to know we can get it from here, too.

So let's plug the new query into our template:

```
<mtl biz.feature_category business_id=$bus_id business_type=$type>
     <b><mtl biz.feature_category.name></b><br>
     <mtl biz.feature_name category_name=biz.feature_category.name>
          <mtl biz.feature_name.name>:<br>

          <mtl biz.feature business_id=$bus_id
          name=biz.feature_name.name>
               <mtl biz.feature.value><br>
          </mtl biz.feature>

     </mtl biz.feature_name>
     <p>
</mtl biz.feature_category>
```

Here's a sample of what we get back:

Golf Course Hole
hole number:
1
2
3
4
5
6
7
8
9
1
2
3
4
par:
3
4
4
4
4
…

It's unwieldy, yes. But at least we can get to the values.

Biz.Feature - Advanced

So far we've looked at the path of least resistance to pulling out a business' feature values. In this chapter we'll talk about how feature values are actually stored in the database, and examine more flexible, but more complicated, methods of pulling them back out. For 98% of the cases, what we've covered so far will meet your needs. What follows is for the other two percent.

We'll start with a simple query that brings back everything - but just for the business(es) specified by $bus_id:

```
<mtl biz.feature business_id=$bus_id orderby=category_name>
     <mtl biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

The `<mtl biz.feature.name>` tag pulls back the name of the column. The `<mtl biz.feature.value>` tag gets us the value in the column - for the row it's on. Here are the first few lines of the output:

```
hole number: 1
Average Meal Cost: $20+
Photo: http://yahoo.com/images/boo.gif
par: 3
Width:
yardage: 372
Height:
Handicap: 9
Caption:
Paragraph:
Course Name: SampleWood
hole number: 2
par: 4
yardage: 356
Handicap: 13
Course Name: SampleWood
hole number: 3
par: 4
yardage: 406
Handicap: 1
Course Name: SampleWood
hole number: 4
…
```

Notice that it comes out in no particular order. Notice also that we get *every* feature our business has, including information about the clubhouse restaurant.

The moral is that you *can* use this query to retrieve every feature applied to a business, but you might not always want to do that. Especially since it's entirely possible to have not-for-public-consumption administrative features. (We'll go into more on that later.)

Right now, for instance, we're only interested in our Golf Course Hole data so we add the category_name qualifier:

```
<mtl biz.feature business_id=$bus_id category_name=$cat>
     <mtl biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

Here's a sample of what we get back:

hole number: 1
par: 3
yardage: 372
Handicap: 9
Course Name: SampleWood
hole number: 2
par: 4
yardage: 356
Handicap: 13
Course Name: SampleWood
…
hole number: 1
par: 3
yardage: 327
Handicap: 12
Course Name: SampleVistas
Hole number: 2
par: 4
yardage: 420
Handicap: 3
…

It wouldn't take much to dress this up and make it presentable, but there are a few things to be aware of. First, if you don't specify an orderby, Forge will start at the top left corner of your feature category (if you specified one - remember when we didn't, our records came out in arbitrary order) and spit out records working left to right, top to bottom like we have above.

But if you do specify an orderby, it gets a little more interesting. For instance, when I do this:

```
<mtl biz.feature business_id=$bus_id CATEGORY_NAME=$cat
orderby=name(course name,hole number)>
     <mtl biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

I get this back:

Course Name: SampleWood
Course Name: SampleWood
Course Name: SampleWood
Course Name: SampleVistas
Course Name: SampleWood

Course Name: SampleVistas
Course Name: SampleWood
Course Name: SampleVistas
Course Name: SampleWood
Course Name: SampleWood
Course Name: SampleWood
Course Name: SampleWood
Course Name: SampleVistas
hole number: 5
hole number: 3
hole number: 1
hole number: 3
hole number: 9
hole number: 4
hole number: 4
hole number: 2
hole number: 7
hole number: 1
hole number: 6
hole number: 2
hole number: 8
yardage: 280
Handicap: 9
yardage: 411
yardage: 180
Handicap: 3
Handicap: 17
Handicap: 7
…

Now, why in the world did it do that? Because there's a record in the database for each cell in our "Golf Course Hole" feature category.

Confused? Let's plug in some more data tags and see what we get:

```
<mtl biz.feature business_id=$bus_id CATEGORY_NAME=$cat>
     <mtl biz.feature.index>. row <mtl biz.feature.row_number>.
     <mtl biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

The output looks like this:

1. row 0. hole number: 1
2. row 0. par: 3
3. row 0. yardage: 372
4. row 0. Handicap: 9
5. row 0. Course Name: SampleWood
6. row 1. hole number: 2
7. row 1. par: 4
8. row 1. yardage: 356
9. row 1. Handicap: 13
10. row 1. Course Name: SampleWood
11. row 2. hole number: 3
12. row 2. par: 4

…

We dropped the orderby, so the data comes back in the original order. Notice that each cell in our "Golf Course Hole" feature category gets a new index number - because each cell is actually its own record.

In biz.feature_row you saw features retrieved a row at a time, and you learned how to ask for the value of a specific column in a row (`<mtl biz.feature_row.NAME>.`)

So when we used biz.feature_row to get Golf Course Hole features for the Samples Country Club, Forge acted like it found 13 records, and we used the feature NAMEs to get the values.

But actually, Forge found 65 values - 13 rows of five values each. It just grouped them by row for your convenience. When you use biz.feature, you have to do that row grouping manually if you want your records returned that way.

But why do the row numbers start at 0? Well, it's because that's where computers (and the engineers) start counting.

So back to the original question: why was the data so jumbled when we added this to the query: `orderby=name(course name,hole number)`?

Hopefully the answer is beginning to dawn on you: that orderby tells Forge to spit out first all the records where the name is "course name," followed by all the records where the name is "hole number." All the rest of the records (one for each individual cell in our feature category) tumble out afterward in arbitrary order.

So maybe now you understand the problem, but what's the solution?

Well, think about it. What were we expecting?

I don't know about you, but I was expecting Forge to keep the cells (records) in a row together, but to pull out Course Name and hole number first. So maybe we need to add row_number to the orderby:

```
<mtl biz.feature business_id=$bus_id CATEGORY_NAME=$cat
orderby=name(course name,hole number),row_number>
     <mtl biz.feature.index>. row <mtl biz.feature.row_number>. <mtl
biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

Closer, but still not quite there:

1. row 0. Course Name: SampleWood
2. row 1. Course Name: SampleWood
3. row 2. Course Name: SampleWood
4. row 3. Course Name: SampleWood
5. row 4. Course Name: SampleWood
6. row 5. Course Name: SampleWood
7. row 6. Course Name: SampleWood
8. row 7. Course Name: SampleWood
9. row 8. Course Name: SampleWood
10. row 9. Course Name: SampleVistas

11. row 10. Course Name: SampleVistas
12. row 11. Course Name: SampleVistas
13. row 12. Course Name: SampleVistas
14. row 0. hole number: 1
15. row 1. hole number: 2
16. row 2. hole number: 3
17. row 3. hole number: 4
18. row 4. hole number: 5
19. row 5. hole number: 6
20. row 6. hole number: 7
21. row 7. hole number: 8
22. row 8. hole number: 9
23. row 9. hole number: 1
24. row 10. hole number: 2
25. row 11. hole number: 3
26. row 12. hole number: 4
27. row 0. yardage: 372
28. row 0. par: 3
29. row 0. Handicap: 9
30. row 1. par: 4

It still pulled all the "Course Name" cells out first, but at least it put them in order by the row they came from.

So let's try putting `row_number` first in the `orderby=` so Forge will put all the "row 0" cells together, then subsort by name:

```
<mtl biz.feature business_id=$bus_id CATEGORY_NAME=$cat
orderby=row_number,name(course name,hole number)>
     <mtl biz.feature.index>. row <mtl biz.feature.row_number>. <mtl
biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

This is more like it:

1. row 0. Course Name: SampleWood
2. row 0. hole number: 1
3. row 0. Handicap: 9
4. row 0. par: 3
5. row 0. yardage: 372
6. row 1. Course Name: SampleWood
7. row 1. hole number: 2
8. row 1. par: 4
9. row 1. yardage: 356
10. row 1. Handicap: 13
11. row 2. Course Name: SampleWood
12. row 2. hole number: 3
...

Notice, the names we didn't list in the orderby still come out in arbitrary order, but it's fairly obvious now what to do about that.

Biz.Feature **qualifiers** of note:

- `business_id=` only features for this business or businesses will be returned
- `category_id=` only return feature values for this category
- `category_name=` only return feature values for this category
- `city_id=` Only features for businesses in this city will be returned. Remember, a given city may have more than one id. For more on this, see the `city_id=` discussion in "Biz.Business - geographic selection" above.
- `city_name=` only features for businesses in this city will be returned. Be aware that city names are not usually unique in the world.
- `market_id=` only features for businesses in this market will be returned
- `market_name=` Only features for businesses in this market will be returned
- `name=` only return values for features with this name.
- `id=` Only return values for this specific feature. Note though, that this id does *not* correspond to a cell in the table, but to a column. This is what you'd use to, for instance, get just the hole handicaps.

The `id=` qualifier in this query is a little different from most queries. Usually you use `id=` to get one specific record. This time you'll use it to get a column of records.

Let's just hammer that home:

```
<mtl biz.feature business_id=$bus_id CATEGORY_NAME=$cat id=$id>
     <mtl biz.feature.index>. id: <mtl biz.feature.id>
     <mtl biz.feature.name>: <mtl biz.feature.value><br>
</mtl biz.feature>
```

We've gone back to the original, unordered, query but added data tags. Let's look at the output:

1. id: 1429638 hole number: 1
2. id: 1429639 par: 3
3. id: 1429640 yardage: 372
4. id: 1429641 Handicap: 9
5. id: 1429642 Course Name: SampleWood
6. id: 1429638 hole number: 2
7. id: 1429639 par: 4
8. id: 1429640 yardage: 356
9. id: 1429641 Handicap: 13
10. id: 1429642 Course Name: SampleWood
11. id: 1429638 hole number: 3
12. id: 1429639 par: 4
13. id: 1429640 yardage: 406
14. id: 1429641 Handicap: 1
15. id: 1429642 Course Name: SampleWood

...

Notice that the values of the id tag repeat. And that they repeat pretty regularly. In fact, every time the `<mtl biz.feature.name>` is "hole number" the `<mtl biz.feature.id>` is 1429638. Same thing with par, number 149639; yardage, 1429639 &etc.

Biz.Business - feature-based selection

So now you know what features are, where they come from and how to pull them back out. But how do you pick businesses based on their features?

For that matter, why would you want to?

Let's say you're doing dining guide and want to be able to list restaurants by cuisine.

Assuming you have a feature category set up for this, and assuming you've done the data entry for it, you could use these **qualifiers** to make sure your results list contained only restaurants with the appropriate food type:

- feature_value(arg)= Return only businesses that have this feature set to this value, arg may be a feature id or a feature name in the form "category:name." The inequality operator (!=) does not work with this qualifier.
- feature_contains(arg)= Same as feature_value except that a wildcard search is used. The inequality operator (!=) does not work with this qualifier.
- feature_exists(arg)= Return only businesses that have this feature set to some value. Value on right side of equals must be "yes" or "no". The inequality operator (!=) does not work with this qualifier.

Before we start querying, let's set out the feature category we'll be working with. We'll name it "Restaurant" and give it these features: cuisine, average dinner cost, average lunch cost, reservations, style.

Now, to use it:

```
<mtl biz.business feature_value(restaurant:cuisine)=$cuisine>
     <mtl biz.business.name>
</mtl biz.business>
```

Notice what's in the parentheses: restaurant:cuisine. This translates to [category name]:[feature name]. Why do you have to specify both feature category name and feature name? You use the unique category name because more than one category could have a feature named "cuisine." Similarly, you specify the feature you want because categories can have more than one feature.

Alternately, you could put a feature id in the parentheses [i.e. `feature_value(1234)=$cuisine`] and be done with it.

We've hard-coded the [category name]:[feature name] combo in the template above, but you could easily use a variable for it - as long as the value is a feature id or follows the [category name]:[feature name] pattern. Like this, for example:

```
<mtl biz.business feature_value($category:$feature)=$cuisine>
```

So now, we can put "cuisine=Italian" or "Chinese" or "Mexican" in a URL to this template and get a list of restaurants that serve only that cuisine.

Also, you can use as many feature_value(arg)= qualifiers as you like in a query, so if you're looking for a family-style Chinese restaurant, you can easily do this:

```
<mtl biz.business feature_value(restaurant:cusine)=$cuisine
feature_value(restaurant:style)=$style>
```

But what about restaurants with hyphenated cuisine, like Greek-Italian or Chinese-Thai? Shouldn't a Greek-Italian restaurant come up when you search for "Greek"?

Of course it should. And that's where the `feature_contains(arg)=` qualifier comes in:

```
<mtl biz.business feature_contains(restaurant:cuisine)=$cuisine>
     <mtl biz.business.name>
</mtl biz.business>
```

This is pretty much the same template as last time, but we've changed "feature_value" to "feature_contains." So this time when we put "cuisine=italian" in the URL, we'll get all the straight Italian restaurants, the Greek-Italian restaurants, the Italian-Greek, Italian-American and so on. I.e. all the restaurants where the Restaurant:Cuisine value *contains* "Italian."

The last feature qualifier in biz.business is `feature_exists(arg)=`. It takes the same `[category name]:[feature name]` style argument, but there are only two legal values for the right side of the equals: "yes" and "no."

As with `feature_value(arg)=` you can have as many `feature_contains(arg)=` and `feature_exists(arg)=` qualifiers in your query as you like.

There aren't any biz.business **data tags** for features. You'll need to use one of the feature-specific queries if you'd like to display the feature values.

Biz.Business - the rest of the story

Before we wrap up, there are a few more query **qualifiers** to touch on:

- `keyword_query=` Return businesses whose name, description, short_description or keywords feature contains *all* the words specified in this text string.

  The words don't have to appear in the business listing in the same order as they do in the `keyword_query=` value, but they *do* all have to appear in the same part of the listing. I.e. if the `keyword_query=` value has two words, a business with one of the words in its name and the other in its description does *not* match the keyword query.

- `event_id=` Only businesses for this event will be returned. This qualifier ties businesses to events stored in mdCalendar (the cal.event* tags.) Typically, this will be used to get venue information for an event.

  For instance, let's say you've got an mdCalendar listing for an art show at "The Broad Street Gallery." Because there's already a business listing for the gallery in the business database, there's no need to re-type the phone, address and hours of the gallery. Instead, you can associate the event with its venue (through the mdCalendar client, TimeLink) then use biz.business, qualified by `event_id=`, to get the venue information.

- type= Only businesses of this type will be returned. NOTE: This is different from BUSINESS_TYPE and SIC. Type is an internal marker - a way to isolate certain records in the database. For instance, calendars are stored in the business database, but with type set to calendar.

  You can't set type through BizLink, nor can you get to businesses where type is set (i.e. non-null) through BizLink. Typically, type is set by an automated data loader or some other client (like TimeLink.) You can, however, use this qualifier to get a list of records in the business database of a given type.

- status= Only businesses with this status are returned. Typical statuses are "active," "inactive," and "" (i.e. null.) May be multiple values.

- section_id= Only returns businesses that are in the given mdDynaPub section. This qualifier ties businesses to mdDynaPub. It's relevant for businesses with associated documents (such as press releases) stored in mdDynaPub.
- section_type= Only the business in a section of the specified type will appear. Section_type represents an aggregation of sections. Like the section_id= qualifier, this qualifier ties businesses to related sections of documents in mdDynaPub.

Appendix A: Exercise Answers

Biz.Business_Type

Ex. 1:

Template properties:

```
status=active       level=1
type=brg            default=true
sub-type=browse
```

Template body:

```
<mtl biz.business_type>
     <mtl biz.business_type.index>. <mtl biz.business_type.id> <mtl
biz.business_type.name><br>
</mtl biz.business_type>
```

Biz.SIC

Ex. 1:

Template properties:

```
status=active       level=2
type=brg            default=true
sub-type=browse
```

Template body:

```
<mtl if {$type=''}>
     <mtl $type='dining'>
</mtl if>

<mtl biz.sic business_type=$type>
     <mtl biz.sic.sic> <mtl biz.sic.title><br>
</mtl biz.sic>
```

Ex. 2:

Template body: (copied from above. New code in italics)

```
<mtl biz.business_type>
     <mtl biz.business_type.index>. <a href="/forge?classification=<mtl
$classification fmt=url>&temp_type=<mtl $temp_type>&tl=2&type=<mtl
biz.business_type.name fmt=url>"><mtl biz.business_type.name></a><br>
</mtl biz.business_type>
```

Note the use of variables for temp_type and classification in constructing the link to the next template. Since you presumably used a "long style" URL to get to this template in the first

place, `<mtl $classification>` is defined in the URL. The temp_type variable will only work if [1]both your templates have the same sub-type, and [2]you've used the "long style" URL with &temp_type= in it.

(Okay, yes, you could set those variables at the top of your template if they're blank, but you *should* be passing them through your URL anyway.)

Biz.Market

Ex. 1:

Template properties:

```
status=active       level=3
type=brg            default=true
sub-type=browse
```

Template body:

```
<mtl biz.market>
     <mtl biz.market.index>. <mtl biz.market.id> <mtl
biz.market.name><br>
</mtl biz.market>
```

Biz.City

Ex. 1:

Template properties:

```
status=active       level=4
type=brg            default=true
sub-type=browse
```

Template body:

```
<mtl if {$market=''}>
     <mtl $market='Jacksonville'>
</mtl if>

<mtl biz.city market_name=$market>
     <mtl biz.city.id> <mtl biz.city.name><br>
</mtl biz.city>
```

Ex. 2:

Template body: (copied from above. New code in italics)

```
<mtl biz.market>
```

```
     <mtl biz.market.index>. <a href="/forge?classification=<mtl
$classification fmt=url>&temp_type=<mtl $temp_type>&tl=<mtl
{$tl+1}>&market=<mtl biz.market.name fmt=url>"><mtl
biz.market.name></a><br>
</mtl biz.market>
```

Note the use of the `<mtl {$tl+1}>` in the construction of the URL to the next template. It takes the level of the current template, as provided in the URL, and adds one to it to derive the level of the next template. This will only work if [1]your template levels are numbered sequentially and [2]your (long-style) URL indicates the level of the current template.

(Okay, yes, you *could* set that variable at the top of your template if it's blank, but you *should* be passing that info through the URL to start with.)

Biz.Business - geographic selection

Ex. 1:

Template properties:

```
status=active       level=1
type=brg            default=true
sub-type=detail
```

Template body:

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
<mtl biz.business market=$market city=$city state_abbr=$state zip=$zip
orderby=name>
     <mtl biz.business.name><br>
     <mtl biz.business.street_city><p>
</mtl biz.business>
```

Ex. 2:

Template body: (copied from above. New code in italics)

```
<mtl if {$market=''}>
     <mtl $market='Jacksonville'>
</mtl if>

<mtl biz.city market_name=$market orderby=name>
     <a href="/forge?classification=<mtl $classification
     fmt=url>&temp_type=detail&city=<mtl biz.city.name
     fmt=url>&state=<mtl biz.city.state_abbr>"><mtl biz.city.name>, <mtl
     biz.city.state_abbr></a><br>
</mtl biz.city>
```

Biz.Business - topical selection

Ex. 1:

Template body: (copied from above. New code in italics)

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
<mtl biz.business business_type=$type sic=$sic market=$market city=$city
state_abbr=$state zip=$zip orderby=name>
     <mtl biz.business.name><br>
     <mtl biz.business.street_city><br>
     phone: <mtl biz.business.phone><br>
     fax: <mtl biz.business.fax><br>
     <a href=mailto:<mtl biz.business.email>><mtl biz.business.email></a>
     <p>
</mtl biz.business next=name>
```

Ex. 2:

Template body: (copied from above. New code in italics)

```
<mtl if {$type=''}>
     <mtl $type='dining'>
</mtl if>

<mtl biz.sic business_type=$type>
     <a href="/forge?classification=<mtl $classification
     fmt=url>&temp_type=detail&sic=<mtl biz.sic.sic>"><mtl
     biz.sic.title></a><br>
</mtl biz.sic>
```

Specific businesses

Ex. 1:

Template properties:

```
status=active       level=1
type=brg            default=true
sub-type=search
```

Template body:

```
<form action="/forge">
     <input type=hidden name=classification value="$classification">
     <input type=hidden name=temp_type value=detail>


     Business name: <input type=text name=namecontains><p>

     Area:
     <select name="market">
       <option value=""></option>
```

```
     <mtl biz.market orderby=name>
       <option value="<mtl biz.market.name>"><mtl
       biz.market.name></option>
     </mtl biz.market>
     </select><p>

     Industry:
     <select name="type">
       <option value=""></option>
     <mtl biz.business_type orderby=name>
       <option value="<mtl biz.business_type.name>"><mtl
       biz.business_type.name></option>
     </mtl biz.business_type>
     </select><p>

     <input type=submit value="Find it!">
</form>
```

Body of business list template (copied from above. New code in italics)

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
<mtl biz.business name_contains=$namecontains business_type=$type
sic=$sic market=$market city=$city state_abbr=$state zip=$zip
orderby=name>
     <mtl biz.business.name><br>
     <mtl biz.business.street_city><p>
</mtl biz.business next=name>
```

Ex. 2:

Template properties:

```
status=active       level=2
type=brg            default=true
sub-type=detail
```

Template body:

```
<mtl biz.business name=$busname market=$market city=$city state=$state
zip=$zip orderby=street1>
     <mtl if {biz.business.index=1}>
          <b><mtl biz.business.name></b><p>
          <ul>
     </mtl if>

     <li><mtl biz.business.street_city>, <mtl biz.business.state_abbr><p>

     <mtl biz.business.description><p>

     <a href="<mtl biz.business.url>"><mtl biz.business.url></a><br>
     <a href=mailto:<mtl biz.business.email>><mtl biz.business.email></a>
     phone: <mtl biz.business.phone><br>
```

```
     fax: <mtl biz.business.fax><p>

     <mtl if {biz.business.index=biz.business.num_rows}>
          </ul>
     </mtl if>
</mtl biz.business>
```

Body of business list template (copied from above. New code in italics)

```
<mtl if {$market=''}>
     <mtl $market='Augusta'>
</mtl if>
<mtl biz.business name_contains=$namecontains business_type=$type
sic=$sic market=$market city=$city state_abbr=$state zip=$zip
orderby=name>
     <a href="/forge?classification=<mtl $classification
     fmt=url>&temp_type=<mtl $temp_type>&tl=2&market=<mtl $market
     fmt=url>&city=<mtl $city fmt=url>&state=<mtl $state>&zip=<mtl
     $zip>&busname=<mtl biz.business.name fmt=url>"><mtl
     biz.business.name></a><br>
     <mtl biz.business.street_city><p>
</mtl biz.business next=name>
```

Biz.Feature_Row

Ex. 1:

Template properties:

```
status=active       level=1
type=brg            default=true
sub-type=alt_detail
```

Template body: Consult examples in chapter and your Support rep. for help.

Biz.Feature_Category

Ex. 1:

Template properties:

```
status=active       level=2
type=brg            default=true
sub-type=alt_detail
```

Template body:

```
<mtl biz.feature_category business_id=$bus_id business_type=$type>
     <mtl biz.feature_category.name><br>
</mtl biz.feature_category>
```

Biz.Feature_Name

Ex. 1: Code copied from Biz.Feature_Category, Ex. 1 above. New code in italics

Template body:

```
<mtl biz.feature_category business_id=$bus_id business_type=$type>
     <b><mtl biz.feature_category.name></b><br>
     <mtl biz.feature_name category_name=biz.feature_category.name>
          <mtl biz.feature_name.name><br>
     </mtl biz.feature_name>
     <p>
</mtl biz.feature_category>
```