

Write high performance, Java data access applications, Part 1: Introducing pureQuery annotated method style

Maximize security and configurability

Skill Level: Introductory

[Heather E. Lamb \(hlamb@us.ibm.com\)](mailto:hlamb@us.ibm.com)
Software Engineer
IBM

10 Apr 2008

pureQuery is a high-performance Java™ data access platform focused on simplifying the tasks of developing and managing applications that access data. It consists of tools, APIs and a runtime. This article introduces the pureQuery annotated method coding style -- a simple, flexible style falling under the named-query paradigm, capable of executing SQL statically or dynamically. This article explains why a developer might choose to write a pureQuery application in the annotated method style, explains some of the differences between the annotated method style and the pureQuery inline coding style, and gives a brief overview of the most powerful features of pureQuery annotated methods.

Overview

pureQuery

pureQuery is a feature available for IBM Data Studio and can be obtained with [IBM Data Studio pureQuery Runtime](#) or with [IBM Data Studio Developer](#).

This article covers the following topics related to the annotated method programming style:

- Description of the annotated method programming style
- Motivation for choosing the annotated method coding style
- Steps for developing pureQuery applications in the annotated method style (Go to this [section](#) now.)
- Description of code generation and examples of generated code
- Requirements for defining annotated methods in a pureQuery interface
- Use of a pureQuery interface to execute SQL
- Introduction to a few select features of the annotated method style, such as batching, generated `RowHandlers` and `ParameterHandlers`, generated keys, and use of XML configuration files to modify code generator output

If you are ready to start coding, skip to the [technical breakdown](#). A brief example follows to show why developers might choose to develop a pureQuery application using the annotated method style.

What is the annotated method programming style?

To introduce the annotated method coding style, it helps to understand the background behind each of the two pureQuery coding styles.

The *inline coding style* was developed in response to customer demand for quick, simple coding style that would be easy to learn by developers familiar with Java™ Database Connectivity (JDBC) -- only simpler and quicker to code. The inline style originally was targeted to reduce the repeated coding tasks familiar to the JDBC programmer, as well as to provide an API that tools could easily use to tie in data access development with Java development. The coding is referred to as "inline" because of the way SQL statements are defined in the application. In the inline style, SQL statements are declared or constructed at runtime, and passed as instances of `String` to common `Data` interface methods. The inline style maximizes coding speed and development flexibility, and supports dynamic execution. An overview of the common `Data` interface APIs will follow in a subsequent article. More information on the inline style can be found in the pureQuery documentation (See [Resources](#)).

This article focuses on the *annotated method coding style*, which evolved similarly to the inline coding style but with the additional goal of maximizing configurability and security for the resulting pureQuery application. Annotated method style is designed specifically to support both dynamic and static database access. It was developed in response to customer demand for a named query programming interface for data access that was similar to Java Persistence API (JPA) -- only simpler, quicker to

code, and capable of supporting static execution when required.

Like the inline coding style, the annotated method is also rooted in the way SQL statements are defined in the application. In the annotated method style, the SQL string is defined as an element of a Java 5.0, `pureQuery` Annotation. The method annotations defined by `pureQuery` for this purpose are `@Select` (which annotates SQL queries), `@Update` (which annotates SQL DML statements), and `@Call` (which annotates SQL CALL statements). The annotations are placed on user-defined method declarations within a user-defined interface. A code generator pre-processes the interface to generate implementation code for each declared, annotated method. The generated implementation code executes the SQL statements defined in the annotation using the `pureQuery` runtime. Pre-defining the SQL string in an annotation element simplifies static execution support.

Data Studio tooling support for the annotated method style includes a code generator which creates implementations of annotated methods written by the user. The result of code generation is a second implementation class which is compiled and used to execute the SQL statements declared in the original interface. Figure 1 illustrates the relationship between the user-defined annotated method interface, the code generator, and the generated implementation class.

Figure 1. pureQuery annotated method code generation



A motivating example

This section introduces a fictional example: the data access development team for Silver Castles, a growing company which sells a variety of silver products. The company is developing a new web-based storefront on the Silver Castles website and has decided to use the `pureQuery` environment to develop their data access application's persistence layer. It was an easy decision to use `pureQuery` once they saw the [pureQuery tools demo](#) and ran through the [pureQuery tutorial](#) for themselves. They learned how easy it could be to quickly develop, test, and deploy a data-access, Java-based persistence layer targeted for multiple platforms. A few additional factors come into play as the fictional team decides when to use the annotated method coding style, and when to use the inline coding style. One of the

developers named Bob decides to become an expert on the annotated method coding style. A description follows of the knowledge he gathers to help the team decide when to use each coding style.

The decision to use annotated methods

Bob begins by assessing the annotated method coding style. After a few hours, he comes back to his team with a list of highlights that describe the annotated method style.

The paradigm:

- Follows the named-query style, targeted to match or exceed the capabilities of existing object-relational mappers and persistence solutions
- Encourages separation of persistence layer (CRUD) statements from the rest of the application inside the annotated interface. The result is that development changes to the SQL statements are not scattered throughout the application and will not impact developers working on other layers. The annotated interface provides a central control point to manage the SQL statements for the entire application.
- SQL statement text is known at development time. It can be analyzed ahead of time, allowing developers to optimize assignment to output objects.

Static support:

- pureQuery annotated method coding style makes the decision to execute static SQL a deployment option rather than a design option. Annotated method code can either be deployed dynamically or by using static SQL packages on IBM Data Servers without a single line of code change
- Transparent transition to static execution from the perspective of application developers and front-end applications
- A decision to deploy for static execution obtains the universal benefits of static SQL: security, performance, monitoring, and pre-runtime optimization

Code generation:

- Minimal handwriting of code: interface method signatures and the SQL statement are declared; then the pureQuery code generator produces the data access code implementation

- Query results are returned pre-processed by the pureQuery runtime into a wide choice of objects, such as `Iterators` of populated [pureQuery beans](#), or `Lists` of column name to column value `Maps`
- Customize behavior of generated code using a [Hook](#) for specialized pre- and post- data access processing
- Use XML [configuration files](#) to define different SQL for each target database

Developing and using annotated methods

Listing 1 below shows part of a pureQuery interface Bob writes to practice using the annotated method style. He uses the interface to explain the fundamentals of the annotated method style to the team. The pureQuery annotated interface is the building block of the annotated method style. A pureQuery interface consists of the following:

1. A Java `interface` definition containing:
 - a. One or more method declarations
 - b. A pureQuery data access annotation for each method declaration - `@Select`, `@Update`, or `@Call`. Each annotation includes an SQL element containing the SQL statement to execute when the method is called. (The SQL statement can optionally be provided separately in a configuration file instead of in an annotation element.)
 - c. A Java return type for each method declaration which indicates the desired object format for the pureQuery runtime to return data access results to the caller
 - d. Java parameter types for each method signature, including parameters indicating the object type of parameters used to execute the SQL statement

The interface definition file contains no other method declarations or definitions other than those described above. Once the pureQuery interface is defined, other layers of the application can conceptually treat the interface as a database -- one that is easy to access from Java simply by calling the declared Java data access methods. When another application layer calls one of the annotated methods, it receives the result of executing the associated SQL statement in a conveniently formatted Java object. No knowledge of SQL is required to develop the other application layers.

Listing 1. pureQuery annotated method style interface

```
package com.ibm.db2.pureQuery;

import java.util.Iterator;
import com.ibm.pdq.annotation.Select;

public interface CustomerData {
    // Select all PDQ_SC.CUSTOMERS and populate Customer beans with results
    @Select(sql="select CID, NAME, COUNTRY, STREET, CITY, PROVINCE, ZIP, PHONE,
              INFO from PDQ_SC.CUSTOMER")
    Iterator<Customer> getCustomers();

    // Select PDQ_SC.CUSTOMER by parameters and populate Customer bean with results
    @Select(sql="select CID, NAME, COUNTRY, STREET, CITY, PROVINCE, ZIP, PHONE,
              INFO from PDQ_SC.CUSTOMER where CID = ?")
    Customer getCustomer(int cid);
    ...
}
```

Generating an implementation

Once the interface above is defined, it is automatically fed to the [pureQuery code generator](#) by pureQuery project tooling in Data Studio. The pureQuery code generator produces an implementation of the interface in a file named `CustomerDataImpl.java`. For the reader's information, the generated file is available in [Listing 2](#). Note: none of the code in the generated file ([Listing 2](#)) is handwritten by a developer.

It is not necessary to examine the generated code. If you would rather see how the Silver Castle application uses the interface above, skip to the [next step](#). Continue to read to learn more about the generated code.

The code generator and generated code

The code generator takes user-defined pureQuery interface as input, such as the simple `CustomerData` defined by the SilverCastles developers. It generates the code required to execute each SQL statement annotated for each method in the interface. It also generates code to process the results into the declared result types. The generated code calls the pureQuery runtime to perform the processing required for each method.

A few important notes about the generated code:

- **Generated elements:** For each declared method, the generated elements include the method definition, an internal pureQuery `StatementDescriptor`, a generated [RowHandler](#) or [ResultHandler](#), as required, and an internal pureQuery `ParameterHandler`, as required.

- **Implementation class name:** The name of the generated implementation file is based on the name of the original user-defined interface, with “Impl” appended at the end of the name. In this example, the developers wrote the `CustomerData` interface, and the code generator produced the `CustomerDataImpl` class. The generated class implements the `CustomerData` interface. The implementation class name is never used by other layers of the data access application; they always reference and use the user-defined interface. However, it’s useful to know the name of the implementation file in case the generated code needs to be examined.
- **Implementation superclass:** In addition to implementing the user-defined interface, the generated implementation class extends the internal `pureQuery` class `BaseData`, which in turn implements the external `Data` interface. This superclass is part of the `pureQuery` runtime and handles the rote, repeated operations required to access a database and process results.
- **Don’t get confused looking at the generated code.** Developers never need look at it unless they choose to.

Using annotated methods

At this point, the project has been built, Data Studio `pureQuery` tooling has invoked the `pureQuery` code generator, and the interface implementation has been generated and compiled. The developers are ready to access data from other layers of the application by calling the annotated methods declared in the `pureQuery` annotated interface. This process is simple. The application instantiates an instance of the `pureQuery` interface (the conceptual database) with a request to the [DataFactory](#). It then calls the data access methods, as follows:

Listing 3. Calling annotated methods

```
...
java.sql.Connection con = ...;

// use the DataFactory to instantiate the CustomerData interface
CustomerData cd = DataFactory.getData(CustomerData.class, con);

// execute the SQL for getCustomers() and get the results in Customer beans
Iterator<Customer> cust = cd.getCustomers();

// the application can now consume the Iterator of Customer beans
...
```

With a few lines of code, the database is accessed, the desired SQL statement is executed, and the results are processed into a convenient `Iterator` of data beans of class `Customer`, ready to be consumed by the application. What is even more impressive is that even to implement the persistence layer method, the developers

still had no more work to do than to declare the method called here. The pureQuery code generator and runtime handled the rest.

Breaking down the annotated method style

This section breaks down the example into the important elements of the annotated method coding style. The pureQuery documentation contains complete coverage of these and other pureQuery coding concepts (See [Resources](#)).

The pureQuery interface

In the example above, the user-defined pureQuery interface is named `CustomerData`. It is helpful, as in the Silver Castles example, to name the interface to reflect the information source it represents. For example, `CustomerData` methods retrieve and update information about the company's customers.

As described above, the pureQuery interface contains only annotated method declarations.

Annotated method declarations

Each method declaration in the pureQuery interface contains the following required elements:

- One of three pureQuery annotations: `@Select`, `@Update`, or `@Call`
- One `sql=<string>` element for each annotation, where the string contains a valid SQL statement to be executed when the method is called (optionally, the SQL statement can be provided in an XML configuration file instead)
- A standard Java method declaration

Declared return types

The return type of the method declaration indicates in what object format the results of the SQL statement will be returned. Supported return types vary depending on the annotation used. For example, a variety of collection and simple types are returned from queries. Standard update count formats are available for updates. And among others, a `StoredProcedureResult` return type conveniently encapsulates the results of a stored procedure call.

Results can also be processed by the pureQuery engine into a collection of user-defined pureQuery beans. In our example, the developers declare that an `Iterator` of `Customer` beans should be returned. pureQuery uses a set of [bean conventions and requirements](#) to map database query results directly to the

user-defined bean class. [Annotations in pureQuery beans](#) can override the default mapping behavior. Also, manual mapping of results into beans can be performed by a user-provided `RowHandler`. Otherwise, the generated `RowHandler` automatically carries out default mapping to the user-defined bean class, prior to returning bean results to the caller.

Declared parameter types

The parameter types in the annotated method declaration determine how pureQuery obtains SQL statement parameter values at runtime. There can be, but there is not necessarily, a one-to-one mapping of parameter markers in the SQL statement to declared parameters in the annotated method. pureQuery follows the [rules for parameter marker syntax](#) to determine how SQL statement parameters are mapped from the declared parameters in the annotated method's parameter list.

Parameters can more easily take on a wide variety of types in pureQuery than in other data access APIs. For example, a single declared pureQuery bean parameter can potentially provide runtime values for [several SQL statement parameter markers](#). pureQuery beans and Java collection types are supported for pureQuery annotated methods. View the annotated method [syntax diagram](#) for a complete listing of possible declared parameter types. The declared parameter type for an annotated method can make a significant difference in how the SQL statement is executed. For example, batch updates can be initiated by the use of certain parameter types in an `@Update` annotated method. For batch updates, a collection parameter is used vertically to execute an SQL statement more than once, with different parameter values each time.

Annotated method parameters can be used in a straightforward, simple way, or they can be assigned to take advantage of specific processing capabilities of the pureQuery engine. Batch update processing is one example of the underlying power of the pureQuery engine to automatically optimize code and streamline development effort. Specialized pureQuery processing for data bean parameters also saves development time and efforts. For example, the `@GeneratedKey` pureQuery bean annotation allows the pureQuery engine to automatically update a bean parameter's field with a database-generated value following an insert or update operation.

Hooks

Some of the pureQuery development team is initially wary of the fact that most code is generated using annotated method style. Bob explains that a [Hook](#) provides an easy way to conduct specialized processing during execution of generated code. Rather than hand code method calls to surround each application call to an annotated interface methods, another option is to define a `Hook` to register with the annotated interface. The `Hook` is called back by the pureQuery runtime at entry to and exit from each annotated method. This provides a callback mechanism to surround generated data access code with specialized processing. A `Hook` defines

the required `pre()` method which, when the `Hook` is registered, is called immediately upon entrance to an annotated method. A `Hook` will also define a `post()` method which, when the hook is registered, will be called immediately before control is returned from an annotated method.

`Hook` methods provide context awareness for specialized runtime callback processing in case it is needed by the user-defined implementation of `pre()` and `post()`. This awareness is enabled by the parameters on the `pre()` and `post()` method calls. Special processing can be designed to vary according to the values of these parameters. For example, processing may vary depending on the name of the interface method calling the `Hook` method, the runtime parameter values provided to that method, or the SQL statement type which the method executes. A handle to the `pureQuery` interface with which the `Hook` is registered is also available, in the form of a `Data` parameter. Each of these values is available for examination and modification by the implementer of `Hook`. In addition, the return value is available to the implementer of `post()` before it is returned to the caller. Here an example of `Hook` processing code Bob wrote as a demonstration for the Silver Castles team:

Listing 4. Hooks for specialized processing

```
public static class TrackingHook implements Hook {
    public void pre(String methodName, Data objectInstance,
        SqlStatementType sqlStatementType, Object... parameters) {
        System.out.println(methodName + "***Customer data has been accessed***");
    }

    public void post(String methodName, Data objectInstance,
        Object returnValue, SqlStatementType sqlStatementType,
        Object... parameters) {
        // do nothing
    }
}
```

Now that specialized processing has been defined using `Hook`, the developers ensure it will run by registering an instance of their `Hook` with the interface upon instantiated. Listing 5 shows how to register a `Hook`:

Listing 5. Registering a Hook

```
...
Connection con = ...;

// use the DataFactory to instantiate the interface and
// provide an instance of Hook to be registered with the instance
CustomerData cd = DataFactory.getData(CustomerData.class, con, new TrackingHook());

// execute the SQL for getCustomers() and get the results,
// the pre() and post() methods are automatically called
Iterator<Customer> cust = cd.getCustomers();

// the application now consumes the Iterator of Customer beans
```

...

This is a very simple example of specialized processing with `Hook`. See [the Hook example in the pureQuery documentation](#) for a more complex example of the type of processing `Hook` can be used to implement.

Developing for multiple targets

The final feature of annotated methods Bob explains to his team is the use of an [XML configuration file](#) to completely separate SQL statements from Java code implementation of their application. This provides the ability to avoid re-coding a Java application when it is deployed against a target database which requires different SQL statements. For example, if the team wants to deploy the same application against a legacy data source with a slightly different schema, they won't need to re-code their annotated interface or pureQuery bean.

Listing 6 shows the SQL statement for the original schema.

Listing 6. SQL statement for original schema

```
select CID, NAME, COUNTRY, STREET, CITY, PROVINCE, ZIP, PHONE, INFO
from PDQ_SC.CUSTOMER
```

Listing 7 shows the SQL statement for the legacy schema.

Listing 7. SQL statement for legacy schema

```
select CUSTID, NAME, COUNTRY, STREET, CITY, PROV, ZIP, PHONE, INFO
from PDQ_SC.CUSTOMER
```

Note that in Listing 7, the names of the `CID` and `PROVINCE` columns become `CUSTID` and `PROV`. This changes the SQL statement required to issue the query, and it changes the default mapping of results to the Customer data bean.

Listing 8 shows the customer pureQuery bean.

Listing 8. Customer pureQuery bean

```
package com.ibm.db2.pureQuery;

public class Customer {

    // Class variables
    protected int cid;
    protected String name;
    protected String country;
    protected String street;
    protected String city;
    protected String province;
```

```
protected String zip;  
protected String phone;  
protected String info;  
...
```

Instead of writing a new annotated interface or `Customer` bean class to support the legacy schema, the team uses an XML configuration file to provide additional input to the generator to support the legacy schema. A fragment of the XML configuration file is listed below, showing how one of the SQL strings and the user-defined bean class mapping are overridden. To prepare to deploy their application against the legacy system, the team provides their original annotated interface definition to the generator along with the XML file below. The generator produces correct generated code to deploy against the legacy database:

Listing 9. XML configuration file to generate alternate code

```
<?xml version="1.0" encoding="UTF-8"?>  
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm">  
  
  <named-native-query name=" com.ibm.db2.pureQuery.CustomerData#getCustomers()">  
    <query><![CDATA[select CUSTID, NAME, COUNTRY, STREET, CITY, PROV, ZIP, PHONE, INFO  
                      from PDQ_SC.CUSTOMER]]>  
    </query>  
  </named-native-query>  
  ...  
  <entity class="com.ibm.db2.pureQuery.Customer">  
    <attributes>  
      <basic name="cid">  
        <column name="CUSTID" />  
      </basic>  
      ...  
      <basic name="province">  
        <column name="PROV" />  
      </basic>  
      ...  
    </attributes>  
  </entity>  
</entity-mappings>
```

For more information about providing XML configuration files to the generator to provide alternate SQL statements and/or object mappings for pureQuery interfaces, refer to the pureQuery [online documentation](#).

Summary

This article provided a high-level introduction to the pureQuery annotated method coding style, along with likely motivations for a development team to choose to code using pureQuery annotated methods. It also outlines the basic steps required to develop an annotated method style application. Selected features of the style were introduced.

If you are interested in learning more about developing pureQuery annotated

method style applications, please follow the links throughout the article and in the Resources section to the pureQuery online documentation, additional articles, and helpful tutorials.

Resources

Learn

- Read the [pureQuery documentation](#) for a complete description of the tool.
- The [pureQuery demo](#) is the easiest introduction to the pureQuery IDE tooling.
- The "Increase productivity in Java database development with new IBM pureQuery tools" [series](#) covers the basics of the tooling, shows you how to detect and fix SQL problems in Java programs, and provides a full tutorial about pureQuery rapid application development.
- [developerWorks Information Management zone](#): Learn more about Information Management. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with [developerWorks technical events and webcasts](#).
- [Technology bookstore](#): Browse for books on these and other technical topics.

Get products and technologies

- [pureQuery javadoc](#) is available as part of the online documentation.
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the author

Heather E. Lamb

Heather Lamb is a developer on the pureQuery runtime team, in Silicon Valley.