Table of contents

The Fitnesse Grails plugin provides a bridge between Fitnesse and Grails.

Fitnesse Grails Plugin - Reference Documentation

Authors: Erik Pragt (jWorks.nl), Marcin Erdmann (jWorks.nl)

Version: 2.0

Table of Contents

1 Introduction

The Fitnesse plugin provides a integration point between the popular Open Source testing framework Fitne

This guide documents the usage of the plugin and provides an easy starting point to get you up to speed wi

> ⚠ Note that the **SLIM** version of Fitnesse is supported. FIT, the older Fitnesse protocol, i supported in the near future.

Release History

- June 10, 2011
  - 1.0 Release!
    - Feature: Support for GivWenZen
    - Bugfix: Fixed the tutorial documentation (thanks Pierre D. Tremblay!)
    - Bugfix: Fixed the dependency resolution of libraries (thanks Steef de Bruijn)
    - Bugfix: Fixed plugin scopes (thanks Steef de Bruijn)
- May 18, 2011
  - 0.95 Release
    - Feature: Enabled easy mapping of enums classes
    - Improvement: Improved Query Fixture DSL to allow dotted notation
    - Improvement: Enhanced reloading of application
- March 22, 2011
  - 0.9 Release
    - We upped the version number to 0.9 because the plugin is quite feature complete now, and
    - Feature: Introduced @Fixture annotation, so fixtures can now be given any name (they do annotated with the annotation.
    - Feature: Added Transaction support
    - Bugfix: Fixed JSON conversion for domain classes and collections
    - Bugfix: Fixed reloading classes and fixtures that use inheritance
    - Bugfix: Fixed grails test-app
    - Bugfix: Fixed issues in the documentation
- March 1, 2011

- 0.5 Release
  - Feature: Fitnesse testrunner created. Grails applications can now be te "FrontPage.GrailsTestSuite.SlimTestSystem?suite"
  - Feature: Fitnesse integrated. Fitnesse can now be started using: grails run-fitnesse
  - Feature: Fitnesse can be disabled by setting the grails.plugin.fitnesse.disabled property to tr
  - Improvement: Internal Fitnesse upgraded to the newest Fitnesse version
  - Improvement: Naming of configuration parameters have changed to be more consistent. *E 'plugin' (without the 's')
  - Improvement: Bundled testproject in Github
  - Improvement: Fixture classes are now Spring beans
  - Bugfix: Fixed lazyloadingexception
  - Bugfix: Fixed exception message when a given constructor is not found for a fixture
  - Bugfix: Fixed reloading of services to give errors in Fixtures
  - Bugfix: Marcin's surname is fixed in the documentation!
- November 16, 2010
  - 0.4 Release, thanks to Marcin Erdmann!
    - Complete refactoring of the plugin thanks to Marcin Erdmann (ie using Artefacts, Artefact more!)
    - **Important**: All fixtures should now be end with the suffix 'Fixture', as in 'CalculateFixtur but the plugin adds a Fixture suffix when looking for the class.
    - Improved error messages
    - You can now create complex objects from within Fitnesse by using JSON syntax.
    - More documentation!
    - Fixed some bugs in documentation (thanks Olivier Hedin for reporting!)
- October 12, 2010
  - 0.3 Release.
    - Added more documentation (configuration options, quickstart), refactored the internals
    - Added verbose logging switch
    - Fixed a Grails reloading bug, which caused ports to be opened twice. Now the plugin clos restart
    - You can now throw StopTest Exceptions from Fixture constructors (which is not possible i
- September 19, 2010
  - 0.2 release. First public release. Includes lots of documentation, including a tutorial with 3 fixtur
- September 15, 2010
  - initial 0.1 release.

1.1 Features

The Fitnesse plugin supports some additional features besides the standard Fitnesse functionality.

Support for GivWenZen (since 1.0)

Since version 1.0, we support the GivWenZen library. GivWenZen allows a user to use the BDD Given W
team get the words right and create a ubiquitous language to describe and test a business domain. See the (

To use GivWenZen in Grails Fitnesse, import the `fitnesse.grails` package and use the `GivWenZe`
the example below:

```
|import|
|fitnesse.grails|

|script|
|start|giv wen zen for grails|

|script|
|given|the number 5|
|when|incrementing it by 3|
|then|the result is 8|

|script|
|given|the number 1|
|when|incrementing it by 2|
|and|incrementing it by 3|
|then|the result is 6|
```

To make this test work, you'll need a step class like the one below:

```
import org.givwenzen.annotations.DomainStep
import org.givwenzen.annotations.DomainSteps

@DomainSteps
class GivWenZenSupportSteps {
    private int number

@DomainStep("the number (\\d+)")
    void setNumber(int number) {
        this.number = number
    }

@DomainStep("incrementing it by (\\d+)")
    void incrementNumber(int by) {
        number += by
    }

@DomainStep("the result is (\\d+)")
    boolean expect(int result) {
        number == result
    }
}
```

Automatic Enum mapping (since 0.95)

Since version 0.95, it's possible to automatically map enum values. Just specify the value of the enum in th

An example:

```
class MyEnumFixture {
    Color color
}

enum Color { RED, GREEN, BLUE }
```

By specifying the name of the enum value in the test (see below), the mapping is automatically done:

```
|my enum     |
|color|color?|
|RED  |RED   |
```

Nested property mapping for Query Fixtures (since 0.95)

It was already possible (since version 0.4) to create a simple Query Fixture using the Query Fixture DSI properties using a dotted (.) notation, as can be seen below:

```
class NestedPropertyKeyValueQueryFixture {
    static queryFixture = true

static mapping = ["name":"author.name", "birthYear":"author.birthYear", "title":"

def queryResults() {
        return [new Book(title:'Grails in Action', author: new Author(name:"Peter
    }
}
```

Fixture annotation (since 0.9)

Since version 0.9, it's possible to annotate Fixture classes with the @Fixture annotatation. This means that to be suffixed with Fixture), as long as they are annotated with the @Fixture annotation.

```
@Fixture
class BuyBook {
    // contents here
}
```

Transaction support (since 0.9)

Transaction support provides three fixtures called BeginTransaction, Commit and Rollback. To use you ne SuiteSetUp:

```
|import|
|fitnesse.grails|
```

The transaction support was introduced mainly for scenarios where test pages via fixtures modify the persisted after the end of the test page execution. It means that the test page can be run multiple times with it doesn't affect other tests by database contamination. To run a test page within a rolledback transaction tables to your test page:

```
|begin transaction|
--some test tables--
|rollback|
```

If you wish to run all of your test pages contained within a suite in rolledback transactions simply ad TearDown pages respectively for the given suite.

Templates (since 0.4)

You can now easily create Fixtures by typing:

```
grails create-fitnesse-query-fixture <name of fixture>
```

Or

```
grails create-fitnesse-fixture <name of fixture>
```

This will create a fixture in the grails-appfitnesse directory, which should give you a head start on how to c

Complex objects (since 0.4)

You can now create complex objects from within Fitnesse. This uses the JSON format, since it's easy to re

An example can be seen below:

Wiki

```
|create book inventory                          |
|book                                  |amount|
|{author: Stephen King, title: IT}     |     3|
|{author: Dean Koontz, title: Chase}   |     5|
```

This JSON code in the test is mapped to the following Fixture and Domain class:

Fixture

```
class CreateBookInventoryFixture {
    Book book

int amount

def bookService

CreateBookInventoryFixture() {
        Book.list()*.delete()
    }

void execute() {
        amount.times {
            book.id = null
            bookService.addBook(book)
        }
    }
}
```

And the very simple Domain class:

```
class Book {
    String author
    String title
}
```

Version 0.9 brings support for collections into JSON object conversion. The feature works for typed c
(because information about generics is is kept at runtime only for fields). It also works for hasMany relatio

Wiki

```
|json objects conversion with collections|
|producer|models|match?|
|{name: 'Audi', models: [{name: 'A3'}, {name: 'A4'}]}|[{name: 'A3'}, {name: 'A4'}
```

Fixture

```
class JsonObjectsConversionWithCollectionsFixture {
    CarProducer producer

List<CarModel> models

boolean match() {
        producer.models*.name.containsAll(models*.name)
    }
}
```

Domain classes

```
class CarProducer {
    String name
    static hasMany = [models: CarModel]
}

class CarModel {
    String name
}
```

Query Fixture DSL (since 0.4)

To make writing Query fixtures much easier, we've introduced the concept of a simple mapping DSL. Th
becomes almost trivial.

Consider the following Service method:

```
def checkInventory() {
    Book.executeQuery("select b.title, b.author, count(*) from Book b group by ti
}
```

If you want to use this method in a Fixture, you can do by using the mapping DSL. An example can
fixtures!

```
class CheckBookInventoryFixture {
    static queryFixture = true                              // indication that th
    static mapping   = [title: 0, author: 1, amount: 2]  // the mapping

def bookService        // injected service

def queryResults() {  // queryResults() method, which must be named like this!
        bookService.checkInventory()
    }
}
```

The mapping property determines how the query results will be mapped before being sent back to FitN
FitNesse query table for the given fixture. The values might be indexes of columns returned by the implen
from a executeQuery) or property names (i.e. if queryResults returns a collection of domain objects). The
property names of the result map one to one to table column names. An example can be found below:

```
class BookService {
     List<Book> checkInventory() {
        return Book.list()
     }
}

class CheckBookInventoryFixture {
    static queryFixture = true                                // indication that th
    static mapping = ["title", "author", "amount"]        // a different mapping

def bookService        // injected service

def queryResults() {  // queryResults() method, which must be named like this!
        bookService.checkInventory()
    }
}
```

or:

```
class CheckBookInventoryFixture {
    static queryFixture = true                                  // indication that th
    static mapping = ["title":"objectTitle", "author":"theAuthor", "amount":"amou
is the name of the test, and the value the property name of the object

def bookService        // injected service

def queryResults() {  // queryResults() method, which must be named like this!
        bookService.checkInventory()
    }
}
```

Since 0.9 you have to declare both queryFixture and mapping properties as static. It wasn't necessary bef
support for non static properties was dropped.

Strings as methods

Groovy supports methods like `"this is a method"`. The Fitnesse plugin also supports this, making s

Example

```
class MyFixture {
  boolean "check if customers exists"(int customerNumber) {
     // …
  }
}
```

Default arguments

Groovy supports default arguments. The Grails Fitnesse plugin also supports this:

```
class MyFixture {
  MyFixture(boolean clearDatabase = false) {
    // …
  }
}
```

Wiki:

```
|my fixture|true|
```

Untyped arguments

Groovy supports untyped method arguments. The Grails Fitnesse plugin also supports this:

```
class MyFixture {
  boolean checkCustomer(customerNumber) {
    // …
  }
}
```

> ⚠ Note that the Fitnesse SLIM protocol only supports Strings and Lists. A result of this, is t
> things could go wrong. An example of this is when using an integer. This integer is interpre
> converted to it's value, so int 1 becomes int 49. So, if you're unsure, use types in your fixtures.

Automatic reloading of Fixtures

The Fixtures in the `grails-app/fitnesse` directory are automatically reloaded and injected by Grail

Functions are getters

Fitnesse decision tables do **not** have getters and setters, but setters and functions. A function is the same a example:

Normal Fitnesse

```
|my decision fixture|
|digit | roman?|
|1     |      I|
|5     |      V|
```

This would result in the following Fixture:

```
class MyDecisionFixture {
    int digit
    String roman

void execute() {
        roman = RomanNumberConverter.convertDigit(digit)
    }

String roman() {
        return roman
    }
}
```

This has been improved, so that Fitnesse also looks at getters:

Improved Fitnesse

This would result in the following Fixture:

```
class MyDecisionFixture {
    int digit
    String roman

void execute() {
        roman = RomanNumberConverter.convertDigit(digit)
    }
}
```

Improved Stop Test Exceptions

You can now throw StopTest Exceptions from Fixture constructors, to halt all further test execution. This Fitnesse.

2 Getting started

You can install the plugin by typing

```
$ grails install-plugin fitnesse
```

Now, whenever you start up your Grails application (using run-app for example), a Fitnesse server will als
to the Grails application.

> ⚠ Note that currently you'll need to start up and close your Grails application yourself; the plugi
> change in the future, but for now, it's a manual step.

The internal Fitnesse server allows connections made by the Fitnesse Wiki to be interpreted by Fixtures, w

2.1 Quickstart

For the impatient, this is the quickstart to get Fitnesse to work with Grails.

Step 1: Install the Fitnesse Plugin

```
grails install-plugin fitnesse
```

Step 2: Start the Grails application

```
grails run-app
```

Step 3 (in a differtent console or terminal): Start Fitnesse

```
grails run-fitnesse
```

Step 4: Configure Fitnesse (for Linux/MacOS)

Browse to http://localhost:9090, and paste the following code (See 'Edit' in the left side menu) into the Fitr

```
!define TEST_SYSTEM {slim}
!define COMMAND_PATTERN {echo}
```

⚠ The COMMAND_PATTERN needs to be something which executes, but does noting. On empty batch file.

Step 5: Add the Test

Paste the following also into the Fitnesse wiki:

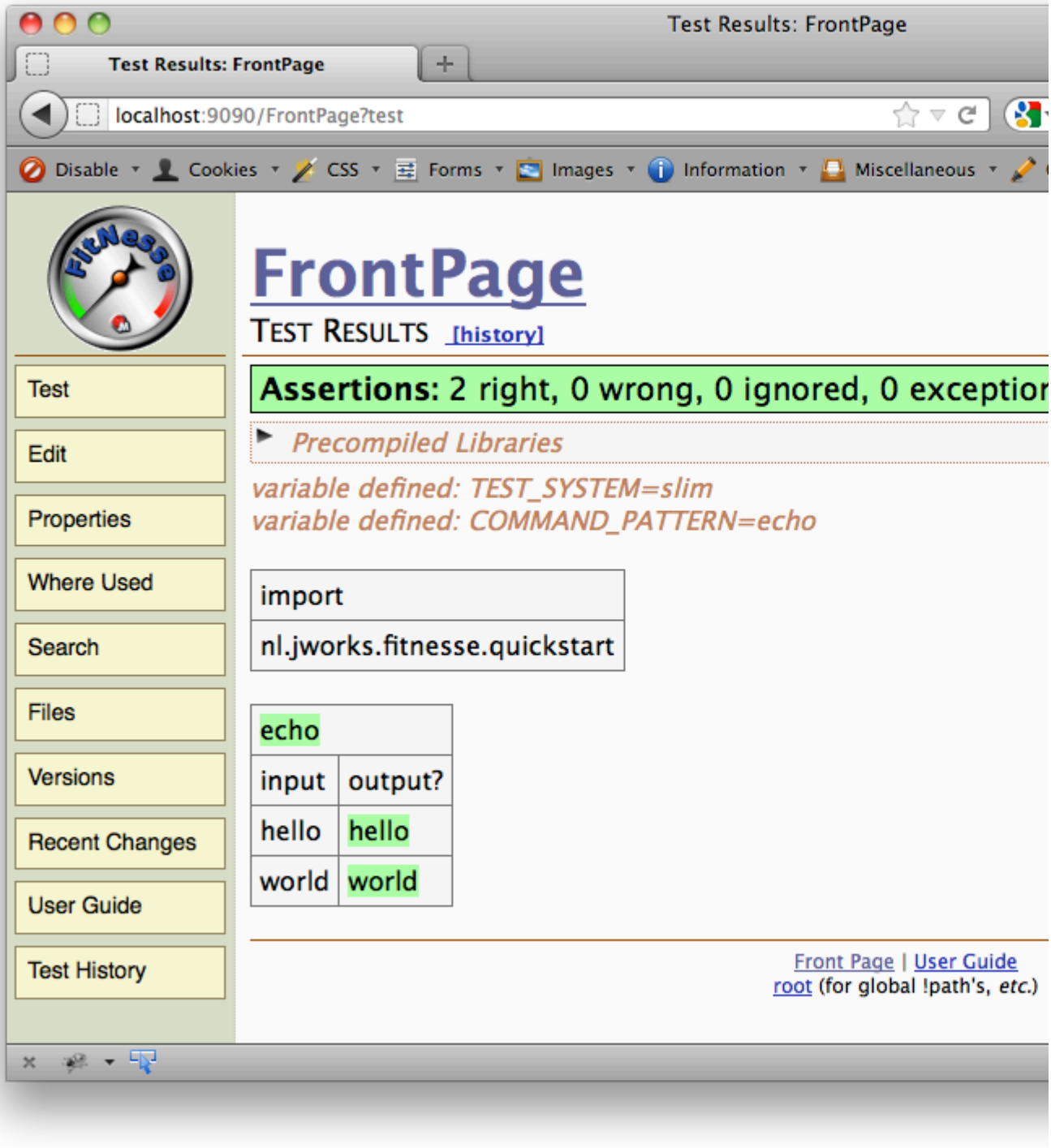
```
|import                          |
|nl.jworks.fitnesse.quickstart|

|echo|
|input|output?|
|hello|  hello|
|world|  world|
```

Save the page.

Step 6: Verify it works

Click on **Properties**, select **Test** for '`Page type`', click on **Save properties** to save the page.

Now press **Test**. Everything should be Green now!



2.2 Fixtures

Fitnesse Fixtures, not to be confused with Grails Fixtures, allow a bridge between the Fitnesse Wiki and th

Fixtures are similar to Grails artifacts and share most of their characteristics, like hot reloading and Depe 'Fitnesse' directory in the `grails-app` directory.

Example

Below you'll find an example Wiki test plus the Fixture which will serve as the bridge between the Wiki ar

Wiki:

```
|loan calculator       |
|income|debts|category?|
|10000 |500  |A        |
|5000  |10000|C        |
```

And the Fixture:

```
class LoanCalculatorFixture {
    // Input parameters
    int income
    int debts

// Output parameter
    String category

// Dependency Injected
    def loanCalculationService

void execute() {
        this.category = loanCalculationService.calculateLoanCategory(income, debt
    }
}
```

Explanation

The Wiki Tests consists of 3 parts:

- the fixture name (`loan calculator`)
- the header(income, debts, and category)
- and the body (the rest of the table). As you can see, `category` is suffixed with a question mark to i ('get'). As such, first the input values are set, then the execute method is called by Fitnesse, and finally

> ⚠ A small note: normally output parameters are 'functions' and not getters. This means you v output parameters. In the Grails version of the Fitnesse runner, this isn't necessary (fortunatel 'functions' as well as the Groovy properties. Less code to write, less code to maintain!

2.3 Wiki

The Wiki contains all the tests. The wiki consists of two parts:

- Formatted text
- Tables

Tables are the tests, and only tables are executed.

The following tests are at least supported by the Grails Fitnesse plugin:

Name	Description
Decision Table	Supplies the inputs and outputs for decisions. This is similar to the Fit Column Fixture
Query Table	Supplies the expected results of a query. This is similar to the Fit Row Fixture
Script Table	A series of actions and checks. Similar to Do Fixture.
Import	Add a path to the fixture search path.

The following tables might work, but at the moment, they haven't been tested yet. They are included here f

Name	Description
Table Table	Whatever you want it to be!
Subset Query Table	Supplies a subset of the expected results of a query.
Ordered query Table	Supplies the expected results of a query. The rows are expected to be in order. This
Comment	A table that does nothing.
Scenario Table	A table that can be called from other tables.
Library Table	A table that installs fixtures available for all test pages

For more information, please check out the Fitnesse Slim documentation on Test Tables.

2.4 Testrunner

The Fitnesse Plugin can be used when testing your application from the command line by using the bundle

```
grails test-app integration:fitnesse <name of suite or test>
```

An example of this can be found below:

```
grails test-app integration:fitnesse "FrontPage.GrailsTestSuite.SlimTestSystem?su
```

It is also possbible to run multiple tests or suites. This can done by appending the suite or tests names:

```
grails test-app integration:fitnesse "FrontPage.GrailsTestSuite.FirstSuite?suite"
"FrontPage.GrailsTestSuite.SecondSuite?suite"
```

Running these commands will start up the Grails application, launch the Fitnesse tests, and return the resul

Run sequence

The default Wiki Suite or Test which will be run using `grails test-app` will be determined in the fol

1. First, if a suite or test is specified as an argument to `grails test-app`, the argument will be used
2. Second, if a suite or test is specified in the Config.groovy, that one will be used
3. Thirdly, all tests defined on the Fitnesse Frontpage (FrontPage?suite) will be run if none of the above

3 Configuration options

The following describes the list of configuration options. As in the convention over configuration princ used to change the behavior of the plugin a litte.

Wiki options

The Wiki Server port

The Wiki contains all the test suites. It is an embedded HTTP server which by default is accessible on port

This can be done by setting the `grails.plugin.wiki.port` option. An example can be found below

```
grails {
  plugin {
    wiki {
        port = 9090
    }
  }
}
```

The Wiki Server directory

The Wiki pages are stored in a specific directory, <project-root>/'wiki' by default.

This can be changed if needed, by setting the `grails.plugin.fitnesse.wiki.dir` option. An ex

```
grails {
  plugin {
    wiki {
      dir = 'otherwiki'
    }
  }
}
```

The default Wiki Suite or Test

When supplying no arguments to `grails test-app`, all tests on the Fronpage will be run.

This can be changed if needed, by setting the `grails.plugin.fitnesse.wiki.defaultSuite` can be found below:

```
grails {
    plugin {
        fitnesse {
            wiki {
                defaultSuite = ["FrontPage.GrailsTestSuite.SlimTestSystem?suite"]
            }
        }
    }
}
```

It is also possible to supply the testsuite to run by passing it as an argument to `grails test-app`.

Note: see 'Integration testing' to see which option takes presedence over the other.

Slim Server options

Slim Server Port

The SlimServer opens ports and listens to a server socket. The default is 8085 and it cycles through convenient port number for you, you can set the SLIM_PORT variable in the Fitnesse Wiki to any por Guide.

Changing the port in the Fitnesse Wiki also requires a change in the plugin configuration to tell the plugin

To change the port in the Grails application, you can set it in the Config.groovy. Add the following:

```
grails {
  plugin {
    fitnesse {
      slim {
        port = 8085
      }
    }
  }
}
```

Logging

To add more logging to the application, add the following to the log4j DSL:

```
debug  'nl.jworks.grails.plugin.fitnesse'
```

Verbose

To enable verbose logging, which outputs more information than normally needed, set the `verbose` valu

```
grails {
  plugin {
    fitnesse {
      slim {
        verbose = true
      }
    }
  }
}
```

Disable the plugin per environment

By default, the plugin is enabled for all environments, including production. However, it is possible to sel done by setting the `grails.plugin.fitnesse.disabled` option to `true`.

An example from `Config.groovy` which disables the plugin in production:

```
environments {
  production {
    grails {
      plugin {
        fitnesse {
          disabled = true
        }
      }
    }
  }
}
```

4 Differences from other test frameworks

Fitnesse is an Acceptance Test framework, and, as such, is different from Functional or Unit test framewor

While Unit Tests focuses more on the design of the code and the functional correctness on a unit level (as and flow of pages, Fitnesse fills the gap between those areas. Fitnesse focuses on the correct behaviour of a UI, but it focus is much wider than a single unit of work.

Fitnesse allows complete scripts of business functionality to be tested, with complete integration in the with implementation details, Fitnesse Tests can be written Test First, allowing a fully integrated TDD a with most other frameworks.

For more information about Fitnesse, please check the Fitnesse website.

4.1 Comparison Matrix

This comparison matrix provides a quick overview of the possibilities of different frameworks.

Name	Supports TDD	Refactor Safe	Collaborative	Target audience
(x)Unit	Yes	No	No	Developers
Selenium	No	Yes	No	Developers/Testers
Fitnesse	Yes	Yes	Yes	Developers/Testers/Business

5 Tutorials

5.1 Integration testing

This tutorial describes the creation of a simple application and how to test it.

1. Create your Grails application.

```
$ grails create-app bookstore
$ cd bookstore
```

2. Install the plugin.

```
$ grails install-plugin fitnesse
```

We are going to create a small bookstore with a small domain to test the core functionality without using a
using Unit tests after writing the Fitnesse Acceptance Test, but that is outside the scope of this tutorial.

To start in a Test Driven approach, we'll start by creating the test first.

To do so, we'll need to start Fitnesse as well as Grails.

3. Start the Grails server.

```
$ grails run-app
```

4. Start the Fitnesse server.

Since Fitnesse is bundled with the plugin, there's no need to download anything.

The following will extract Fitnesse and start it on the default port, 9090.

```
$ grails run-fitnesse
```

5. Create the test

Open a web browser and point it to http://localhost:9090 . The Fitnesse front page will show up.

Click on **Edit**, and add the following to the text area: `^BookStoreTest` . Press **Save**.

You will now have a text with a question mark next to it. Click on the question mark to create a new page.

In the new page, remove the contents of the text area and replace it by the following:

```
!define TEST_SYSTEM {slim}
!define COMMAND_PATTERN {echo}

|import|
|bookstore|

|create book inventory|
|author|title|amount|
|Stephen King|IT|3|
|Dean Koontz|Chase|5|

|script|buy book scenario|
|customer buys|2|books with title|IT|
|customer buys|1|books with title|Chase|

|query:check book inventory|
|author|title|amount|
|Stephen King|IT|1|
|Dean Koontz|Chase|4|
```

Press save after you created the page.

> ⚠ The COMMAND_PATTERN should (currently) have a dummy value! I have chosen 'echo' s users you need to create a batch file, for example, echo.bat, without any contents. The re normally starts a Slim Server itself. However, in our case, our Grails application starts the starting the server, we use a dummy command pattern.

In this test, you've used 3 kinds of Test tables:

- a Decision Table (`create book inventory`) used to setup the data
- a Script Table (`buy book scenario`) to execute the test
- a Query table (`check book inventory`) to verify the results of the test

Click on **Test** on the left side of the page. The test will execute, and will fail with errors li `CreateBookInventory[0]`. This is because no Fixtures have been created yet. This way of workin our test clearly is red (or yellow, in this case), so we need to focus on getting the test green. To make the te

6. Creating the Fixtures

We'll start by creating the first Fixture mentioned in the error message, which is the `CreateBookInver` @create-fitnesse-fixture**. You can involve the script by typing the following:**

```
$ grails create-fitnesse-fixture CreateBookInventory
```

This will create a file called @CreateBookInventoryFixture.groovy `in the` grails-app/fitnesse/boo

Because the Fitnesse plugin will reload the Fixtures automatically, pressing the Test* button in the V correctly, and the cell is colored green by Fitnesse. If for some reason this doesn't work, please restart the G

Now, we'll have to create the author, title and amount properties in the Fixture and the corresponding Book

> ⚠ Note, Grails might crash due to the Fixture missing the domain classes. Whenever this happen

6.1 The Decision Table

The Fixture to support the Decision Table should look like this:

```
class CreateBookInventoryFixture {
    String author
    String title

int amount

def bookService

CreateBookInventoryFixture() {
        Book.list()*.delete()
    }

void execute() {
        amount.times {
            bookService.addBook(new Book(author: author, title: title))
        }
    }
}
```

The Fixture has an `execute` method. This method is called after input parameters have been set, but bef to have a place to do something with the input, and prepare the output.

The Book class like this:

```
class Book {
    String author
    String title
}
```

We also need a BookService class, since the Fixtures should not hold any logic; they are just a translatio this:

```
class BookService {

static transactional = true

def addBook(book) {
        book.save()
    }

def buyBook(title) {
        Book.findByTitle(title).delete()
    }

def checkInventory() {
        Book.executeQuery("select b.title, b.author, count(*) from Book b group b
    }
}
```

Execute the test again, and the first part of the test should be green. If not, and you're sure you've executed mean there's an error in the tutorial, or I've been unclear in the way to explain things. In any way, if you're

6.2 The Script Table

The Script Fixture to execute the Script Table should look like this:

```
class BuyBookScenarioFixture {
    def bookService

void customerBuysBooksWithTitle(int amount, title) {
        amount.times {
            bookService.buyBook(title)
        }
    }
}
```

> ⚠ Note that this implementation is very simplistic. In this example, I simply delete the bo
> assigning them to a customer or a basket.

6.3 The Query Table

Finally, to verify if our assertions match the execution of our code, we need to check the results of our acti

```
class CheckBookInventoryFixture {
    static queryFixture = true  // indication that this is a query fixture
    static mapping = [title: 0, author: 1, amount: 2]  // the mapping

def bookService       // injected service

def queryResults() {  // queryResults() method, which must be named like this!
        bookService.checkInventory()
    }
}
```

> ⚠ The mapping is used here to map the column names to the position of the values returned by
> information, please check the 'Query DSL' part in the 'Features' documentation.

Now, when running the Test again in Fitnesse, everything should be green, and you have your first test ru
check if the result is still valid by rerunning the Tests!

5.2 Refcard

I (Erik Pragt) have written a Refcard to explain the basics of Fitnesse. It provides a starting point for those
some more advanced tips for those who have worked with Fitnesse before.

It has been published by DZone, and you can read it here.

5.3 Example project

To properly test our plugin, we have creates an example project which goes through all the features the
conversions, etc.

The example project is not bundled in the plugin itself, but can be found as a separate download, which ca