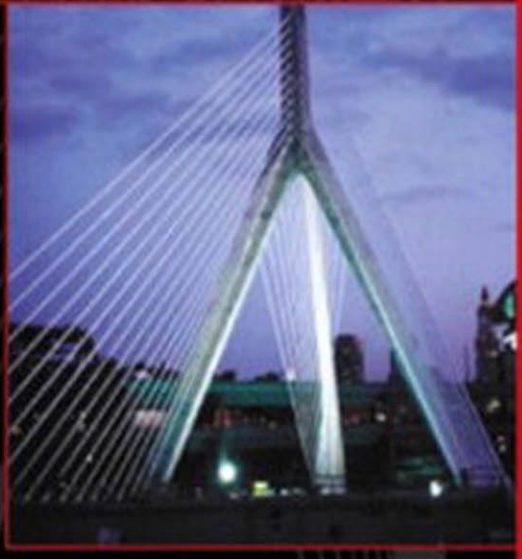
The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Patterns of Enterprise Application Architecture

Martin Fowler

With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford



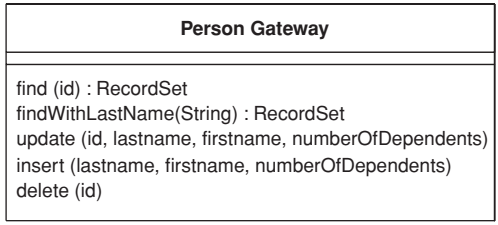
Patterns of Enterprise Application Architecture

Chapter 10

Data Source Architectural Patterns

Table Data Gateway

An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table.



Mixing SQL in application logic can cause several problems. Many developers aren't comfortable with SQL, and many who are comfortable may not write it well. Database administrators need to be able to find SQL easily so they can figure out how to tune and evolve the database.

A *Table Data Gateway* holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes. Other code calls its methods for all interaction with the database.

How It Works

A *Table Data Gateway* has a simple interface, usually consisting of several find methods to get data from the database and update, insert, and delete methods. Each method maps the input parameters into a SQL call and executes the SQL against a database connection.The *Table Data Gateway* is usually stateless, as its role is to push data back and forth.

The trickiest thing about a *Table Data Gateway* is how it returns information from a query. Even a simple find-by-ID query will return multiple data items. In environments where you can return multiple items you can use that for a single row, but many languages give you only a single return value and many queries return multiple rows.

One alternative is to return some simple data structure, such as a map. A map works, but it forces data to be copied out of the record set that comes from the database into the map. I think that using maps to pass data around is bad form because it defeats compile time checking and isn't a very explicit interface,

leading to bugs as people misspell what's in the map. A better alternative is to use a *Data Transfer Object (401)*. It's another object to create but one that may well be used elsewhere.

To save all this you can return the *Record Set (508)* that comes from the SQL query. This is conceptually messy, as ideally the in-memory object doesn't have to know anything about the SQL interface. It may also make it difficult to substitute the database for a file if you can't easily create record sets in your own code. Nevertheless, in many environments that use *Record Set (508)* widely, such as .NET, it's a very effective approach. A *Table Data Gateway* thus goes very well with *Table Module (125)*. If all of your updates are done through the *Table Data Gateway*, the returned data can be based on views rather than on the actual tables, which reduces the coupling between your code and the database.

If you're using a *Domain Model (116)*, you can have the *Table Data Gateway* return the appropriate domain object. The problem with this is that you then have bidirectional dependencies between the domain objects and the gateway. The two are closely connected, so that isn't necessarily a terrible thing, but it's something I'm always reluctant to do.

Most times when you use *Table Data Gateway*, you'll have one for each table in the database. For very simple cases, however, you can have a single *Table Data Gateway* that handles all methods for all tables. You can also have one for views or even for interesting queries that aren't kept in the database as views. Obviously, view-based *Table Data Gateways* often can't update and so won't have update behavior. However, if you can make updates to the underlying tables, then encapsulating the updates behind update operations on the *Table Data Gateway* is a very good technique.

When to Use It

As with *Row Data Gateway (152)* the decision regarding *Table Data Gateway* is first whether to use a *Gateway (466)* approach at all and then which one.

I find that *Table Data Gateway* is probably the simplest database interface pattern to use, as it maps so nicely onto a database table or record type. It also makes a natural point to encapsulate the precise access logic of the data source. I use it least with *Domain Model (116)* because I find that *Data Mapper (165)* gives a better isolation between the *Domain Model (116)* and the database.

Table Data Gateway works particularly well with *Table Module (125)*, where it produces a record set data structure for the *Table Module (125)* to work on. Indeed, I can't really imagine any other database-mapping approach for *Table Module (125)*.

Just like *Row Data Gateway (152)*, *Table Data Gateway* is very suitable for *Transaction Scripts (110)*. The choice between the two really boils down to how they deal with multiple rows of data. Many people like using a *Data Transfer Object (401)*, but that seems to me like more work than is worthwhile, unless the same *Data Transfer Object (401)* is used elsewhere. I prefer *Table Data Gateway* when the result set representation is convenient for the *Transaction Script (110)* to work with.

Interestingly, it often makes sense to have the *Data Mappers (165)* talk to the database via *Table Data Gateways*. Although this isn't useful when everything is handcoded, it can be very effective if you want to use metadata for the *Table Data Gateways* but prefer handcoding for the actual mapping to the domain objects.

One of the benefits of using a *Table Data Gateway* to encapsulate database access is that the same interface can work both for using SQL to manipulate the database and for using stored procedures. Indeed, stored procedures themselves are often organized as *Table Data Gateways*. That way the insert and update stored procedures encapsulate the actual table structure. The find procedures in this case can return views, which helps to hide the underlying table structure.

Further Reading

[Alur et al.] discusses the *Data Access Object* pattern, which is a *Table Data Gateway*. They show returning a collection of *Data Transfer Objects (401)* on the query methods. It's not clear whether they see this pattern as always being table based; the intent and discussion seems to imply either *Table Data Gateway* or *Row Data Gateway (152)*.

I've used a different name, partly because I see this pattern as a particular usage of the more general *Gateway (466)* concept and I want the pattern name to reflect that. Also, the term *Data Access Object* and its abbreviation *DAO* has its own particular meaning within the Microsoft world.

Example: Person Gateway (C#)

Table Data Gateway is the usual form of database access in the windows world, so it makes sense to illustrate one with C#. I have to stress, however, that this classic form of *Table Data Gateway* doesn't quite fit in the .NET environment since it doesn't take advantage of the ADO.NET data set; instead, it uses the data reader, which is a cursor-like interface to database records. The data

reader is the right choice for manipulating larger amounts of information when you don't want to bring everything into memory in one go.

For the example I'm using a Person Gateway class that connects to a person table in a database. The Person Gateway contains the finder code, returning ADO.NET's data reader to access the returned data.


```
class PersonGateway...

      public IDataReader FindAll() {
         String sql = "select * from person";
         return new OleDbCommand(sql, DB.Connection).ExecuteReader();
      }
      public IDataReader FindWithLastName(String lastName) {
         String sql = "SELECT * FROM person WHERE lastname = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add(new OleDbParameter("lastname", lastName));
         return comm.ExecuteReader();
      }
      public IDataReader FindWhere(String whereClause) {
         String sql = String.Format("select * from person where {0}", whereClause);
         return new OleDbCommand(sql, DB.Connection).ExecuteReader();
      }
```

Almost always you'll want to pull back a bunch of rows with a reader. On a rare occasion you might want to get hold of an individual row of data with a method along these lines:

```
class PersonGateway...

      public Object[] FindRow (long key) {
         String sql = "SELECT * FROM person WHERE id = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add(new OleDbParameter("key",key));
         IDataReader reader = comm.ExecuteReader();
         reader.Read();
         Object [] result = new Object[reader.FieldCount];
         reader.GetValues(result);
         reader.Close();
         return result;
      }
```

The update and insert methods receive the necessary data in arguments and invoke the appropriate SQL routines.

```
class PersonGateway...

      public void Update (long key, String lastname, String firstname, long numberOfDependents){
            String sql = @"
               UPDATE person
                  SET lastname = ?, firstname = ?, numberOfDependents = ?
                  WHERE id = ?";
```

```
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("last", lastname));
        comm.Parameters.Add(new OleDbParameter ("first", firstname));
        comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }
```

```
class PersonGateway...

    public long Insert(String lastName, String firstName, long numberOfDependents) {
        String sql = "INSERT INTO person VALUES (?,?,?,?)";
        long key = GetNextID();
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.Parameters.Add(new OleDbParameter ("last", lastName));
        comm.Parameters.Add(new OleDbParameter ("first", firstName));
        comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
        comm.ExecuteNonQuery();
        return key;
    }
```

The deletion method just needs a key.

```
class PersonGateway...

    public void Delete (long key) {
        String sql = "DELETE FROM person WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }
```

Example: Using ADO.NET Data Sets (C#)

The generic *Table Data Gateway* works with pretty much any kind of platform since it's nothing but a wrapper for SQL statements. With .NET you use data sets more often, but *Table Data Gateway* is still useful although it comes in a different form.

A data set needs data adapters to load the data into it and update the data. In find it useful to define a holder for the data set and the adapters. A gateway then uses the holder to store them. Much of this behavior is generic and can be done in a superclass.

The holder indexes the data sets and adapters by the name of the table.

```
class DataSetHolder...

    public DataSet Data = new DataSet();
    private Hashtable DataAdapters = new Hashtable();
```