

Test Driven Database Development With DbFit

Gojko Adzic

Marisa Seal

Test Driven Database Development With DbFit

Gojko Adzic

Marisa Seal

Published 2008-08-22

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where these designations appear in this book, and the authors were aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This is a free document, and you have the right to redistribute it unmodified, in original form. All other rights are reserved by the authors.

This document contains copyright protected portions from Test Driven .NET Development with FitNesse (ISBN:978-0-9556836-0-2, published by Neuri Limited in 2008), used with publisher's permission.

You can access the most recent version of this document online at <http://www.fitnessse.info/dbfit>, where you will also find links to contact the authors or provide feedback.

1. Introduction	1
Project goals	1
Features	1
What's wrong with xUnit	2
How this document is organised	4
2. Database unit tests	5
Installing DbFit	5
Hello World from the database	8
Step 1: Creating a new test page	8
Step 2: Setting up the environment	9
Step 3: Connect to the database	9
Step 4: Testing a simple query	10
Step 5: Running the test	11
Traffic lights	13
Managing Wiki content	14
Organising pages	14
Writing tests in Excel/Word	15
Formatting text	16
Preventing unwanted formatting	16
A note on flow and standalone modes before we continue	17
3. DbFit for Integration tests	19
Why use DbFit for integration tests?	19
Installing DbFit	19
Why not use generic ADO.NET/JDBC interfaces?	21
Connecting to the database	21
Connecting in flow mode	22
Connecting in standalone mode	23
Storing connection properties in a file	24
Transaction management	25
4. Command reference	27
Set-up Script	27
Query	28
Ordering and row matching	29
Using parameters	30
Avoiding parameter mapping	30
Multi-line queries and special characters	31
Working with padded chars	31
Insert	32
Storing auto-generated values	32
Update	34
Execute Procedure	35
Calling Functions	36

Expecting exceptions	36
Execute	37
Inspect	38
Store Query	39
Compare Stored Queries	40
5. Best practices	43
Initialising tests	43
Markup variables can help you keep your data straight	43
Reusing DbFit tables	45
6. Frequently asked questions	49
I'd like to use DbFit with Sybase/PostGRE. Is that possible?	49
NULLs and blank cells	49
DbFit complains that it cannot read columns or parameters. What's wrong?	49
Does DbFit require any special database privileges?	50
Does DbFit support VARBINARY columns?	50
My stored procedure returns a result set. How do I use it?	50
DbFit says that my VARBINARY is System.Byte[]	50
Does DbFit support GUID columns?	51
DbFit complains about an unsupported type. What's wrong?.....	51
Can you extend DbFit to support Oracle collection types?	51
How can we use Windows-integrated authentication?	51
DBFit complains about invalid fixtures/methods	52
Why does DbFit not see the time portion of my Date fields?	52
DbFit complains about registering a SQL Server driver	52

Introduction

DbFit makes test driven database development easy. Depending on whether you are primarily working in a database environment or in a .NET/Java environment, you can look at DbFit as:

- For database developers — a neat unit-testing tool for stored procedures and database objects, which finally allows you to write database tests in a tabular, relational form, without requiring you to learn or use an object-oriented language.
- For .NET/Java developers — a set of FIT fixtures which enables FIT/FitNesse tables to execute directly against a database.

Project goals

This dual nature of DbFit is reflected in two main project goals:

- Support efficient database acceptance and unit testing by providing database developers a good tool to express and manage tests in a relational language, without any .NET/Java knowledge required.
- Support efficient .NET/Java integration testing by providing standardised FitNesse fixtures to manage database state from FitNesse.

Features

Here is a quick overview of DbFit functionality:

- Regression testing for queries and SQL statements
- Functional testing for stored procedures and functions
- Various short-cuts to make writing test scripts easier and more efficient: automatic transaction control, building regression tests for queries, inspecting database meta-data, and more.
- Support for Oracle, SQLServer 2000 and 2005, DB2, MySql 5 and Derby (MySQL and Derby are supported only in the Java version. Java version supports SqlServer partially — not all data types are implemented at the moment.)

What's wrong with xUnit

DbFit is the result of a three year long effort to apply agile development practices in a database-centric environment. Lack of proper tools for database-level testing was one of the major obstacles in that effort, and DbFit finally solved that issue. Here is a very short summary of that journey and reasons why DbFit was originally created. If you are interested in finding out more about the wider problem and applying agile practices to databases, see my article *Fighting the monster*¹ and Scott Ambler's site <http://www.agiledata.org>.

Agile practices and databases do not often go hand in hand. For starters, most of the innovation today is in the object-oriented and web space, so database tools are a bit behind. Compared to say Idea or Eclipse, the best IDE available for Oracle PL/SQL development is still in the ice ages. This has influenced the database testing tools and libraries – most of the tools currently available are copies of JUnit translated into the database environment. Examples are utPLSQL² and TSQLUnit.³ Some other tools, like DbUnit⁴ just focus on setting the stage for Java or .NET integration tests, not really for executing tests directly against database code.

The problem with xUnit-like database testing tools is that they require too much boilerplate code. I could never get database developers to really use them when no one was looking over their shoulders. Writing tests was simply seen as too much overhead. All the buzz about object-relational mismatch over the last few years was mostly about relational models getting in the way of object development. This is effectively the other side of the problem, with object tools getting in the way of relational testing.

FIT testing framework, on the other hand, does not suffer from that mismatch. FIT is an acceptance testing framework developed by Ward Cunningham, which is customer oriented and has nothing to do with database unit testing whatsoever. But FIT tests are described as tables, which is much more like the relational model than Java code. FIT also has a nice Web-wiki front-end called FitNesse, which allows database developers to write tests on their own without help from Java or .NET developers. DbFit utilises the power of these two tools to make database tests easy.

¹<http://gojko.net/2007/11/20/fighting-the-monster/>

²<http://utplsql.sourceforge.net/>

³<http://tsqlunit.sourceforge.net/>

⁴<http://www.dbunit.org/>

My goal with DbFit was not just to enable efficient database testing — it was to motivate database developers to use an automated testing framework. That is why DbFit has quite a few shortcuts to make database testing easier through DbFit than even doing manual validations in PL/SQL or TSQL. I will explain these later on, but for starters — DbFit automatically manages transactions for you (rolling back by default to make tests repeatable), retrieves the correct data types from metadata, and declares variables and parameters.

Here is a preview of what you will be able to do with DbFit (everything will be explained in more detail later). To call stored procedures, just create a table with the `Execute Procedure` command, put the procedure name after the command, and list your procedure parameters in second row. Put a question mark after output parameter names. Then put different combinations of inputs and expected values for output parameters into the table. The table in Figure 1.1 shows three tests for the `ConcatenateStrings` stored procedure. Notice that there are no variable declarations, no type guessing, no special code to compare values. Just the table.

Figure 1.1. Test stored procedures by just listing parameter values

The screenshot shows a web browser window titled "XpDay ConcatenationTest" with the URL "http://localhost:8085/XpDay.ConcatenationTest". The page content includes a sidebar with navigation options like "Test", "Edit", "Versions", "Properties", "Refactor", "Where", "Used", "Recent Changes", "Files", and "Search". The main content area displays the test setup and results for the "ConcatenationTest".

Set Up: `..XpDay.SetUp` [Expand All](#) | [Collapse All](#)

```
dbfit.MySqlTest
Connect localhost:root dbfit
```

CONCATENATESTRINGS JOINS TWO STRINGS AND ADDS A BLANK

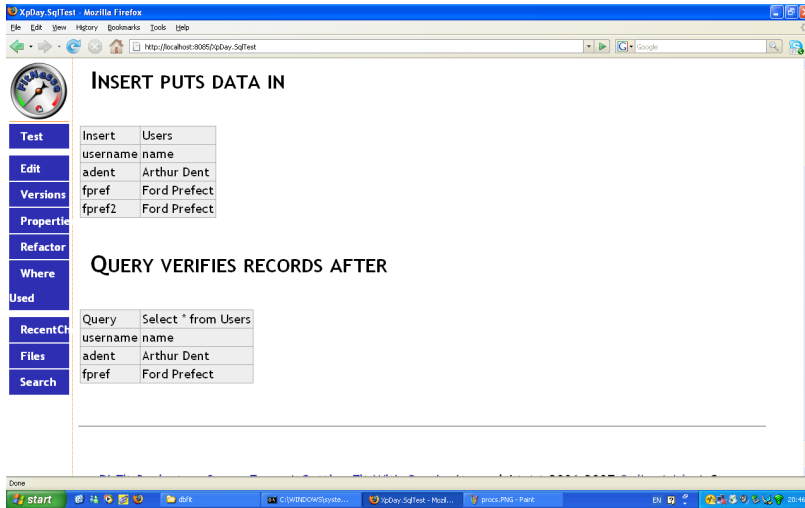
it's not really that complicated :)

Execute Procedure	ConcatenateStrings:		
first string	second string	concatenated?	
Hello	World	Hello World	
Arthur	Dent	Arthur Dent	
Ford	Prefect	Ford Prefect	

Relational data access is very similar — again using tables. The `Insert` command puts data into a table or a view. It again reads the table, looking for the column names in the second row, and data in all subsequent rows. There is again no type information or any kind of any boilerplate code. The `Query` command will execute any SQL query you specify and compare the actual results with what you specified in the table below the command. See

Figure 1.2. This is database testing in a pure relational form, very close to how you are used to thinking about database objects.

Figure 1.2. Manage data in a tabular form, like you are used to thinking about it



How this document is organised

As database developers and .NET/Java developers would use DbFit somewhat differently, the following two chapters will introduce DbFit to each of those groups. It will not harm you to read both chapters, but you might as well skip one if you want and then come back at a later time. The introduction for database developers focuses more on how to use FitNesse generally. The introduction for .NET/Java developers focuses more on how DbFit works with FIT/FitNesse and discusses integration tests.

After those introductory chapters, we review all test table types (fixtures) available in DbFit. The document ends with a list of frequently asked questions and some pointers about where to go next.

Database unit tests

This chapter introduces DbFit to database developers, and explains how to use DbFit for database unit testing. DbFit is an extension to *FitNesse*¹, so you will use the FitNesse server to manage and run DbFit tests. This chapter will give you a brief introduction to installing and using FitNesse to write and manage tests. We will also do a quick sanity check to make sure that you installed and set up everything properly, and then you can continue with Chapter 4 where you will learn what types of test tables are available and how to use them.

I will not explain how DbFit works under the hood or how it fits into the larger picture of FitNesse fixtures — if you are interested in that topic read the next chapter as well. For more information on using FitNesse, tips and tricks for test management, and information about how to include FitNesse tests into your version control and continuous build system, see my book *Test Driven .NET Development with FitNesse*².

Installing DbFit

There are two ways to run DbFit — through Java or through .NET. As a database developer, you *do not have to know Java or .NET to write and run the tests*. The only significant difference between the two implementations is that the Java and .NET versions support different databases. Microsoft SQL Server is fully supported only in the .NET version, and MySQL and Derby is supported only in the Java version. Both .NET and Java versions support Oracle and DB2.

¹<http://www.fitnesse.org>

²<http://gojko.net/fitnesse/book>

Figure 2.1. Databases supported in DbFit

	DbFit Java	DbFit .NET
Oracle	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
MySQL	<input checked="" type="checkbox"/>	
Microsoft SQL Server	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Derby (JavaDB)	<input checked="" type="checkbox"/>	
IBM DB2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

If you decide to use the Java version, you need only Java JRE 5 or later (get it from <http://java.sun.com>). To use the .NET version, you need both the Java JRE and Microsoft's .NET Framework runtime 2 or later (you should have that already installed if you are running Windows, but if you do not, get it from <http://msdn.microsoft.com/netframework>).

To install DbFit for database unit testing, I suggest that you download the `dbfit-complete` package from <http://sourceforge.net/projects/dbfit>. That package includes DbFit libraries and all required dependencies, including .NET and Java test runners for FitNesse, and the FitNesse server itself. It also contains this document in a Wiki form and lots of examples for all supported functions and databases in the `AcceptanceTests` test suites. (The package does not include Java JRE or the .NET Framework, so you'll have to download those separately). If you already know your way around FitNesse, or want to upgrade an existing installation, then you can get only the DbFit library without any dependencies by downloading either the `dbfit-dotnet-binaries` or `dbfit-java-binaries` package from the DbFit SourceForge site. In this chapter, I presume that you are using the `dbfit-complete` package.

There is no special installation procedure required — just unpack `dbfit-complete-XXX.zip` somewhere on your disk, and run `startFitnesse.bat` (or `startFitnesse.sh` on Linux). FitNesse works as a web application with its own web server. The batch file you started will try to set up FitNesse on port 8085 by default. If this port is already taken on your machine, open `startFitnesse.bat` in any editor and change 8085 to some other free port number. I use 8085 in the examples, so if you use another one, remember to enter the correct port when you try out the examples. When FitNesse starts, you should see a command window with this message:

```
FitNesse (20070619) Started...
```

```
port:      8085
root page: FitNesse.wiki.FileSystemPage at ./FitNesseRoot
logger:    none
authenticator: FitNesse.authentication.PromiscuousAuthenticator
html page factory: FitNesse.html.HtmlPageFactory
page version expiration set to 14 days.
```

Open `http://localhost:8085/` and you should see the welcome page (Figure 2.2).

FitNesse is up and running. When you want to shut it down later, just press `Ctrl+C` in the command window (or close the command window).

You might want to set up a test database to try out some examples. There are no specific requirements for anything to be in the database for DbFit to work, but you might want to create a database user for testing and grant the user privileges to connect and create resources. If you want to try out examples from the `AcceptanceTests` suite in the `dbfit-complete` package, you'll need to create some test objects as well. Scripts to create the objects can be found in the `scripts` folder of the release — Oracle, SQL Server, DB2 and MySQL scripts are provided.

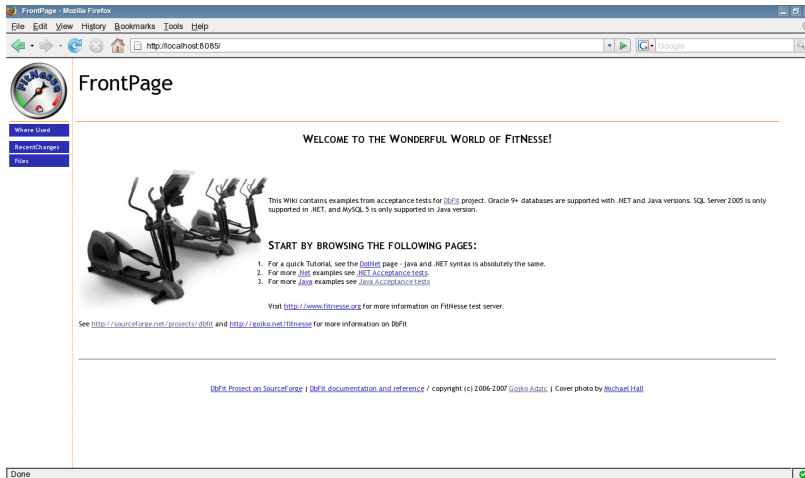


FitNesse.bat failed. What's wrong?

Read the exception from the command window. If the error mentions versions, check that you have Java 5 or 6 installed and that the correct version is being executed when you run `java.exe`. Run `java.exe -version` from a command window to see which version of Java is being executed by default. You can run FitNesse with a different Java version either by pointing to the correct JRE in the system executable path (right-click **My Computer**, select **Properties**, then go to the **Advanced** tab, click **Environment Variables**, and edit the **Path** variable), or by entering the full path to a different `java.exe` in `startFitnesse.bat`.

If the error report states that there is a security problem or the port is unavailable, enter a different port number in `startFitnesse.bat` and try again.

Figure 2.2. DbFit/FitNesse welcome page



Hello World from the database

Let's run a quick test to make sure that you have everything set up correctly and that FitNesse can connect to your test database. In doing so, we'll also explain how to manage tests with FitNesse. FitNesse is a collaborative Wiki site for building and executing tests. Tests are described by tables that contain both input values and expected results. FitNesse runs tests by reading HTML files, looking for tables, and using data in the tables to execute tests and compare results to expectations. To keep things simple for now, we'll just run a query and verify the results.

Step 1: Creating a new test page

Open <http://localhost:8085/HelloWorld> in your browser. You should see a screen telling you that there is no *HelloWorld* page and a link to create a new page. Click on the link and FitNesse opens the page editor: a big text box with several buttons. This is where we'll create our new test page. Notice that the page name is a CamelCase word. FitNesse is really strict about that. All page names have to start with a capital letter, have at least one more capital letter, and all capital letters have to be separated by at least one lowercase letter. This convention causes a lot of headaches for FitNesse newbies, but after a while you'll get used to it. Here are some good page names:

- HelloWorld
- TestFluxCapacitor

- IsPaymentWorkingCorrectly

Here are some page names that will get you in trouble:

- helloworld (no capital letters)
- Testfluxcapacitor (just one capital letter)
- isPaymentWorkingCorrectly (starts with a lowercase letter)
- TestFCapacitor (two consecutive capital letters)

Step 2: Setting up the environment

In order to load the DbFit extension into FitNesse, your test pages have to load the correct libraries. To run the .NET version of DbFit, paste the following into your test page:

```
!define COMMAND_PATTERN {%m %p}
!define TEST_RUNNER {dotnet2\FitServer.exe}
!define PATH_SEPARATOR {;}
!path dotnet2\*.dll
```

To run the Java version, paste this into your test page:

```
!path lib/*.jar
```

Step 3: Connect to the database

DbFit requires two commands to connect to the database. The first line specifies the database type (or test type), and the second defines connection properties. These two lines will typically be the first on every test page. Here is how to connect to a MySQL database:

```
!|dbfit.MySqlTest|
!|Connect|localhost|dbfit_user|password|dbfit|
```

Notice the `MySqlTest` in the first line above. That tells DbFit which type of database driver to use. For SQL Server 2005, you should use `SQLServerTest`. For MySQL use `MySqlTest`. For Oracle, use `OracleTest`. For Db2, use `DB2Test`. For Derby (JavaDB), use `DerbyTest`. If you are using an older version of SQLServer, try `SqlServer2000Test`.³ The `Connect` command has several flavours. The default is to use four parameters:

³ `SqlServer2000` is only supported in the .NET version of DbFit.

```
!|Connect|SERVICE_NAME|USER_NAME|PASSWORD|DATABASE_NAME|
```

SERVICE_NAME is the host, instance, or service name, depending on the type of driver used. For Oracle in .NET, this can be a TNS name as well (in that case, the fourth argument can be omitted). In the Java version, Oracle Thin driver is used, so the second argument should be the host name (with optional port separated by a colon), and you will have to specify the database SID as fourth argument as well.

If you want to use non-standard connection properties, or initialise your connection differently, call `Connect` with a single argument – the full ADO.NET or JDBC connection string. Here is an example:

```
!|Connect|data source=Instance;user id=User;password=Pwd;database=TestDB;|
```

You can use this feature, for example, to utilise Windows integrated authentication or to use the OCI driver for Oracle under Java. A typical database developer will not know these settings directly, but you can ask Java or .NET developers on your team to help out with the correct connection string.

There is one more option to connect to the database – store connection properties in a file on the server. You can use this, for example, if your administrators require that the database password is not shown in plain text on the test pages. See section *“Storing connection properties in a file”* on page 24 for more information.



Command structure

Notice how each command starts with an exclamation mark (!), followed by a pipe symbol (|). Command arguments are then separated by the pipe symbol as well. In FitNesse, tables are used to describe commands, tests, inputs and expected results (you will see the table when the page is saved). In the FitNesse wiki syntax, tables are described simply by separating cells with the pipe symbol. The exclamation mark before the first row of the table is optional, and tells FitNesse not to apply any smart formatting to table contents.

Step 4: Testing a simple query

Now let's write a simple query test. We will send a request to the database, pull out the result set, and compare it with our expectations. In DbFit, that

is done with the `Query` command. The second cell of the first table row, after the `Query` keyword, should contain the query we are executing. The second row then contains the result set structure — names of the columns that we want to inspect. You don't have to specify the full result set here, just the columns that are interesting for a particular test. All rows after that contain expected results. `Query` disregards result set order — if the order is important you can use `OrderedQuery`. Here is a simple MySQL query:

```
!|Query| select 'test' as x|
|x|
|test|
```

The same syntax should work for `SQLServer`. For `Oracle`, use this table:

```
!|Query| select 'test' as x from dual|
|x|
|test|
```

Step 5: Running the test

Now, click *Save*. FitNesse will create a new page and display it in your browser. Next, you have to tell FitNesse that this is a test page (Figure 2.3) — click the *Properties* button on the left, check the *Test* check-box and then click *Save Properties* (Figure 2.4).

Figure 2.3. Our new page is stored in FitNesse

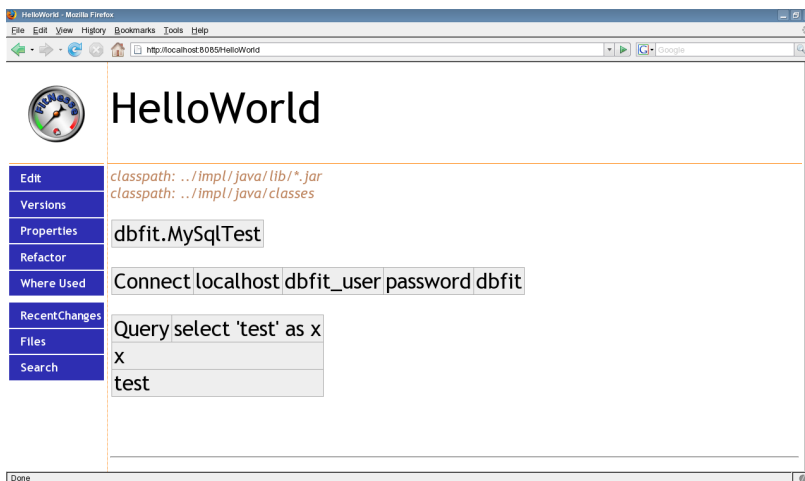
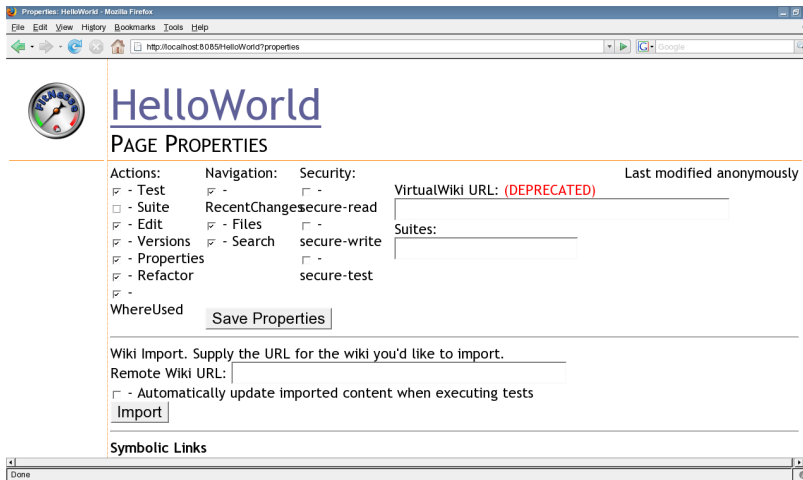
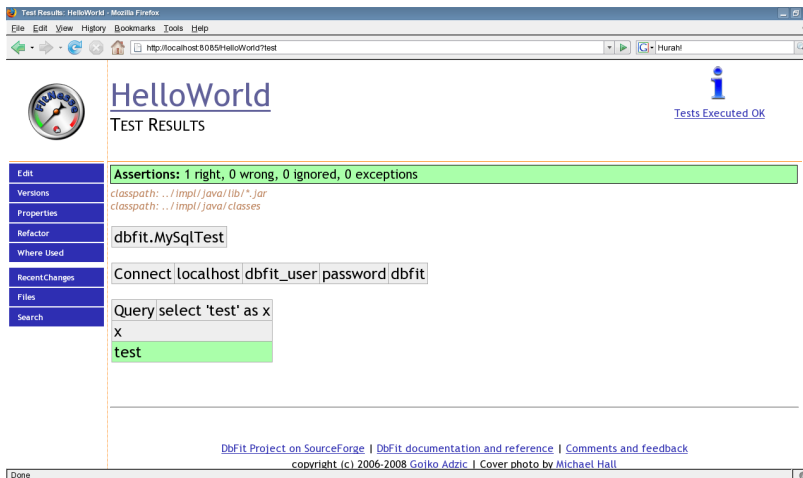


Figure 2.4. Use page properties to tell FitNesse that the page contains a test



Page properties define what the user can do with the page — more precisely, which buttons will be offered in the left-hand menu. When the page reloads, you will notice a new button on the left: *Test*. Click it to make FitNesse run the test. You should see a page similar to Figure 2.5 telling you that the test passed.

Figure 2.5. Our first test passed. Hurrah!



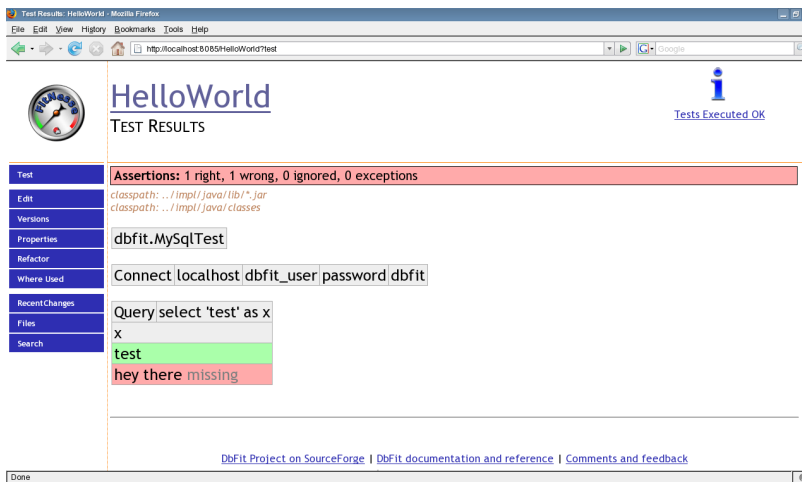
If the test result is green, your setup works, you can connect to the database, and we can continue. If the test was result yellow, something is wrong. Read the error messages to check if the database connection properties are wrong. Double-check that you have entered correct paths to DbFit library files. The paths are relative to the folder in which you started FitNesse, so if you use the `dbfit-complete` package, Java libraries will be in the `lib` folder, and .NET libraries will be in the `dotnet2` folder. If the test result did not contain a table at all, you probably wanted to use the .NET test runner but specified the wrong path in the `TEST_RUNNER` variable.

Traffic lights

Click the *Edit* button on the left and add another row to the expected results, so that you can see how FitNesse prints the results when a test fails.

FitNesse is a traffic light which can tell you whether your code is ready to be released or not. If a test fails, the light turns red, and FitNesse will show both expected and actual results. In the case of queries, it will compare the contents of the FitNesse table with what actually came out from the database, and print out missing or surplus rows (Figure 2.6).

Figure 2.6. FitNesse clearly shows what's wrong



A note on transaction management

To make tests automatically repeatable, DbFit executes each test in a separate transaction, and rolls back on the end of the test. You can commit or rollback manually as well if you want to persist the effects of a test (this will be explained in Chapter 4). Note, however, that it is a very good practice to make tests repeatable. If you intend to persist anything, make sure that unique constraints do not prevent the test from repeating.

Managing Wiki content

Those are the basics of FitNesse. Tables describe commands, inputs and expected outputs. You can use DbFit commands in FitNesse tables to do things such as execute stored procedures, and execute and compare queries. The rest of this chapter will focus on managing content in FitNesse — see Chapter 4 for detailed information on all available DbFit commands and their table syntax.

Organising pages

In FitNesse, subwikis are the equivalent of web folders, database schemas or C# namespaces. They can be used to manage related pages more easily as a group. Instead of a slash /, which is the separator in a web folder name, the dot symbol . is used to separate levels of hierarchy in FitNesse.

For example, URL *PurchaseTicketSuite.NotEnoughFunds* leads to the *NotEnoughFunds* page in the *PurchaseTicketSuite* subwiki. Just as a page can be turned into a test via the *Properties* button, a subwiki can be turned into a test suite. A test suite is a group of related tests that allows us to control their common properties from one place.

To create a subwiki (and a test suite), first create the main subwiki page. In the previous example, that would be *PurchaseTicketSuite*. You can put the environment definitions from section “*Step 2: Setting up the environment*” on page 9 into that page, so that you do not have to repeat them for individual tests. Instead of defining any test tables in that page, just enter ! contents -R as the page content. This automatically builds and shows a table of contents for the subwiki. As the subwiki is probably empty now, the page will be empty, but as you add subpages to it, they will automatically appear in the table of contents. Click *Properties* on the left, and mark the page as a *Suite* — not as a *Test*.

Next, create individual test pages under that subwiki, and mark them as tests in page properties. You will be able to run individual tests by clicking on the *Test* button. You can also run all tests in the suite together by clicking on the *Suite* button when viewing the main suite page.

There are two special pages for a test suite: *SetUp* and *TearDown*. If they exist, those two pages are executed before and after every test. You can use these pages to extract common preparation and clean-up steps for all related tests and manage them together. When using DbFit, it is a good practice to include the database connection in the *SetUp* page:

```
!|dbfit.MySqlTest|
!|Connect|localhost|dbfit_user|password|dbfit|
```

In Chapter 4 you will learn how to insert data and execute procedures, and you can include those steps as well in the *SetUp* if they are common for a group of tests.

A subwiki hierarchy is considered a namespace for links. So, for example, link *BasicCase* from the *PurchaseTicketSuite.SetUp* page leads directly to *PurchaseTicketSuite.BasicCase*. However, the main suite page *PurchaseTicketSuite* is not in the same namespace, but one level above. If you put a link named *BasicCase* in the main suite page, it will lead to a top-level *BasicCase* page. To reach a subpage, prefix the name with a caret (^*BasicCase*). In FitNesse release 20070619, symbols < and > are also used to point one level up or down in the hierarchy. To go to the top level, prefix a page name with a dot. So the link to *.FrontPage* always leads to the home page of the site.

Writing tests in Excel/Word

Although FitNesse Wiki syntax is really simple, you do not have to use it to write scripts. You can write your tables in Excel (or almost any other spreadsheet program), and then just copy them into the FitNesse page editor. Clipboard automatically picks up data from most spreadsheet programs in tab-separated format, which can be directly converted to FitNesse with the *Spreadsheet to FitNesse* button that is available when editing a page. If your spreadsheet program behaves differently, it should be able to export tab-separated files.

You can also convert a FitNesse table to tab-separated data with the *FitNesse to Spreadsheet* button in the page editor, and then copy that into Excel for editing.

Formatting text

FitNesse is a Wiki – a relatively free-form content management system which allows users to build pages and link them together. Instead of using HTML directly, Wikis use a special markup syntax. You have already seen pipes (|) used to create tables. Here are a few more interesting markup symbols:

- !1 Apply Heading 1 style to the rest of the line.
- !2 Apply Heading 2 style to the rest of the line.
- !3 Apply Heading 3 style to the rest of the line.
- !c Align to centre.
- ---- Horizontal line (4 or more dashes).
- !img url Display image from url.
- '''text''' Bold – three single quotes enclosing text on each side.
- ''text'' Italics – two single quotes enclosing text on each side.
- # Comment – ignore the rest of the line.

FitNesse automatically recognises most links and builds proper HTML code for them – external links should just begin with `http://` and internal links are built from CamelCase words (beginning with a single capital letter and containing at least one more capital letter). If the url ends with `.gif` or `.jpg`, FitNesse will automatically replace the url with the image. You can create additional links yourself by putting `[[label]][url]` anywhere on the page. This can be used to create links which FitNesse does not recognise (if the word is not in CamelCase), or to change the default label for the link.

See <http://FitNesse.org/FitNesse.MarkupLanguageReference> for a detailed reference of the Wiki markup language used in FitNesse.

Preventing unwanted formatting

FitNesse does a lot of formatting on its own, most of the times guessing the right thing to do. However, in some cases you explicitly want to prevent “smart” formatting. For example, formatting should not be applied to code examples, class names, and generally to test tables.

You already know that you can use an exclamation mark (!) to prevent any smart formatting of table contents. However, some basic formatting (such as variable replacement) will still be done. If you want to prevent all formatting, enclose the text into `!-` and `-!`. To prevent FitNesse from parsing and

formatting large blocks of text, enclose those blocks into three curly braces ({{{ and }}}) – you should typically do this with code examples, but you can use that trick to enclose any pre-formatted block of text.

A note on flow and standalone modes before we continue

If you just want to use DbFit for database tests, and not integrate them with .NET/Java tests, then you can skip the next chapter. However, you will see references to flow and standalone mode in Chapter 4 which might confuse you, so here is a short explanation.

If you connect to the database as suggested in this chapter, using `SqlServerTest`, `MySQLTest` or a similar table, then you are using flow mode. In that case, the test type you choose is controlling the whole test page. The standalone mode is used if you need to mix DbFit tables with other .NET and Java tables, and transaction and database environment control should come from outside of DbFit. In that case, a `DatabaseEnvironment` table is used to connect to the database.

DbFit for Integration tests

This chapter introduces DbFit to Java and .NET developers, and explains how to utilise DbFit to set up, modify and verify the data layer in .NET or Java FIT/FitNesse tests. I will not explain how to use or set-up and use FitNesse here, as I would expect you to already know that. If not, see <http://www.fitnesses.org> or my book *Test Driven .NET Development with FitNesse*¹.

Why use DbFit for integration tests?

Even for projects where the database is used just as a simple persistence layer, it has an impact on automated tests. Integration and acceptance tests should run in an environment as close to the production environment as possible, which today often involves a database. This means that data needs to be set up before the test, cleaned up after, and that changes to data may need to be verified in the database. Writing code to do this in Java and .NET is not rocket science, but it is dull and error-prone, and I'd rather avoid it.

DbFit is an extension library to FIT that enables tests to be executed directly against a database. DbFit fixtures take care of all the database integration plumbing, including automated transaction management, parameter declarations and selecting the right column or parameter type. Because of this, it is easier to write database tests with DbFit than it is to implement manual validations.

Installing DbFit

To use DbFit fixtures in your tests, download the *dbfit-dotnet-binaries* package or *dbfit-java-binaries* package from <http://sourceforge.net/projects/dbfit>, depending on which platform you are using for development. The features and fixtures are more or less the same, but they support different databases. DB2, Oracle and Microsoft SQL Server are supported in both .NET and Java versions. Derby (JavaDB) and MySQL are supported only in the Java version. Microsoft JDBC driver is not redistributable, so you'll have to download it from *their web site*² and deploy in the same folder as the dbfit JAR archive.

¹<http://gojko.net/fitnesses/book>

²<http://www.microsoft.com/downloads/details.aspx?>

FamilyId=C47053EB-3B64-4794-950D-81E1EC91C1BA&displaylang=en

Figure 3.1. Databases supported in DbFit

	DbFit Java	DbFit .NET
Oracle	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
MySQL	<input checked="" type="checkbox"/>	
Microsoft SQL Server	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Derby (JavaDB)	<input checked="" type="checkbox"/>	
IBM DB2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

To install DbFit, just unpack the ZIP and copy `dbfit.dll` or `dbfit-XXX.jar` into your fixture path. All the test fixtures that manipulate database objects (which will be explained in the following chapter) are in the `dbfit.fixture` package/namespace. Flow-style fixtures that provide automated transaction control are in the `dbfit` package/namespace. There are also some utility classes and fixtures in the `dbfit.util` namespace/package.

All FitNesse features, such as symbols and markup variables, work with DbFit fixtures as well. In addition to that, .NET symbol syntax (using `>>` and `<<` to access symbols directly in cells) and the `fail` keyword work in the Java version with DbFit fixtures.

DbFit aims to use the same fixture classes and table structure/syntax for all supported databases, but still provide full access to vendor-specific database features. That is why all database-flavour specific information is abstracted into a `DBEnvironment` instance (`IDBEnvironment` in .NET). This class is used to connect to the database, fetch meta-data like procedure parameters or column types, prepare and parse native queries and serve as a factory for related object types. A database environment instance is used as configuration for all DbFit fixtures. That is why all test fixtures have two constructors – one default, which will use the default environment, and one which allows the caller to specify the environment.

This is important if you want to integrate your own fixtures into the same transaction as DbFit fixtures – you can either use the active environment object from DbFit to retrieve the JDBC/ADO.NET connection object or implement your own `DbEnvironment` instance and pass that to DbFit fixtures. If the flow mode is used (modes will be explained shortly), the environment is stored in a protected field of the `DatabaseTest` class, so you can easily access it by extending that class (or the appropriate subclass like `OracleTest`). In

standalone mode, the active environment can be set or retrieved by using `DbEnvironmentFactory` class.

If you want to extend `DbFit` to support a new type of database, then you just need to implement the `DbEnvironment` interface. `AbstractDbEnvironment` has some stubs that are re-used in all current implementations, so it may be a good idea to start looking at that first.

Why not use generic ADO.NET/JDBC interfaces?

Although database interfaces in .NET and Java are in theory database-independent, several key differences in SQL syntax and in driver implementation make it virtually impossible to support effective testing with a completely generic approach. The main differences are:

1. Reading table columns and mapping their datatypes to host types
2. Reading stored procedure/function parameters and mapping datatypes to host types
3. Extracting parameter names from ad-hoc SQL queries
4. Building plumbing SQL commands like insert/returning primary key
5. Instantiating objects such as DB connection, commands, or parameters

To be useful, `DbFit` needs to provide access to vendor-specific features as well as to generic features, so the database-connectivity layer had to be abstracted one level more.

Connecting to the database

`DbFit` fixtures can work in two modes:

- **Flow mode:** a `DatabaseTest` fixture controls the whole page and coordinates testing. You can use other fixtures as well, but no other fixture can take over flow mode processing. In flow mode, `DbFit` automatically rolls back the current transaction at the end to make tests repeatable, and provides some additional features such as inspections of stored procedure error results.
- **Standalone:** you can use individual fixtures without a `DatabaseTest` coordinating the whole page. In this case, you are responsible for transaction management. This enables you to have more control over the database testing process, and even supply your own database connection to make sure that Java/.NET integration tests are running in the same transaction.

The mode in which you are using DbFit fixtures affects how you connect to the database and how the connection is shared between fixtures. Note that in flow mode, the methods of the `DatabaseTest` class have the same names as the fixtures they relate to. If you import the namespace/package for standalone fixtures, the table syntax in both modes is the same in most cases. The flow mode is there to provide you with better isolation and automated transaction management. Standalone mode is there to allow you to have greater control over the database calls and to allow you to embed DbFit fixtures in your flow-style tests.



Which mode should I use?

If you can, use flow mode. It gives you automatic transaction management and some other shortcuts. If your test relies on some other fixture controlling the page in flow mode, use standalone fixtures. The syntax is, in most cases, the same.

Connecting in flow mode

In flow mode, the current database connection is kept in a protected field of the `DatabaseTest` instance. `SqlServerTest` is a subclass of `DatabaseTest` that just initialises it to work with `SqlServer 2005`. Similarly, `SqlServer2000Test` initialises `DatabaseTest` such that it will work with `SqlServer 2000`. `OracleTest` works with `Oracle` databases, `DB2Test` with `IBM DB2`, `DerbyTest` with `Derby (JavaDB)` and `MySQLTest` connects to `MySQL`. All of these fixture classes are in the `dbfit` namespace/package.

Use the `Connect` method to initialise the database connection. Pass the server (optionally followed by the instance name), username, password, and the database name as arguments. This is how I connect to a `SqlServer 2005 Express`³ instance on my laptop:

```
!|dbfit.SqlServerTest|  
!|Connect|LAPTOP\SQLEXPRESS|FitNesseUser|Password|TestDB|
```

If you are connecting to a default database, you can omit the fourth parameter. For the .NET version, you can do this for Oracle, because the second argument is the TNS Name. The Java version of DbFit uses the Thin driver for Oracle, and expects the second argument to be the host name (with an optional port) and the fourth argument to be the service identifier.

³ free version of `SqlServer 2005` for developers. See <http://www.microsoft.com/sql/editions/express/>.

If you want to use non-standard connection properties, or initialise your connection differently, call `Connect` with a single argument — the full ADO.NET or JDBC connection string. Here is an example:

```
|Connect|data source=Instance;user id=User;password=Pwd;database=TestDB;|
```

You can use this feature, for example, to utilise Windows integrated authentication or to use the OCI driver for Oracle under Java.

For flow mode to work correctly, the `SqlServerTest` fixture must be the first one on the page — not even `import` can be before it. This is why we explicitly specify the namespace.

DbFit does not require any special database privileges other than what is required to execute the commands that you specify directly. DbFit will attempt to read the schema meta-data, but select access to those tables should be allowed in most cases by default. For a detailed list of meta-data tables accessed, see section “*Does DbFit require any special database privileges?*” on page 50.



Fixture class is more important than connection string

Connection strings in both .NET and Java may allow you to specify the type of database provider — effectively the kind of database you are connecting to. Theoretically you could instantiate a SQL Server test fixture and pass an Oracle connection string, but this will not work in practice. Test fixture already contains database-specific logic, so it will not work with an incompatible connection string.

Connecting in standalone mode

In standalone mode, the connection properties are stored in the public `DefaultEnvironment` singleton field inside `dbfit.DbEnvironmentFactory`. You can initialise it from your own fixtures if you want to pass an existing database connection (to make sure that your .NET tests are using the same transaction as DbFit fixtures). From FitNesse pages, you can use the `DatabaseEnvironment` fixture from the `dbfit.fixture` package to define the connection. To change the default environment (or initialise it for the first time), pass the new environment type as the first argument to the fixture. Environment type values are as follows:

- `SqlServer 2005` — `SQLSERVER`

- Earlier versions of SQL Server — SQLSERVER2000
- Oracle — ORACLE
- MySQL — MYSQL
- DB2 — DB2
- Derby (JavaDB) — DERBY

`DatabaseEnvironment` is a `SequenceFixture` that wraps the `DefaultEnvironment` singleton as a system under test, so that you can then call all of its public methods directly — including the `Connect` method explained earlier.

```
|import|
|dbfit.fixture|

!|DatabaseEnvironment|SQLSERVER|
|Connect|LAPTOP\SQLEXPRESS|FitNesseUser|Password|TestDB|
```

Notice that there is no space between `DatabaseEnvironment` and `Connect` — they have to be in the same table. Because we are not using flow mode, we can use the `import` fixture as well. Most DbFit fixtures are in the `dbfit.fixture` namespace, so it is a good practice to include this namespace.

Storing connection properties in a file

You can also store connection properties in a file, then initialise the connection using the `ConnectUsingFile` method. This allows you to hide actual database usernames and passwords from FitNesse users, should you need to do so.

`ConnectUsingFile` has only one argument — the path of the file on the server, either absolute or relative to the folder from which you started FitNesse (the one containing `run.bat`). The connection properties file is a plain text file, containing key/value pairs separated by the equals symbol (=). Lines starting with a hash (#) are ignored. Use the following keys (they care case-sensitive):

1. `service` — service name. In the previous example, it was `LAPTOP\SQLEXPRESS`.
2. `username` — username to connect to the database. In the previous example, it was `FitNesseUser`.
3. `password` — password to connect to the database. In the previous example, it was `Password`.
4. `database` — optional fourth argument, allowing you to choose the active database. In the previous example, it was `TestDB`.

5. `connection-string` — alternative to the four previous parameters, this allows you to specify the full connection string. This parameter should not be mixed with any of the four other keys. Use either the full string or specify individual properties.

Here is an example:

```
# DBFit connection properties file
#
#1) Either specify full connection string
#connection-string=
#
#2) OR specify service, username and password as separate properties
service=localhost
username=root
password=
#optionally specify a database name
database=dbfit
```

Transaction management

In flow mode, the current transaction is automatically rolled back at the end of the page. If you want to commit it to make changes permanent, put the `Commit` table on the page. There are no arguments or additional parameters — the table contents contain just this one word. Likewise, you can roll back manually in your test using the `Rollback` table.

In standalone mode, use the `DatabaseEnvironment` fixture again, but do not specify a fixture argument. This tells the `DatabaseEnvironment` to use the current default database connection, without attempting to initialise it. Call `Commit` or `Rollback` in the second row.

```
!|DatabaseEnvironment|
|rollback|
```

It is a very good idea to put this table in a *TearDown* page for your test suite when you use standalone DbFit fixtures. This will make sure that your tests are repeatable.



Can I use both modes in the same test suite?

Yes, in different tests. Note that the imported namespace may give you some problems in flow mode. If you want to mix and match, then either do not import the `dbfit.fixture` namespace for standalone tests, or use the utility `Export fixture` to cancel the namespace import after the standalone test.

```
!|dbfit.util.Export|  
|dbfit.fixture|
```


Command reference

This chapter will introduce and explain all available DbFit commands (tables, fixture types). To see more examples, get the `dbfit-complete` package and run the FitNesse wiki from that package. All commands, for all supported database, are shown in the *AcceptanceTests* subwiki. Examples in this chapter are, unless stated otherwise, written for MySQL. The *AcceptanceTests* subwiki contains equivalent examples for all other supported databases.

Set-up Script

Here is a simple script to create the objects required for examples in this chapter (you'll also find it in the `scripts` directory of the `dbfit-complete` package):

```
create database dbfit;

grant all privileges on dbfit.* to dfctest@localhost identified by 'dfctest';

grant all privileges on dbfit.* to dfctest@127.0.0.1 identified by 'dfctest';

grant all privileges on dbfit.* to dbfit_user@localhost identified by
'password';

grant all privileges on dbfit.* to dbfit_user@127.0.0.1 identified by
'password';

grant select on mysql.* to dbfit_user;

flush privileges;

use dbfit;

create table users(name varchar(50) unique, username varchar(50), userid int
auto_increment primary key) type=InnoDB;

CREATE PROCEDURE ConcatenateStrings (IN firststring varchar(100), IN
secondstring varchar(100), OUT concatenated varchar(200)) set concatenated =
concat(firststring , concat( ' ' , secondstring ));

create procedure CalcLength(IN name varchar(100), OUT strlength int) set
strlength =length(name);
```

```
CREATE FUNCTION ConcatenateF (firststring VARCHAR(100), secondstring
varchar(100)) RETURNS VARCHAR(200) RETURN CONCAT(firststring,' ',secondstring);

create procedure makeuser() insert into users (name,username) values
('user1','fromproc');

create procedure createuser(IN newname varchar(100), IN newusername
varchar(100)) insert into users (name,username) values (newname, newusername);

create procedure Multiply(IN factor int, INOUT val int) set val =val*factor;
```

Working with parameters

DbFit enables you to use Fixture symbols as global variables during test execution, to store or read intermediate results. The .NET syntax to access symbols (>>parameter to store a value and <<parameter to read the value) is supported in both .NET and Java versions. In addition, you can use the `Set Parameter` command to explicitly set a parameter value to a string.

```
|Set parameter|username|arthur|
```

DbFit is type sensitive, which means that comparing strings to numbers, even if both have the value 11, will fail the test. Most databases will allow you to pass strings into numeric arguments, but if you get an error that a value is different than expected and it looks the same, it is most likely due to a wrong type conversion. Keep that in mind when using `Set parameter`. A good practice to avoid type problems is to read out parameter values from a query. This will be explained in detail soon.

You can also use the keyword `NULL` to set a parameter value to `NULL`.

Query

Query is similar to traditional `FIT RowFixture`, but uses SQL Query results. You should specify query as the first fixture parameter, after the `Query` command. The second table row contains column names, and all subsequent rows contain data for the expected results. You do not have to list all columns in the result set — just the ones that you are interested in testing.

```
!|Query| select 'test' as x|
|x|
|test|
```

Ordering and row matching

Query ignores row order by default. In flow mode, the `Ordered Query` command provides order checking.

Partial key matching is supported, like in `RowFixture`: columns with a question mark in their name are not used to match rows, just for value comparisons. You can use this to get better error reports in case of failed tests. It is a good practice to put a question mark after all column names that are **not** part of the primary key.

Rows in the actual result set and FitNesse table are matched from top to bottom, looking for equal values in all cells that are not marked with a question mark. If there are no key columns, then the first row will be taken as a match (which effectively acts as the `Ordered Query`). All non-key columns are used for value comparisons, not for deciding whether or not a row exists in the result set.

`Query` will report any rows that exist in the actual result set and not in the FitNesse table (those will be marked as *surplus*), rows that exist in the FitNesse table but not in the actual result set (marked as *missing*). All matched rows are then checked for values in columns, and any differences will be reported in individual cells. You can use a special `fail[expected value]` syntax to invert the test, making it fail if a certain value appears in the row:

This will fail because the order is wrong

```
|Ordered Query|SELECT n FROM ( SELECT 1 as n union select 2 union select 3 ) x |
|n|
|fail[2]|
|fail[1]|
|3|
```

This will pass because the order is correct

```
|Ordered Query|SELECT n FROM ( SELECT 1 as n union select 2 union select 3 ) x|
|n|
|1|
|2|
|3|
```

Using parameters

You can use query parameters (DB-specific syntax is supported, eg. @paramname for SQLServer and MySQL, and :paramname for Oracle). Corresponding fixture symbol values are automatically used for named query parameters.

```
|Set Parameter|depth|3|  
  
|Query|SELECT n FROM ( SELECT 1 as n union select 2 union select 3 union select  
4) x where n<@depth |  
|n|  
|2|  
|1|
```

You can store elements of the result set into parameters — to re-use them later in other queries and stored procedures. Use >>parameter to store a cell value into a parameter. You can also use <<parameter to read a cell value from a parameter (for comparisons, for example).

If you use the query just to read out stuff into parameters, then make sure to mark the columns with the question mark to avoid row matching. There will be nothing to match the rows with in this case, so a proper comparison would fail.

```
!|query|select now() as currd|  
|currd?|  
|>>tsevt|
```

To test for an empty query, you still need to specify the second row (result set structure), but don't supply any data rows.

Avoiding parameter mapping

If you want to prevent DbFit from mapping parameters to bind variables (eg to execute a stored procedure definition that contains the @ symbol in Sql Server), disable bind symbols option before running the query.

```
|set option|bind symbols|false|  
  
|execute| insert into users (name, username) values ('@hey','uuu')|  
  
|query|select * from users|  
|name|username|
```

```
|@hey|uuu|
```

Remember to re-enable the option after the query is executed. You can use the same trick with the Execute command.

Multi-line queries and special characters

You can use multi-line queries by enclosing them into `!-` and `-!`. This will also prevent any special character formatting. This trick can also be used with Oracle to prevent the concatenation operator `||` from being treated as a FitNesse cell boundary.

```
|Ordered Query|!-
select n from (
  select 1 as n union
  select 2 union
  select 3)
x
-!|
|n|
|1|
|2|
|3|
```

Working with padded chars

Some databases treat `CHAR` type as fixed length and fill content up to the specified length with spaces. FitNesse strips trailing spaces by default from cell contents, which makes it hard to compare `CHAR` types. DbFit provides a workaround for this, that must be enabled manually since it modifies standard string parsing. To enable this option, include the following table in your tests:

```
|set option|fixed length string parsing|true|
```

After that, you can enclose strings into single-quotes ('my string') and put trailing spaces before the closing quote. This allows you to ensure that the correct length of the string is used for comparisons. Here is an example (this example is for SQL Server, since MySQL strips trailing spaces):

```
!3 use fixed string length parsing to test blank-padded chars

|Execute|Create table datatypeetest (s1 char(10), s2 nchar(10))|

|set option|fixed length string parsing|true|
```

```
|insert|datatypeest|
|s1|s2|
|testch|testnch|

direct comparison will fail

|query|select * from datatypeest| | |
|s1?|s2?|
|fail|[testch]|fail|[testnch]|

use single quotes to pad to appropriate length

|query|select * from datatypeest|
|s1?|s2?|
|'testch  '| 'testnch  '|
```

Insert

Insert is the database equivalent of FitLibrary SetupFixture — it builds an insert command from the parameters in a data table (and executes the insert once for each row of the table). The view or table name is given as the first fixture parameter. The second row contains column names, and all subsequent rows contain data to be inserted.

```
|Execute|Create table Test_DBFit(name varchar(50), luckyNumber int)|

|Insert|Test_DBFit|
|name|luckyNumber|
|pera|1|
|nuja|2|
|nnn|3|

|Query|Select * from Test_DBFit|
|name|lucky Number|
|pera|1|
|nuja|2|
|nnn|3|

|Execute|Drop table Test_DBFit|
```

Storing auto-generated values

Columns with a question mark are used as outputs. When an output column is used, it will contain the value of the column in the new record. This is

especially handy for retrieving an auto-generated primary key. For Oracle, this works regardless of whether the column was actually the ID or something else populated with a trigger. For MySQL and SQL Server, only single-column actual primary keys can be returned. The only thing that makes sense to do at this point is to store values of the output cells into variables.

```
!3 Use ? to mark columns that should return values
```

```
!|Insert|users|  
|username|name|userid?|  
|pera|Petar Detlic|>>pera|  
|Mika|Mitar Miric|>>mika|  
|Zeka|Dusko Dugousko|>>zeka|  
|DevNull|null|>>n11|
```

```
!3 Confirm that IDs are the same as in the database
```

```
!|Ordered Query|Select * from users|  
|username|name|userid|  
|pera|Petar Detlic|<<pera|  
|Mika|Mitar Miric|<<mika|  
|Zeka|Dusko Dugousko|<<zeka|  
|DevNull|null|<<n11|
```

```
!3 Stored values can be used in queries directly
```

```
|Query|Select * from users where userid=@zeka| |
|username|name|userid|  
|Zeka|Dusko Dugousko|<<zeka|
```

When the test runs, you will see actual values being stored into variables (Figure 4.1).

Figure 4.1. Insert can return auto-generated keys

USE ? TO MARK COLUMNS THAT SHOULD RETURN VALUES

Insert	users		
username	name	userid?	
pera	Petar Detic	>>pera = 47	
Mika	Mitar Miric	>>mika = 48	
Zeka	Dusko Dugousko	>>zeka = 49	
DevNull	null	>>nll = 50	

CONFIRM THAT IDS ARE THE SAME AS IN THE DATABASE

Ordered Query Select * from users

username	name	userid
pera	Petar Detic	<<pera = 47
Mika	Mitar Miric	<<mika = 48
Zeka	Dusko Dugousko	<<zeka = 49
DevNull	null	<<nll = 50

STORED VALUES CAN BE USED IN QUERIES DIRECTLY

Query	Select * from users where userid=@zeka	
username	name	userid
Zeka	Dusko Dugousko	<<zeka = 49

Update

Update allows you to quickly script data updates. It builds the update command from the parameters in a data table and executes the update once for each row of the table. Columns ending with = are used to update records (cell specifies new data value). Columns without = on the end are used to select rows (cell specifies expected column value for the select part of update command). The view or table name is given as the first fixture parameter. The second row contains column names, and all subsequent rows contain data to be updated or queried. This example updates the `username` column where the name matches `arthur dent`.

```
|insert|users|
|name|username|
|arthur dent|adent|
|ford prefect|fpref|
|zaphod beebblebrox|zaphod|

|update|users| |
|username|=|name|
|adent2|arthur dent|

|query|select * from users|
|name|username|
|arthur dent|adent2|
```



```
|ford prefect|fpref|
|zaphod beebblebrox|zaphod|
```

You can use multiple columns for both updating and selecting, and even use the same column for both operations. You can also use parameters — eg. <<paramname — in any cell.

Execute Procedure

ExecuteProcedure is the equivalent of ColumnFixture. It executes a stored procedure or function for each row of data table, binding input/output parameters to columns. The procedure name should be given as the first fixture parameter. The second row should contain parameter names (output parameters followed by a question mark). All subsequent rows are data rows, containing input parameter values and expected values of output parameters. Parameter order or case is not important, you can even insert blanks and split names into several words to make the test page more readable.

```
!3 execute procedure allows multiple parameters, with blanks in names
!|Execute Procedure|ConcatenateStrings|
|first string|second string|concatenated?|
|Hello|World|Hello World|
|Ford|Prefect|Ford Prefect|
```

You can store any output value into a parameter with the >> syntax or send current parameter values to procedure using << syntax.

To use IN/OUT parameters, you'll need to specify the parameter twice. Once without the question mark, when it is used as the input; and one with the question mark when it is used as output.

```
!3 IN/OUT params need to be specified twice
|execute procedure|Multiply| |
|factor|val|val?|
|5|10|50|
```

If the procedure has no output parameters, then the Execute Procedure command has no effect on the outcome of the test — unless an error occurs during processing. If the procedure has output parameters, then those values are compared to expectations specified in the FitNesse table, and are used to determine the outcome of the test.

For the case where no parameters are passed to function/procedure, Execute Procedure can be specified with just one row (without a row for column header names).

```
!3 If there are no parameters, Execute Procedure needs just one row

!|Execute Procedure|MakeUser|

|query|select * from users|
|name|username|
|user1|fromproc|
```

Calling Functions

If a function is getting called, then a column containing just the question mark is used for function results.

```
!3 Stored functions are treated like procs - just put ? in the result column
header

!|Execute Procedure|ConcatenateF|
|first string|second string|?|
|Hello|World|Hello World|
|Ford|Prefect|Ford Prefect|

!3 ? does not have to appear on the end (although it is a good practice to put
it there)

!|Execute Procedure|ConcatenateF|
|second string|?|first string|
|World|Hello World|Hello|
|Prefect|Ford Prefect|Ford|
```

Expecting exceptions

In flow mode, this command can also be used to check for exceptions during processing. Normally, the test would fail if a database exception occurs. However, if you want to test a boundary condition that should cause an exception, then use `Execute procedure expect exception variant` of the `Execute procedure` command. You can even specify an optional exception code as the third argument. If no exception code is specified, then the test will pass if any error occurs for each data row. If the third argument is specified, then the actual error code is also taken into consideration for failing the test.

```
!3 create a user so that subsequent inserts would fail
```

```

!|execute procedure|createuser|
|new name|new username|
|arthur dent|adent|

!3 check for any error

!|execute procedure expect exception|createuser|
|new name|new username|
|arthur dent|adent|

!3 check for a specific error code

!|execute procedure expect exception|createuser|1062|
|new name|new username|
|arthur dent|adent|

```

For detailed exception code verifications to work with SQL Server, user message must be registered for that particular error code, or SQL Server throws a generic error code outside the database. Here is how you can declare your error code:

```

sp_addmessage @msgnum = 53120, @severity=1, @msgtext = 'test user defined error
msg'

```

Execute procedure expect exception **variant is not directly available as a separate table in standalone mode. If you need this functionality in standalone mode, then extend the ExecuteProcedure fixture and call the appropriate constructor. That class has several constructors for exceptions and error codes.**

Execute

Execute executes any SQL statement. The statement is specified as the first fixture parameter. There are no additional rows required for this command.

You can use query parameters in the DB-specific syntax (eg. @paramname for SQLServer and MySQL, and :paramname for Oracle). Currently, all parameters are used as inputs, and there is no option to persist any statement outputs.

```

!3 to execute statements, use the 'execute' command

|Execute|Create table Test_DBFit(name varchar(50), luckyNumber int)|

|Execute|Insert into Test_DBFit values ('Obi Wan',80)|

```

```
|Set parameter|name|Darth Maul|  
  
|Execute|Insert into Test_DBFit values (@name,10)|  
  
|Query|Select * from Test_DBFit|  
|Name|Lucky Number|  
|Darth Maul|10|  
|Obi Wan|80|  
  
|Execute|Drop table Test_DBFit|
```

Inspect

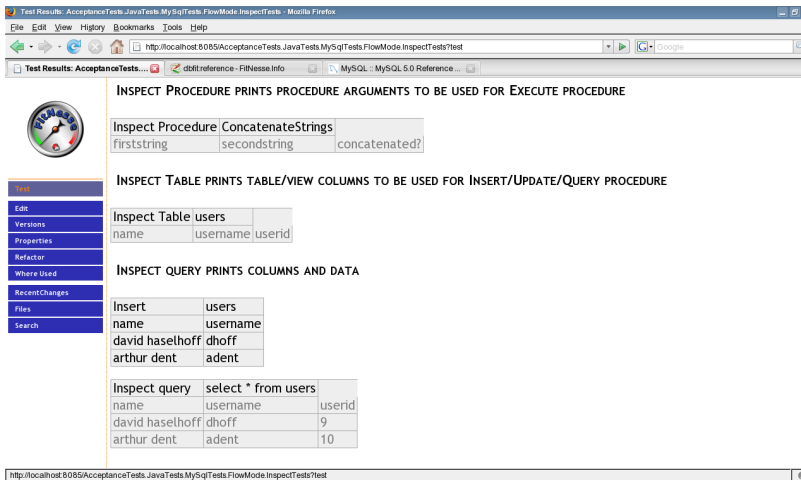
Inspect is a utility fixture class used to quickly extract meta-data information from the database, and print it out in a form which can be easily converted into a test. It can work in three modes: Query, Table or Procedure. In the Query mode, it expects a full query as argument (bound variables are supported), and prints out both the result structure and result data. In the Table mode, it expects a table or view name as an argument and prints out the table or view column names (without actual data, just the structure). In Procedure mode, it expects a procedure name as an argument and prints out the procedure parameter names. These tables can be easily converted into Query, Execute Procedure, Insert or Update tables.

In flow mode, these three inspections are available as individual commands Inspect query, Inspect table and Inspect procedure. In standalone mode, you can extend the Inspect fixture and set the appropriate mode manually while calling the constructor.

```
!3 Inspect Procedure prints procedure arguments to be used for Execute procedure  
  
!|Inspect Procedure|ConcatenateStrings|  
  
!3 Inspect Table prints table/view columns to be used for Insert/Update/Query  
procedure  
  
!|Inspect Table|users|  
  
!3 Inspect query prints columns and data  
  
|Insert|users|  
|name|username|  
|david haselhoff|dhoff|  
|arthur dent|adent|
```

```
!|Inspect query|select * from users|
```

Figure 4.2. Inspect makes regression tests easier to write



When the test is executed, FitNesse will append meta-data and results to the test tables in gray color (Figure 4.2). To convert the results into a new test, select the entire table in the browser, directly from the *rendered results* page (not from the HTML source or wiki source), and copy it. Internet Explorer allows you to get just a few rows at a time, while in some versions of Firefox you have to select the entire table in order to copy it properly. Edit the test page, delete the old table and paste the contents of the clipboard into the page editor. You should see the results table with column values separated by tabs. Click the *Spreadsheet to FitNesse* button below the editor text box. This turns the tab-separated results table into a FitNesse test table, converting the tabs into pipes to separate cells and even putting the exclamation mark before the first row automatically.

Store Query

Store Query reads out query results and stores them into a Fixture symbol for later use. Specify the full query as the first argument and the symbol name as the second argument (without >>). You can then use this stored result set as a parameter of the Query command later:

```
|Store Query|select n from ( select 1 as n union select 2 union select 3) x|
firsttable|
```

```
|query|<<firsttable|  
|n|  
|1|  
|2|  
|3|
```

You can also directly compare two stored queries and check for differences.

Compare Stored Queries

Compare Stored Queries compares two previously stored query results. Specify symbol names as the first and second argument (without <<). The query structure must be listed in the second row. (Use Inspect Query to build it quickly if you do not want to type it.) Column structure is specified so that some columns can be ignored during comparison (just don't list them), and for the partial row-key mapping to work. Put a question mark after the column names that do not belong to the primary key to make the comparisons better. The comparison will print out all matching rows in green (Figure 4.3), and list rows that are in just one query with red (and fail the test if such rows exist). If some rows are matched partially, just by primary key, differences in individual value cells will also be shown and the test will fail.

```
|execute|create table testtbl (n int, name varchar(100))|  
  
!|insert|testtbl|  
|n|name|  
|1|NAME1|  
|3|NAME3|  
|2|NAME2|  
  
|Store Query|select * from testtbl|fromtable|  
  
|Store Query|select n, concat('NAME',n) as name from ( select 1 as n union  
select 3 union select 2) x|fromdual|  
  
|compare stored queries|fromtable|fromdual|  
|name|n?|  
  
|execute|drop table testtbl|
```

Figure 4.3. Comparing stored queries

The screenshot shows the DbFit web interface. The main content area displays the following SQL code:

```
execute create table testtbl (n int, name varchar(100))

insert testtbl
n   name
1   NAME1
3   NAME3
2   NAME2

Store Query select * from testtbl fromtable

Store Query select n, concat('NAME',n) as name from ( select 1 as n union select 3 union
Query      select 2) x fromdual

compare stored queries fromtable fromdual
name      n?
NAME1     1
NAME3     3
NAME2     2

execute drop table testtbl
```

At the bottom of the interface, there is a footer link: [DbFit Select on SourceForge | DbFit documentation and reference | Comments and feedback](#)

Transaction control

By default, each individual test (FitNesse page) in flow mode is executed in a transaction that is automatically rolled back after the test. In standalone mode, you are responsible for overall transaction control.

If in flow mode, you can use the `Commit` and `Rollback` commands to control the transactions manually, but remember that a final rollback will be added at the end of the test. These commands have no additional arguments.

In standalone mode, you will probably control transactions from outside DbFit. Utility commands to commit and rollback are still provided, if you need them, as part of the `DatabaseEnvironment` fixture. For example, use this table to rollback:

```
!|DatabaseEnvironment|
|Rollback|
```

Best practices

Once you are ready to start creating database unit tests or integrating DbFit fixtures into your acceptance tests, you can ensure that your tests will be more easily maintained in the future by taking just a few moments to organise them now.

Initialising tests

We've already mentioned that it is a good practice to initialise the database connection on a suite's *SetUp* page. Another thing to consider is whether you need to clear values stored in symbols. FitNesse persists symbol values from one test to the next in a *Suite* run. If you are using symbols in DbFit tables to store primary keys or other values, you may need to clear the values from your symbols before each test run. The flow-mode of DbFit provides the `ClearParameters` method which is just called within its own table:

```
!|ClearParameters|
```

Calling this method on your Suite's *SetUp* page will help prevent "false fails" due to symbol values being persisted between tests in a Suite run.

Markup variables can help you keep your data straight

Say you are testing a feature in a contact management system that allows users to update a contact's details. Assume for now that only a contact's first, middle, and last name are editable. An outline for a test might be:

- Set-up a contact record in the database with values in the firstname, middlename, and lastname columns
- Update the contact's details in the application under test
- Verify that the contact record was updated correctly in the database

Assuming a developer has created an *ActionFixture* for the contact update feature, your test may look something like this:

```
!|Insert|Contacts|  
|FirstName|MiddTeName|LastName|ContactID?|
```

```
|Joan|Of|Arc|>>contactID|

!|ActionFixture|
|start|ContactManager.ContactUpdateFixture| |
|enter|ContactID|<<contactID|
|enter|FirstName|Johnny|
|enter|MiddleName|Apple|
|enter|LastName|Seed|
|press|Submit|

!|Query|SELECT FirstName, MiddleName, LastName FROM Contacts WHERE ContactID =
@contactID|
|FirstName|MiddleName|LastName|
|Johnny|Apple|Seed|
```

Creating similar tests seems pretty straightforward — you can just copy the contents of your test page into a new test page. You'll just need to remember to change the original and updated data values in your set-up, ActionFixture, and verification tables to suit the needs of each new test. This approach *seems* manageable until you have to deal with 30 fields instead of 3, or with 10 related records instead of 1. FitNesse markup variables can help prevent copy/paste mistakes and can make creating similar tests a bit faster. Using the previous example, here is what the contents of the test page would look like with markup variables, and as a rendered page in FitNesse:

```
!define originalFirstName {Joan}
!define originalMiddleName {Of}
!define originalLastName {Arc}
!define updatedFirstName {Johnny}
!define updatedMiddleName {Apple}
!define updatedLastName {Seed}

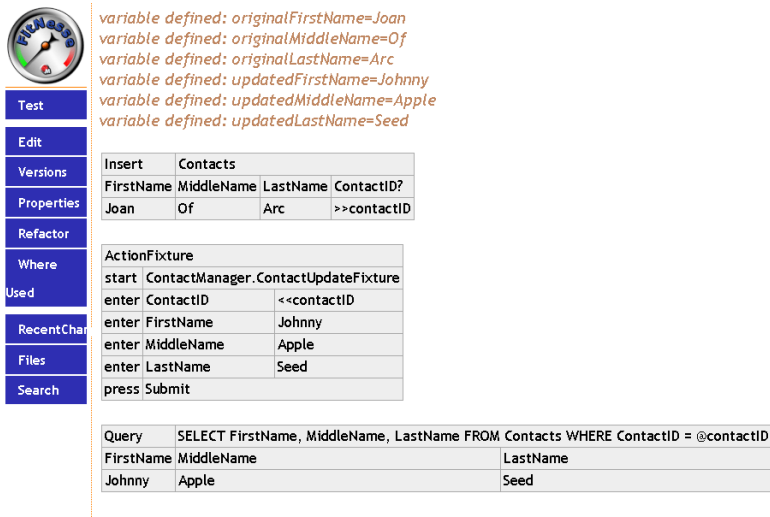
!|Insert|Contacts|
|FirstName|MiddleName|LastName|ContactID?|
|${originalFirstName}|${originalMiddleName}|${originalLastName}|>>contactID|

!|ActionFixture|
|start|ContactManager.ContactUpdateFixture| |
|enter|ContactID|<<contactID|
|enter|FirstName|${updatedFirstName}|
|enter|MiddleName|${updatedMiddleName}|
|enter|LastName|${updatedLastName}|
|press|Submit|

!|Query|SELECT FirstName, MiddleName, LastName FROM Contacts WHERE ContactID =
@contactID|
|FirstName|MiddleName|LastName|
```

```
|${updatedFirstName}|${updatedMiddleName}|${updatedLastName}|
```

Figure 5.1. A rendered Test page with markup variables



variable defined: originalFirstName=Joan
 variable defined: originalMiddleName=Of
 variable defined: originalLastName=Arc
 variable defined: updatedFirstName=Johnny
 variable defined: updatedMiddleName=Apple
 variable defined: updatedLastName=Seed

Insert	Contacts			
FirstName	MiddleName	LastName	ContactID?	
Joan	Of	Arc	>>contactID	

ActionFixture

```
start ContactManager.ContactUpdateFixture
enter ContactID <<contactID
enter FirstName Johnny
enter MiddleName Apple
enter LastName Seed
press Submit
```

Query	SELECT FirstName, MiddleName, LastName FROM Contacts WHERE ContactID = @contactID	
FirstName	MiddleName	LastName
Johnny	Apple	Seed

Now to create a new test, you'll only need to update the values for the 6 markup variables, instead of updating 9 cells. This type of test *template* will come in handy when your application under test grows in functionality and complexity.

Reusing DbFit tables

Unless you have a script that populates your test database with the data required for all tests, you will probably need to use some of your DbFit tables in more than one test for the purpose of setting up test data. (Even if you have a data set-up script, you may need to perform similar data verifications in multiple tests). Sure, you could copy your DbFit tables into multiple test pages...but what happens when a database column name is changed? Using the previous example, if you've got 100 tests that insert a record into the *Contacts* table, you've got 100 test pages to update if the *LastName* column is renamed to *Surname*.

The `!include` widget in FitNesse allows you to include all of the content from an existing page into another page. So, just as a database developer may create a stored procedure to perform a common action, we can create reusable DbFit tables that may be included on multiple test pages.

Included pages are displayed in a collapsible region, so that users can hide that part of the page. Use `!include -c PageName` to make it hidden by default (this is handy if the included part is just preparing data for the test). Use `!include -seamless PageName` to include pages directly, without displaying the border around the included region or allowing people to hide it.

First, it is a good idea to create a main wiki in which to store your reusable pages. Create a new top-level page in FitNesse and name it *UtilityPages* or something similar. Continuing with the previous example, we could create a new page in the *UtilityPages* wiki named *InsertContact*. It would contain the DbFit Insert table from your test:

```
!|Insert|Contacts|
|FirstName|MiddleName|LastName|ContactID?|
|${originalFirstName}|${originalMiddleName}|${originalLastName}|>>contactID|
```

You can do the same for the Query table, naming the reusable page *VerifyUpdatedContact* or something similar. The last step would be to update your test page to include the DbFit tables from your *UtilityPages* wiki.

```
!define originalFirstName {Joan}
!define originalMiddleName {Of}
!define originalLastName {Arc}
!define updatedFirstName {Johnny}
!define updatedMiddleName {Apple}
!define updatedLastName {Seed}

!include .UtilityPages.InsertContact


!|ActionFixture|
|start|ContactManager.ContactUpdateFixture| |
|enter|ContactID|<<contactID|
|enter|FirstName|${updatedFirstName}|
|enter|MiddleName|${updatedMiddleName}|
|enter|LastName|${updatedLastName}|
|press|Submit|

!include .UtilityPages.VerifyUpdatedContact
```

Save the test page, and it will be rendered with the contents of the *InsertContact* and *VerifyUpdatedContact* pages (Figure 5.2).

Now, if the Contacts table changes, you'd only have to update your 2 pages that reference the Contacts table directly instead of potentially dozens or hundreds of tests pages.

Figure 5.2. A rendered Test page with included DbFit tables



Test

Edit

Versions

Properties

Refactor

Where

Used

Recent Changes

Files

Search

▼ *Included page: [.UtilityPages.InsertContact](#)*

Insert	Contacts		
FirstName	MiddleName	LastName	ContactID?
Joan	Of	Arc	>>contactID

ActionFixture

start	ContactManager.ContactUpdateFixture
enter	ContactID <<contactID
enter	FirstName Johnny
enter	MiddleName Apple
enter	LastName Seed
press	Submit

▼ *Included page: [.UtilityPages.VerifyUpdatedContact](#)*

Query SELECT FirstName, MiddleName, LastName FROM Contacts WHERE ContactID = @contactID

FirstName	MiddleName	LastName
Johnny	Apple	Seed

Frequently asked questions

I'd like to use DbFit with Sybase/PostGRE. Is that possible?

Implementing support for a new database takes less than one working day, if you know your way around the database internals. Just implement a new `DbEnvironment` variant and fire away. Sybase, PostGRE and similar databases are not supported simply because I do not have a test database at hand and do not know enough about those systems to extract all relevant meta-data. If you need can provide a test database and one person who can help with database meta-data extraction, please contact me and I'll help with implementing support for your database.

NULLs and blank cells

I need to insert null values to several columns, but I get an error message if a column is empty 'Cannot use input parameters as output values. Please remove the question mark after '.

In FitNesse, empty cell generally means “print the current value, don't test”. That is why you get the message that the input parameter (insert value) cannot be used for output. Use the keyword `NULL` to insert nulls.

DbFit complains that it cannot read columns or parameters. What's wrong?

When I try to insert (or execute a procedure), DbFit complains that it “Cannot read columns/parameters for object”. What's wrong?

There are two possible causes of this problem:

The first is that you misspelled the procedure or table name (it is obvious, but people keep reporting problems caused by this, so I'd like to suggest double-checking that first). Keep in mind that DbFit is executing under the privileges of the user that you supplied in the `Connect` command, so it may need a schema prefix to see your objects.

The second possible cause is that the current user does not have access to table or procedure metadata. See section “*Does DbFit require any special database privileges?*” on page 50 for detailed information on required privileges.

Does DbFit require any special database privileges?

DbFit generally goes not require any special privileges for the database. The only important thing is that the user whose credentials you are using to run the test pages has at least read-only access to the schema meta-data. For MySQL, that means `select grants on mysql.proc` and `information_schema.columns` tables. For Oracle, that means access to `all_arguments`, `all_tab_columns` and `all_synonyms`. For SqlServer, that means access to `sys.columns` and `sys.parameters` tables.

Does DbFit support VARBINARY columns?

Yes, and it treats them as arrays of bytes. You can use the standard FitNesse syntax for byte arrays (comma-separated list of values), or you can use the `0xHEXDIGITS` syntax if you activate the byte array handler. `|CellHandler-Loader| |Load|dbfit.util.ByteArrayHandler|`

My stored procedure returns a result set. How do I use it?

In Oracle, you can store the `REF CURSOR` output parameter into a variable (using `>>varname`) and then execute a query with that variable:

```
|Query|<<varname|
```

With SQL Server, there are no typically output arguments, but a stored procedure just opens a cursor. You can use the `Query` table directly against it. If you would like to use a parameter, put `exec` before the procedure name:

```
|set parameter|hm|3|  
  
|query|exec listusers_p @hm|  
|name|username|  
|user1|user name 1|  
|user2|user name 2|  
|user3|user name 3|
```

DbFit says that my VARBINARY is System.Byte[]

You see `System.Byte[]` because that is how .NET prints a byte array. The object should have been stored correctly as a byte array, and you should be able to use `0xHEXDIGITS` syntax for comparisons. See `BinaryTests` acceptance test for examples.

Does DbFit support GUID columns?

Yes, but you may need to activate that support manually. DbFit has a non-standard extension for FIT.NET which allows it to “understand” GUID fields. That is being implemented now in the standard FIT.NET test runner, so you may not need to load it manually in the future. In any case, put this table in your test to load the GUID handler:

```
|CellHandlerLoader|  
|Load|dbfit.util.GuidHandler|
```

This table should come below the test type definition (below `SQLServerTest`).

DbFit complains about an unsupported type. What's wrong?

To handle types properly, DbFit requires a bit of additional information that does not come from typical database driver meta-data. That is why there is some small amount of work involved in supporting each column type. You can see a list of supported data types for each database server in the *AcceptanceTests* suite. If you are using a column/parameter type that is not there, then no one asked for that yet. Please contact me and I'll be happy to extend DbFit to support that type.

Can you extend DbFit to support Oracle collection types?

A short answer is “Not easily”. The .NET version uses Microsoft's Oracle .NET driver because Oracle ODP requires binary client compatibility (if DbFit is compiled for ODP 9, it will not work with Oracle 10 or 11, and vice-versa). If you really desperately need this, I can create a version-specific variant of ODP driver support for you, with support for Oracle collection types. I started implementing this in the Java version, but it turned out that proper use of Oracle collections in JDBC requires Oracle-specific extensions and meta-data which currently does not get loaded in DbFit. This would require restructuring in the way that DbFit handles types, so it is on my roadmap for some future release, but not a priority. If you need it sooner, contact me.

How can we use Windows-integrated authentication?

Instead of calling `connect` with three or four separate arguments, call it with just one argument and specify the full .NET or JDBC connection string. If you are a database developer and don't know about those things, ask a .NET or Java developer in your organisation to help you out.

DBFit complains about invalid fixtures/methods

If you want to use DbFit in flow mode, then DbFit test name should be the first table on the page – not even imports, cell handler loaders or any set-up can come before it. If you want to use some other fixture to control the flow, then use DbFit in standalone mode.

Why does DbFit not see the time portion of my Date fields?

This issue affects Oracle users on the Java version of DbFit. Oracle's JDBC driver strips the time from Date columns since version 9, so no amount of magic on the client side can fix that. see *Oracle JDBC FAQ*¹ for more information.

In the JDBC FAQ, Oracle suggests setting `-Doracle.jdbc.V8Compatible="true"` to map dates to timestamps. That should instantly solve your problem, but I don't know what else is triggered by that flag. If you want to experiment, change the batch file that starts FitNesse and add that before the FitNesse class name.

DbFit complains about registering a SQL Server driver

This issue affects SQL Server users in the Java version of DbFit, and the message displayed on the screen after the Connect command is `Cannot register SQL driver com.microsoft.sqlserver.jdbc.SQLServerDriver`. You need to download Microsoft SQL Server JDBC driver from *their site*², it is not open-source and I cannot distribute it with DbFit. Deploy the JAR in the same folder as `dbfit-XXX.jar`. If you specify the full JDBC connection string explicitly, use Microsoft's driver in JDBC URL. DbFit does not support 3rd party SQL Server drivers at the moment.

¹http://www.oracle.com/technology/tech/java/sqlj_jdbc/htdocs/jdbc_faq.htm#08_01

²<http://www.microsoft.com/downloads/details.aspx?>

FamilyId=C47053EB-3B64-4794-950D-81E1EC91C1BA&displaylang=en