Getting Started With BumbleBee

Contents

Introduction

About BumbleBee

BumbleBee is an automated test harness for evaluating XQuery engines and validating queries expressed in the XQuery language. BumbleBee takes the pain and uncertainty out of learning and using XQuery. It starts by letting you immediately put several XQuery engines to the test so you know how they stack up against the XQuery specification. Then it lets you easily write your own tests to continually make sure your XQuery expressions produce reliable results when you upgrade your XQuery engine, try different engines, or otherwise make changes to your queries.

The BumbleBee distribution contains a comprehensive suite of tests based on the XQuery specification. It also contains a set of adapters to several popular free and commercial XQuery engines. So, out of the box, BumbleBee will run a healthy dose of compatibility tests against any supported XQuery engine.

The compatibility tests distributed with BumbleBee are just the beginning. Once you've chosen an XQuery engine, you can write your own BumbleBee tests to validate the expected results of your own XQuery expressions.

About This Guide

The goal of this Getting Started guide is to get you started quickly. In less than 5 minutes you should be running a healthy dose of compatibility tests against several XQuery engines distributed with BumbleBee.

This guide covers how to install, run, and configure BumbleBee, write BumbleBee tests, and extend BumbleBee. The latest version of this guide can always be found at http://xquery.com/bumblebee, which also hosts the latest version of BumbleBee and related BumbleBee resources.

Getting Started

Installing a JRE

BumbleBee requires a Java Runtime Environment (JRE) of version 1.4 or higher. You can verify whether you have a compatible JRE version by typing:

```
> java -version
```

If the output doesn't indicate that version 1.4 or higher is installed, you can download the JRE from http://java.sun.com.

Installing BumbleBee

Once you have installed a Java Runtime Environment (JRE) of version 1.4 or higher, install BumbleBee according to the following steps:

1. Download the latest version of BumbleBee from http://xquery.com/bumblebee.

2. Unzip the downloaded BumbleBee distribution file (e.g. bumblebee-1.0.zip) to a directory of your choice referred to hereafter as `$BUMBLEBEE_HOME`. Unzipping the file will automatically create a `bumblebee-<version>` directory.

3. Set the `JAVA_HOME` environment variable to the top-level directory of your Java Runtime Environment (JRE) installation. For example:

   Windows

   ```
   C:\> set JAVA_HOME=C:\java1.4
   ```

   Unix

   ```
   bash% JAVA_HOME=/usr/java1.4; export JAVA_HOME
   tcsh% setenv JAVA_HOME /usr/java1.4
   ```

The BumbleBee Directory Structure

Now that you've installed BumbleBee, the `$BUMBLEBEE_HOME` directory should contain the following directories and their contents:

Directory	Contents
`doc`	Documentation, including this guide
`conf`	`bumblebee.properties` configuration file
`log`	Default log file directory
`lib`	Core JAR files required to run BumbleBee
`lib/ext`	External JAR files required to test XQuery engines
`tests`	BumbleBee test (`.bee`) file directories
`tests/`*`YYYY-MM`*	Directory holding tests written against the given XQuery Working Draft version. For example, 2003-05 corresponds to the May 2003 draft.
`tests/`*`YYYY-MM`*`/custom`	Custom (user-contributed) test files
`tests/`*`YYYY-MM`*`/docs`	Summaries of XQuery engine test results
`tests/`*`YYYY-MM`*`/examples`	Example test files, including this guide's example
`tests/`*`YYYY-MM`*`/nist`	NIST XQuery Test Suite test files
`tests/`*`YYYY-MM`*`/usecases`	W3C Use Cases test files

Running BumbleBee

Now we're ready to put BumbleBee to work buzzing around some tests. Make sure your current working directory is the BumbleBee installation directory (`$BUMBLEBEE_HOME`), then type:

Windows

```
bumblebee.bat
```

Unix

```
./bumblebee.sh
```

By default, without specifying any command-line tests, BumbleBee will run a suite of default tests located beneath the `tests` directory. As each test is run, its name and pass or fail status is printed to the console. For example:

```
Passed   -> Test (Engine 1): Test 1 in 0.415 sec

Failed!  -> Test (Engine 1): Test 2
Expected attribute value '1992' but was '1994' - comparing
<book year="1992"...> at /BumbleBee_Result[1]/bib[1]/book[1]/@year to
<book year="1994"...> at /BumbleBee_Result[1]/bib[1]/book[1]/@year
```

When all the tests have run to completion, you'll see a summary of test results printed to the console for each XQuery engine that was tested. For example:

```
Time: 39.598 seconds

FAILURES!!!
Engine 1: Tests Run: 72, Failures: 22, Disabled: 0 (69.4% passed)
Engine 2: Tests Run: 72, Failures: 1, Disabled: 0 (98.6% passed)
Total   : Tests Run: 144, Failures: 23, Disabled: 0 (84% passed)


(See log/bumblebee.log for failure details.)
```

The test results are also printed in detail to the `log/bumblebee.log` file.

Running Specific BumbleBee Tests

To run BumbleBee against a specific directory containing BumbleBee `.bee` files, or a set of test directories, use the form:

```
bumblebee directory1 [directory2 [...]]
```

For example, to run the August 2003 Use Case tests distributed with BumbleBee and your custom tests contained in the `tests/mytests` directory on Unix, type:

```
./bumblebee.sh tests/2003-08/usecases tests/mytests
```

Depending on your server's compliance level, you may want to use the May 2003 tests instead.

Analyzing The BumbleBee Test Report

After running BumbleBee, the `log/bumblebee.log` file contains a comprehensive report of all tests run. For each failed test, the test report includes the XQuery expression that was run, the actual query result returned by the XQuery engine under test, the query result that was expected by the test, and the failure message. The following is an example test failure:

```
Failed!  -> Test (Engine 1): Test 1

Query:
<bib>
  {
    for $b in doc("tests/2003-05/usecases/xmp/bib.xml")//book
    where $b/publisher = "Addison-Wesley" and $b/@year > 1991
    order by $b/title
    return
        <book>
            { $b/@year }
            { $b/title }
        </book>
  }
</bib>

Actual Result:
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
    </book>
    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
    </book>
</bib>

Expected Result:
<bib>
    <book year="1992">
        <title>Advanced Programming in the Unix environment</title>
    </book>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
    </book>
</bib>

Failure Message:
Expected attribute value '1992' but was '1994' - comparing
<book year= "1992"...> at /BumbleBee_Result[1]/bib[1]/book[1]/@year to
<book year="1994"...> at /BumbleBee_Result[1]/bib[1]/book[1]/@year
```

Notice that the failure message specifies the exact location where the actual and expected query results differ, using an XPath expression.

Now that we've installed BumbleBee and run the default tests, let's write a custom BumbleBee test.

Writing BumbleBee Tests

Your First BumbleBee Test

The best way to learn how to write a BumbleBee test is by getting our hands dirty with an example. The `tests` directory contains numerous tests to learn from, but let's create a BumbleBee test from scratch.

Let's assume we want to write a query against the following XML file, named `tunes.xml`, representing a collection of songs:

```
<Tunes>
  <Tracks>
    <Track>
      <Name>Ready, Steady, Go</Name>
      <Artist>Paul Oakenfold</Artist>
      <Album>Bunkka</Album>
      <Genre>Electronic</Genre>
      <MyRating>10</MyRating>
      <Time>254</Time>
    </Track>
    <Track>
      <Name>Battle</Name>
      <Artist>Hans Zimmer and Lisa Gerrard</Artist>
      <Album>Gladiator Soundtrack</Album>
      <Genre>Instrumental</Genre>
      <MyRating>8</MyRating>
      <Time>193</Time>
    </Track>
    <Track>
      <Name>Orange Wedge</Name>
      <Artist>The Chemical Brothers</Artist>
      <Album>Surrender</Album>
      <Genre>Electronic</Genre>
      <MyRating>7</MyRating>
      <Time>254</Time>
    </Track>
  </Tracks>
</Tunes>
```

We place this XML file in the `tests/2003-05/examples` directory.

We want the query to generate a new XML document representing a playlist of our favorite songs sorted by song name. Using any text editor, we create the following BumbleBee test file named `MyFirstTest.bee` in the `tests/2003-05/examples` directory:

```
!name My First Test

!load tests/2003-05/examples/tunes.xml

!query
<Playlist>
  {
    for $t in doc("tests/2003-05/examples/tunes.xml")//Track
    where $t/Genre = "Electronic" and $t/MyRating > 5
    order by $t/Name
    return
        <Track>
          { $t/Name, $t/Artist, $t/Genre, $t/MyRating }
        </Track>
  }
</Playlist>
!end

!result
<Playlist>
  <Track>
    <Name>Orange Wedge</Name>
    <Artist>The Chemical Brothers</Artist>
    <Genre>Electronic</Genre>
    <MyRating>7</MyRating>
  </Track>
  <Track>
    <Name>Ready, Steady, Go</Name>
    <Artist>Paul Oakenfold</Artist>
    <Genre>Electronic</Genre>
    <MyRating>10</MyRating>
  </Track>
</Playlist>
!end
```

Test directives begin with an exclamation (`!`) symbol. In this example all the test directives have been highlighted in a bold font. We'll step through each directive in detail.

The first test directive is the **`!name`** directive. This directive specifies an arbitrary name for the test. The test name is used in the console and file output to uniquely identify the test. The second test directive is **`!load`**. This requests the server load the file at the given path under the same URI as the path. On some servers this isn't technically necessary but it's always good to have.

Next we find the **`!query`** directive. This directive contains the text of the XQuery expression to be run. The **`!query`** directive must end with a single line containing the **`!end`** directive. Notice that our XQuery expression uses a relative path to our input XML named `tunes.xml` which we placed in the `tests/2003-05/examples` directory. The path is relative to the `$BUMBLEBEE_HOME` directory – the directory from which we run BumbleBee.

The last test directive is the **`!result`** directive. This directive contains the text of the result we expect to be produced by the XQuery expression specified in the **`!query`** directive. The **`!result`** directive must end with a single line containing the **`!end`** directive. If desired, you can add additional test cases within the same `.bee` file; just start the additional tests with a new **`!name`** directive.

That's all there is to it! We simply specify the XQuery expression to be run and the result we expect. Now let's run our test to let it check our expectation.

Since we placed our `MyFirstTest.bee` file in the `tests/2003-05/examples` directory, we can run our test individually by typing:

```
bumblebee tests/2003-05/examples/MyFirstTest.bee
```

Alternatively, if we had multiple test files in our `examples` directory, we could run them all in one fell swoop using:

```
bumblebee tests/2003-05/examples
```

In either case, assuming we only have one test file (`MyFirstTest.bee`) in the `examples` directory, we should see the following console output:

```
BumbleBee: The XQuery Test Harness

Test script: bumblebee/tests/2003-05/examples/MyFirstTest.bee

Passed   -> Test (Qizx): My First Test in 2.902 sec

Test script: bumblebee/tests/2003-05/examples/MyFirstTest.bee

Passed   -> Test (Saxon): My First Test in 1.285 sec

Time: 4.394 seconds

OK!
Qizx : Tests Run: 1, Failures: 0, Disabled: 0 (100% passed)
Saxon: Tests Run: 1, Failures: 0, Disabled: 0 (100% passed)
Total: Tests Run: 2, Failures: 0, Disabled: 0 (100% passed)
```

Notice that in this example our test was run against two XQuery engines: Qizx and Saxon. The test passed in both cases. That is, as a result of running our test through BumbleBee we know that our XQuery expression produces the expected result when run against either of these engines. The XQuery engines that are used by default are based on the current configuration of BumbleBee, which we'll discuss in a later section.

Writing a Negative Test

Our first BumbleBee test was a positive test. It passed only when the XQuery engine under test produced the expected result and not an error condition. Sometimes you want to test that an XQuery engine produces an error when an error is appropriate. The following example ensures the engine reports a divide by zero error:

```
!name A Negative Test

!query
3 idiv 0 = 1
!end

!result
ERROR
!end
```

The **`!result`** directive uses a special **`ERROR`** keyword to indicate that any error reported by the XQuery engine is another permittable result. Any non-error condition is a failure.

Writing a Compound Test

BumbleBee allows you to specify multiple results for a single query. Why would you want this? For one, because in XQuery an expression can be evaluated in any order and some orderings may short circuit to success while others may legitimately return errors. The following BumbleBee test demonstrates:

```
!name A Compound Test

!query
1 eq 2 and 3 idiv 0 = 1
!end

!result
false
!end

!result
ERROR
!end
```

Notice the use of two `!result` directives. The first `!result` directive indicates that `false` may be returned (if the expression is evaluated left to right and it short circuits). The second `!result` directive allows an error as another legal possibility (if the expression is evaluated right to left). To summarize, if either `false` or an error condition is returned by the XQuery engine, then the test will pass. An arbitrary number of possible results can be declared in any BumbleBee test using this format.

Test Syntax

The test syntax is intended to be easy to learn and difficult to get wrong. When BumbleBee parses a test, it only attempts to interpret lines that begin with a test directive it understands. When it doubt, it simply ignores a line.

All BumbleBee test files must have a `.bee` extension (e.g `MyFirstTest.bee`). A test file may contain one or more tests. Each test in a test file must contain all of the following required test directives, and may contain one or more of the following optional test directives. Test directives may not be nested.

Directive:	`!name`
Example:	`!name My First Test`
Description:	Specifies a unique name for the test
Use:	Required
Scope:	Single line

Directive:	`!query`

Example:	`!query`
	`  <XQuery expression text>`
	`!end`

Description:	Specifies the XQuery expression to be run
Use:	Required
Scope:	Until a matching `!end` is encountered

Directive:	`!result`

Example:	`!result`
	`  <expected result text>`
	`!end`

Description:	Specifies the expected result of the XQuery expression
Use:	Required
Scope:	Until a matching `!end` is encountered
Notes:	Multiple `!result` sections may be specified for any test. The test passes if any expected result matches the actual result. Using the special `ERROR` keyword as the expected result text indicates that the XQuery engine should produce an error.

Directive:	`#`
Example:	`# This is a comment`
Description:	Commentary about the test. Ignored by BumbleBee.
Use:	Optional
Scope:	Single line

Directive:	`!load`
Example:	`!load tests/2003-05/examples/tunes.xml`
Description:	Provides a hint to each enabled XQuery engine to load the specified file and map it to an equivalent URI.
Use:	Optional
Scope:	Single line
Notes:	XQuery engines that load XML documents from the file system will generally ignore this hint since the relative path and the URI will be the same. XQuery engines that store XML documents in a database should load the file into the database and allow queries to reference the document using the specified URI. A `!load` directive can be used anywhere in a `.bee` file as long as it's placed before the test that references the loaded URI.

Directive:	`!enable`
Example:	`!enable saxon`
Description:	Enables only the specified XQuery engine for this test
Use:	Optional
Scope:	Single line
Notes:	All other XQuery engines are disabled for this test. Multiple engines can be enabled using multiple `!enable` directives with different engine names. By default, all engines declared in the `conf/bumblebee.properties` file are enabled. The name of the XQuery engine specified must match an engine named in the `conf/bumblebee.properties` file, else it is ignored.

Directive:	`!disable`
Example:	`!disable saxon`
Example:	`!disable`
Description:	Disables the specified XQuery engine for this test
Use:	Optional
Scope:	Single line
Notes:	All other XQuery engines are enabled for this test. The name of the XQuery engine specified must match an engine named in the `conf/bumblebee.properties` file, else it is ignored. Multiple `!disable` directives may be specified on individual lines to disable one or more XQuery engines. To disable all XQuery engines, use `!disable` without any engine name. By default, all engines declared in the `conf/bumblebee.properties` file are enabled.

Directive:	`!preserveWhitespace`
Example:	`!preserveWhitespace`
Description:	Preserves the whitespace of the text specified within the `!query` and `!result` directives, marking it as important during comparison.
Use:	Optional
Scope:	Single line
Notes:	By default, whitespace is not preserved and text within the `!query` and `!result` directives is whitespace normalized before comparison.

Organizing Tests

Any specific BumbleBee test (`.bee`) file can contain multiple tests and any specific test directory can contain multiple test files and test directories.

All the tests distributed with BumbleBee are located beneath the `$BUMBLEBEE_HOME/tests` directory. The tests written against the May 2003 draft are under `tests/2003-05`, tests written against the August 2003 draft are under `tests/2003-08`, and so on. To run all the May 2003 tests in one fell swoop, simply type:

```
bumblebee tests/2003-05
```

BumbleBee will search for all files with a `.bee` extension in the `tests/2003-05` directory and all sub-directories of the `tests` directory, recursively.

Organize your tests to suit your needs – there's no right or wrong way.

Configuring BumbleBee

Now that you've written and run a BumbleBee test, you might want to take a look under the hood to tweak BumbleBee beyond its out-of-the-box configuration.

This section describes BumbleBee's configuration options. All of the configuration variables described in this section can be found in the `conf/bumblebee.properties` file.

Configuring XQuery Engine Adapters

By default, BumbleBee is distributed with the Java runtime libraries for several free XQuery engines. In order for BumbleBee to communicate with each of these engines uniformly, it includes a set of XQuery engine adapters. These adapters allow BumbleBee tests to be run unmodified against any supported XQuery engine.

In addition to including adapters for free engines, BumbleBee includes adapters for several commercial XQuery engines. Use of those engines requires separate licenses and downloads.

When you run a BumbleBee test, or suite of tests, the test will be run against all enabled XQuery engines. For example, if you run one test with 3 XQuery engines enabled, then you will see 3 test results, with each result produced by a different XQuery engine.

The `bumblebee.properties` file contains a list of adapter configuration variables of the form:

```
bumblebee.adapter.<name>.class=bumblebee.adapters.<name>Adapter
```

The `<name>` corresponds to a specific XQuery engine adapter, such as Saxon. To enable Saxon to run BumbleBee tests, for example, simply uncomment the appropriate line in the `bumblebee.properties` file.

Note that if a BumbleBee test uses the `!disable` or `!enable` directive, then the adapter name specified must match (case insensitive) an adapter named in the `bumblebee.properties` file, else the test directive is ignored.

The `bumblebee.properties` file may also contain a set of adapter-specific properties. So, if you're using the commercial Cerisent adapter for example, then you'll want to set the following properties appropriately for your environment:

```
cerisent.url=http://localhost:8999/eval.xqe
cerisent.param=input
cerisent.xdbcHost=localhost
cerisent.xdbcPort=8998
```

These property names are adapter-specific and ignored by BumbleBee itself. That is, these properties are simply passed through to all enabled adapters and the adapters use them as they like.

Configuring Default Tests

BumbleBee is configured to run a default suite of tests when no tests are specified on the command line. The default test suite is controlled in the `bumblebee.properties` file using a list of configuration variables of the form:

```
bumblebee.test.<id>=/path/to/tests
```

Examples:

```
bumblebee.test.examples=tests/2003-05/examples
bumblebee.test.mytest=tests/2003-05/examples/MyFirstTest.bee
```

Each entry must use a unique identifier for `<id>`. The path can be either a directory of tests or an individual `.bee` file.

Changing the Log Directory

By default, BumbleBee writes the `bumblebee.log` file in the `$BUMBLEBEE_HOME/log` directory. To change the default log directory, change the following configuration variable in the `bumblebee.properties` file:

```
bumblebee.log.directory=/path/to/log/directory
```

Writing An XQuery Engine Adapter

You may discover that BumbleBee doesn't include an adapter to your XQuery engine of choice. We'd appreciate hearing about any XQuery engines we may have missed. In any event, you have two options: write an adapter using the BumbleBee API or contact us about writing an adapter for you.

The following steps describe how to write and configure a BumbleBee adapter:

1. First you'll need to write an adapter that BumbleBee can recognize. An adapter must conform to the following two-method Java interface:

```
package bumblebee;

public interface Adapter {


    /**
     * Hint to load the specified file URI, as supplied by the
     * !load directive in a test.
     *
     * @param uri The URI to load (e.g. tests/2003-05/examples/tunes.xml)
     */
    public void load(String uri) throws AdapterException;

    /**
     * Evaluates the specified XQuery expression against the
     * XQuery engine run by this adapter.
     *
     * @param query The XQuery expression to evaluate.
     * @return The result of evaluating the XQuery expression.
     *         Note that the result must not include an XML declaration.
     * @throws AdapterException if the adapter can't evaluate the expression.
     */
    public String evaluate(String query) throws AdapterException;

}
```

Note also that if your `Adapter` implementation defines a constructor taking a `java.util.Properties` instance, this constructor will be invoked with the properties defined in the `conf/bumblebee.properties` file. This is useful to allow your adapter to be dynamically configured.

2. Once you're written an implementation of the `Adapter` interface, you'll need to compile it against the `lib/bumblebee.jar` file.

3. You then need to bundle your resulting Java class file(s) in a JAR file.

4. Place the JAR file containing the adapter class file(s) in the `$BUMBLEBEE_HOME/lib/ext` directory in order for it to be automatically placed on the Java classpath used by BumbleBee at runtime.

5. You'll also need to place any external JAR files in the `$BUMBLEBEE_HOME/lib/ext` directory, such as those JAR files required for a Java-based XQuery engine you want the adapter to interface with.

6. Next, enable your new adapter by configuring it in the `conf/bumblebee.properties` file using the form:

```
bumblebee.adapter.Sample.class=mypackage.SampleAdapter
```

7. Finally, restart BumbleBee to run existing tests against the new adapter.

It's really that easy! Just write an adapter and plug it in. Of course the majority of effort lies in writing the implementation of the `Adapter` interface. Each XQuery engine will have a unique API that the adapter must adapt onto. Of course, we're happy to help licensees with adapter authoring.

Details on Specific Adapters

Here are some details on how to set up and configure the XQuery engine adapters as provided with BumbleBee.

Qexo , Qizx, and Saxon

BumbleBee includes these three XQuery engines in its distribution. You can find their runtime JAR files in `lib/ext`. These three engines run in-process to BumbleBee and require no special customizations.

Cerisent

Cerisent is a commercial XQuery engine from `http://cerisent.com`. BumbleBee includes a `CerisentAdapter` but not the Cerisent engine itself. For BumbleBee to connect to a Cerisent server, you must do a little configuration. First, download the `xdbc.jar` and `xqe.jar` files from the Cerisent support site and place them in BumbleBee's `lib/ext`. Then, using the Cerisent administration web pages, create an HTTP server and an XDBC server running against the same Forest. Place the query file `eval.xqe` (provided under the BumbleBee `doc` directory) under the document root for the HTTP server you created. Lastly, edit the `bumblebee.properties` file and adjust the properties beginning "`cerisent.`" so they point at the HTTP server URL, XDBC hostname, and XDBC port of the Cerisent server you configured. To put Cerisent into the engine run list, make sure to uncomment the `bumblebee.adapter.Cerisent.class` property.

Ipedo

Ipedo is a commercial XQuery engine from `http://ipedo.com`. BumbleBee includes an `IpedoAdapter` but not the Ipedo engine itself. For BumbleBee to connect to an Ipedo server, you have to do a little configuration. First, find the `xmlclient.jar` included with the Ipedo distribution and place a copy in BumbleBee's `lib/ext`. Then, using the Ipedo administration tools, create a collection for use in testing. Lastly, edit the `bumblebee.properties` file and adjust the properties beginning "ipedo." so they will connect to your server using the right username and password for running against the selected collection. To put Ipedo into the run list, make sure to uncomment the `bumblebee.adapter.Ipedo.class` property.

IPSI-XQ

IPSI-XQ is a commercial XQuery engine from `http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html`. BumbleBee includes an `IPSIAdapter` but not the IPSI-XQ engine itself. Because IPSI-XQ can run in-process to BumbleBee, setup is easy. Just copy the ipsi-xq.jar and `xercesImpl.jar` files into BumbleBee's `lib/ext` directory. To put IPSI-XQ into the run list, make sure to uncomment the `bumblebee.adapter.IPSI.class` property.

X-Hive

X-Hive is a commercial XQuery engine from `http://xhive.com`. BumbleBee includes an `XhiveAdapter` but not the X-Hive engine itself. For BumbleBee to connect to an X-Hive server, you must do a little configuration. First, copy the following JARs included with the X-Hive distribution into BumbleBee's `lib/ext` directory: `antlr.jar`, `dom3-intermediate.jar`, `icu4j.jar`, `xercesImpl.jar`, and `xhive.jar`. Then, using the X-Hive administration tools, create a database and remember its name. Edit the `bumblebee.properties` file and adjust the properties beginning "`xhive.`" so they match the configuration of your server. To put X-Hive into the run list, make sure to uncomment the `bumblebee.adapter.Xhive.class` property.

Licensing

Checking The License

You can check the status of your BumbleBee license by typing:

```
bumblebee -license
```

This command simply prints the contents of the `bumblebee.license` file.

A license is valid for a specific version and edition of BumbleBee. To check the version and edition of BumbleBee currently installed, type:

```
bumblebee -version
```

Evaluation Licenses

If you're using an evaluation copy of BumbleBee, the `Expiration Date` field of the license indicates when your license expires. A non-expiring BumbleBee license is available by emailing buzz@xquery.com with your usage requirements.

Support

BumbleBee Mailing List

If you're evaluating BumbleBee and have questions not answered in this guide, or you wish to file a bug report or request new features, please use the mailing list:

http://xquery.com/mailman/listinfo/talk

Email Support

Licensed BumbleBee users are entitled to 30 days of direct email support.