Jags

Table of contents

Note:
This section is now obsolete. Instead of devising our own specification language, we have decided to extend Relax NG, which is an XML schema language with facilities making such extension reasonably straightforward.

1. Introduction

Jags is a shorthand for writing JAppGen specifications. The name may itself be short for **JAGScript**.

Jags may or may not be used by application developers. At least in the short term, it is a convenience. Rather than generating code immediately, it is helpful to have various tools output a specification in an easily parsed form which can be used to create input for other software modules. Specifically, we want to be able to describe an application's data model and then use that description to generate the J2EE front end, the middle tier, and the SQL used to create the database. An intermediate language such as Jags simplifies development and makes testing much easier.

2. Application Structure

In this early form of JAppGen, we assume that J2EE applications fall into three basic parts:

- a front end, the presentation layer;
- a middle bit, which constitutes an object model for the application proper; and
- a persistence layer, the bridge to the underlying database.

In the software, these will be represented by the **web/**, **biz/**, and **db/** directories respectively. We assume that application software is split into three Java packages and that these names (web, biz, db) appear in the qualified package names. So if for example the application package was `org.foo` then middle-tier Java classes would be in or below the package `org.foo.biz`.

We assume that the front end implements the Model-View-Controller paradigm.

3. Line Endings, Comments

Anything following a sharp sign ('#') in a jags script is a comment and is ignored, as are blank lines.

A **jags** specification consists of a number of statements, each of which occupies a single line, with two exceptions: continued lines and multi-statement lines.

A statement can be continued over more than one line by ending the line with a backslash ('\'). On the other hand, several statements can be combined into one line by separating them with

semicolons. Therefore

```
    A B \
      C   # oh say can you see ...
    D E F
```

has exactly the same meaning as

```
    A B C; D E F
```

4. Data Language

Data is specified in a mini-language which includes elements from that used to specify printf formats. The basic pattern is

```
    class-name (S field-spec)+
```

where `class-name` represents a valid Java class name, which we shall take to mean something matching the pattern

```
    [A-Z][a-zA-Z0-9_]*
```

that is, a name beginning with a capital letter followed by zero or more alphanumeric characters, including the underscore ('_').

`S` is shorthand for a delimiter, by which we mean anything matching

```
    [\b\n\f\r]+
```

meaning one or more of either a space or a newline or a formfeed or a carriage return.

4.1. Field Specs

A `field-spec` looks like

```
    field-name ('%' (width)? type-spec)? (quantifier)? ('=>' (class-name
'.' field-name)? ) )?
```

where a `field-name` matches

```
    [a-z][a-zA-Z0-9_]*
```

If `width` is present, it matches

```
 ('-')? number ( '.' number )?
```

If the minus sign is present, this means that the field should be left-justified on output. The first number represents the maximum number of characters to be output or stored. If the second number is present, the field should be numeric and the number represents the number of decimal points.

If the `type-spec` is present, it takes one of two forms. In the first the type is represented by a single character which in turn represents a Java primitive type or a String:

b	boolean

c	char
d	double
f	float
i	integer
l	long
s	String

In the second case the type is represented by the name of a Java class. This may be qualified, but **provisionally** certain very commonly used Java classes are recognized without being qualified:

ArrayList	java.util.ArrayList
Date	java.util.Date
List	java.util.List
String	java.lang.String

A `quantifier` is one of

```
    '?' | '*' | '+' | '!'
```

where as is conventional '?' means 'zero or one of', '*' means 'zero or more of' and '+' means 'one or more of'. Less conventionally, '!' signifies that one and only one value must be present, and this value must be unique.

If the => is present, it signifies that in the database the field is used in an index. If the optional **classname.fieldname** is omitted, then the index is on the class/table being specified. If **classname.fieldname** is present, then the field is in SQL terms a FOREIGN KEY.

5. Examples

5.1. Field Specs

Given what has been said so far it should be easy to understand that the following field specs are identical:

```
    title%-20s
    title%-20String
```

and that both represent a field called `title` which is up to 20 characters in length, is left-justified on output, and may not be omitted, because there is no quantifier. In SQL terms, it could be described as

```
   title VARCHAR(20) NOT NULL,
```

If used to describe a member of a class, the field specs correspond to

```
    // INSTANCE VARIABLE
    private String _title;
    // ACCESS METHODS
    public String getTitle() {
        return _title;
    }
    public void setTitle(String s) {
        if (s == null)
            throw new IllegalArgumentException("null title");
        _title = s;
    }
```

While the test for a null title makes sense, in a J2EE application the test would normally be handled by validating user input.

5.2. Index Examples

By convention, if the first field in a table is called `id`, is a long, and has no quantifier, it is a primary key. Therefore the Jags statement

```
    Product id%l name%-20s!=> components%Component*
```

can generate the SQL DDL statement

```
    CREATE TABLE Product (
        id    INT         PRIMARY KEY,
        name  VARCHAR(20) NOT NULL KEY);
```

`name` is an index into product and because of the '!' quantifier is unique and so is an alternate key.