



Getting Started with SPARQL Rules (SPIN)

Version 1.2

Contents

- Getting Started with SPARQL Rules (SPIN).....3**
 - 1.1 Conventions.....4
 - 1.2 The data.....4
 - 1.3 Calling built-in functions.....5
 - 1.4 Defining your own functions.....6
 - 1.5 Inferencing.....8
 - 1.6 Adding a constructor..... 12
 - 1.7 Adding constraints.....12
 - 1.8 Defining and calling a template.....14
 - 1.9 Summing up..... 16

Getting Started with SPARQL Rules (SPIN)

SPARQL Rules are a collection of RDF vocabularies such as the SPARQL Inferencing Notation (SPIN) that build on the W3C SPARQL standard to let you define new functions, stored procedures, constraint checking, and inferencing rules for your Semantic Web models, all stored using object-oriented conventions and the RDF and SPARQL standards. TopBraid Composer makes it easy to define SPARQL Rules business logic around your data models by clicking, dragging, and filling out dialog boxes, and TopQuadrant's TopSPIN engine implements these business rules so that your applications can use and control their data with many more possibilities than a simple SPARQL engine can offer.

Using SPARQL Rules, your applications can:

- **Calculate the value of a property based on other properties**—for example, the area of a geometric figure as a product of its height and width, the age of a person as a difference between today's date and person's birthday, or a display name as a concatenation of the first and last names.
- **Check constraints and perform data validation**—for example, your application can raise inconsistency flags when currently available information does not fit the specified integrity constraints.
- **Isolate a set of rules to be executed under certain conditions**—for example, to support incremental reasoning, to initialize certain values when a resource is first created, or to drive interactive applications.

The SPARQL Rules framework has three layers:

1. **An RDF vocabulary for SPARQL** A SPARQL Rules implementation stores SPARQL queries as RDF triples, like any other data or metadata in a Semantic Web model. TopBraid Composer lets you view and edit your SELECT, CONSTRUCT, and other SPARQL queries using the same syntax defined in the W3C's [SPARQL Query Language for RDF](#) Recommendation, but storing them as triples makes it possible to link the queries to specific classes, to add metadata about them, and to re-use them in different places. (You can convert between the W3C syntax and the triples version at the [SPIN RDF Converter](#) web site, but when using TopBraid Composer, these conversions are all automated for you, so there is no need to ever view the triples version of your queries.)
2. **SPIN Vocabulary** This vocabulary includes terms such as `spin:constraint` and `spin:constructor`, which let you define business rules and attach them to classes.
3. **Module Library** The library holds frequently needed modeling patterns, with functions and templates to constrain cardinalities and value ranges. SPARQL Rules let you define and store your own re-usable query templates; it also provides several predefined ones based on modeling patterns that are common in real-world applications.

In this tutorial, you'll begin with a small set of RDF data and do the following with it:

- Call built-in extension functions and write, save, and call your own new functions.
- Define inferencing rules that automatically create ISO 8601 "yyyy-mm-dd" formatted dates from date strings formatted as "mm/dd/yy" and attach these rules to the Purchase class, where the MaterialsPurchase and ServiceContract classes can inherit them.
- Define query templates to be called from different rule definitions, making code more modular and reducing redundancy.
- Automate the addition of a postingDate value to each purchase as it is added to the system.
- Define constraints that alert the user to unpaid materials purchase invoices that are more than 90 days old and service contract invoices that are more than 60 days old.

You can use this tutorial with any version of TopBraid Composer, including the [Free Edition](#). (It does demonstrate one extra feature of the Standard and Maestro editions, but this feature is not necessary for the completion of the tutorial.) It assumes basic familiarity with the use of TopBraid Composer.

1.1 Conventions

Class, property and individual names are written in a sans serif font like this.

Names for user interface widgets and menu options are presented in a style **like this**.

Where exercises require information to be typed into TBC a monospaced font is used **like this**.

Exercises and required tutorial steps are presented like this:

1. Execute the first step.
2. Then, execute the second step.
3. Here is the third step to carry out.



Tips and suggestions for using TBC and building ontologies are presented like this.



Potential pitfalls and warnings are presented like this.



General notes are presented like this.

1.2 The data

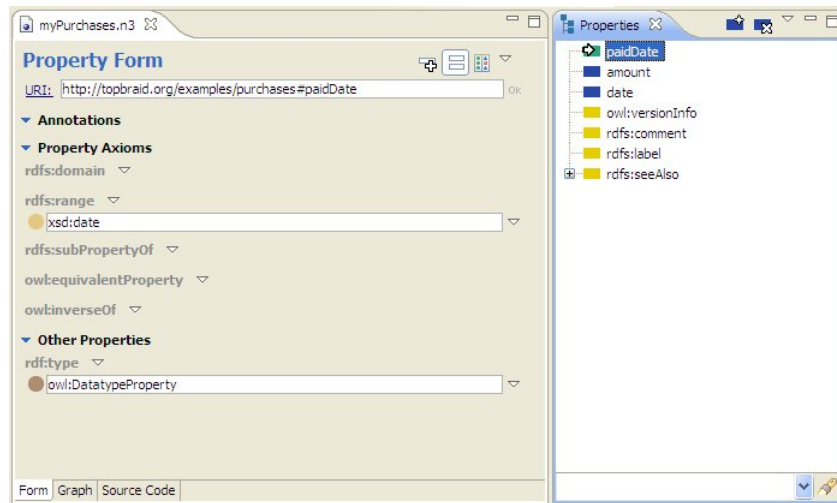
Let's start by creating a file that imports the data we'll be working with and taking a look at it.

1. From the **File** menu, pick **New** and **RDF/OWL** file. Give it a base URI of `http://topbraid.org/myPurchases` and filename of `myPurchases`.
2. On your new file's **Imports** view, click the "Import from URL" icon (with the plus sign in front of a globe) and enter `http://topbraid.org/examples/purchases` as the resource to import. Save your file.
3. In the **Classes** view, expand the **owl:Thing** and then **Purchase** nodes, and you'll see that Purchase has two subclasses named `MaterialsPurchase` and `ServiceContract`.
4. Click the yellowish-brown circle next to either one, and in the **Instances** view you'll see two instances for each.

All the instances have a similar structure and different data, except that the `purchase102` `ServiceContract` includes an additional **paidDate** property. If you click on **paidDate** in the **Properties** view, you'll see on the Property Form that it has an **rdfs:range** of `"xsd:date"`, while the **date** property has no **rdfs:range** assigned, so that its content will be treated as simple text strings.



If the Property Form is not showing in the main editor window, select the "Form" tab at the bottom of that window.



We're going to write a new function that calls a built-in function to convert **date** values to the `xsd:date` ISO 8601 yyyy-mm-dd format, but first let's take a look at some of the built-in functions that are available and how we can try them out.

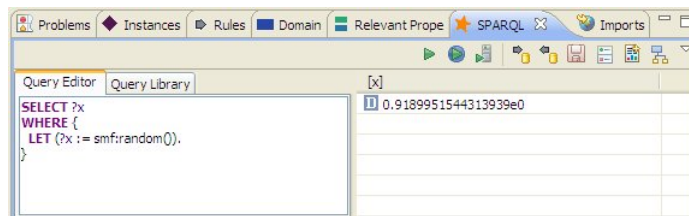
1.3 Calling built-in functions

The "TopBraid SPARQLMotion Functions Library" page of TopBraid Composer's online help lists over 70 functions available for you to call from your applications, including string functions, mathematical functions, logical functions, and ontology functions. Because TopBraid Composer uses the Jena SPARQL engine, it supports the LET keyword for assignment of variables, which gives you a way to experiment with built-in variables using a short, simple SPARQL query.

For example, to try the mathematical function **smf:random()** (a TopBraid Composer extension to the Jena function library), enter the following query in the **SPARQL** view's Query Editor tab:

```
SELECT ?x
WHERE {
  LET (?x := smf:random()).
}
```

To execute it, either click the "Execute SPARQL" green triangle button or press Ctrl+Enter, and you'll see a random number between 0 and 1 appear in the [x] column of the SPARQL query result. Because this function returns a different value each time you call it, go ahead and execute it a few more times.



Another built-in extension function is **smf:parseDate()**, which converts a string in a semi-structured date format into an `xsd:date`, `xsd:dateTime`, or `xsd:time` value. (To learn more about **smf:random()**, **smf:parseDate()**, and other available functions, select **Help Contents** from the **Help** menu and then **TopBraid Composer -> Reference ->**

SPARQLMotion Functions Reference.) This function takes two parameters: a date string and a template showing which pieces of the date are where in that string. Try pasting the following query into the **SPARQL** view and executing it:

```
SELECT ?x
WHERE {
  LET (?x := smf:parseDate("12/3/09", "MM/dd/YY")).
}
```

The value displayed under [x] is "2009-12-03".



The TopBraid Composer online help for this function includes the URL of a web page with greater detail about options for the format string in the **smf:parseDate()** function's second parameter.

We're going to use this function to define a simpler function that converts the Purchase date strings for us.

1.4 Defining your own functions

Before we use the SPARQL Rules vocabulary, we need to import it into our vocabulary. Open your **Imports** view and drag the **spin.rdf** file from the TopBraid/SPIN folder in the **Navigator** view into the **Import** view. You'll see several new nodes appear in the **Properties** view, each with an **sp:** or **spin:** prefix and a plus sign next to it that expands the node to show the full range of SPARQL Rules properties. For this tutorial, you won't need to use these properties from this view, because TopBraid Composer gives you forms to fill out to specify the functions, rules, and constraints that you'll create.

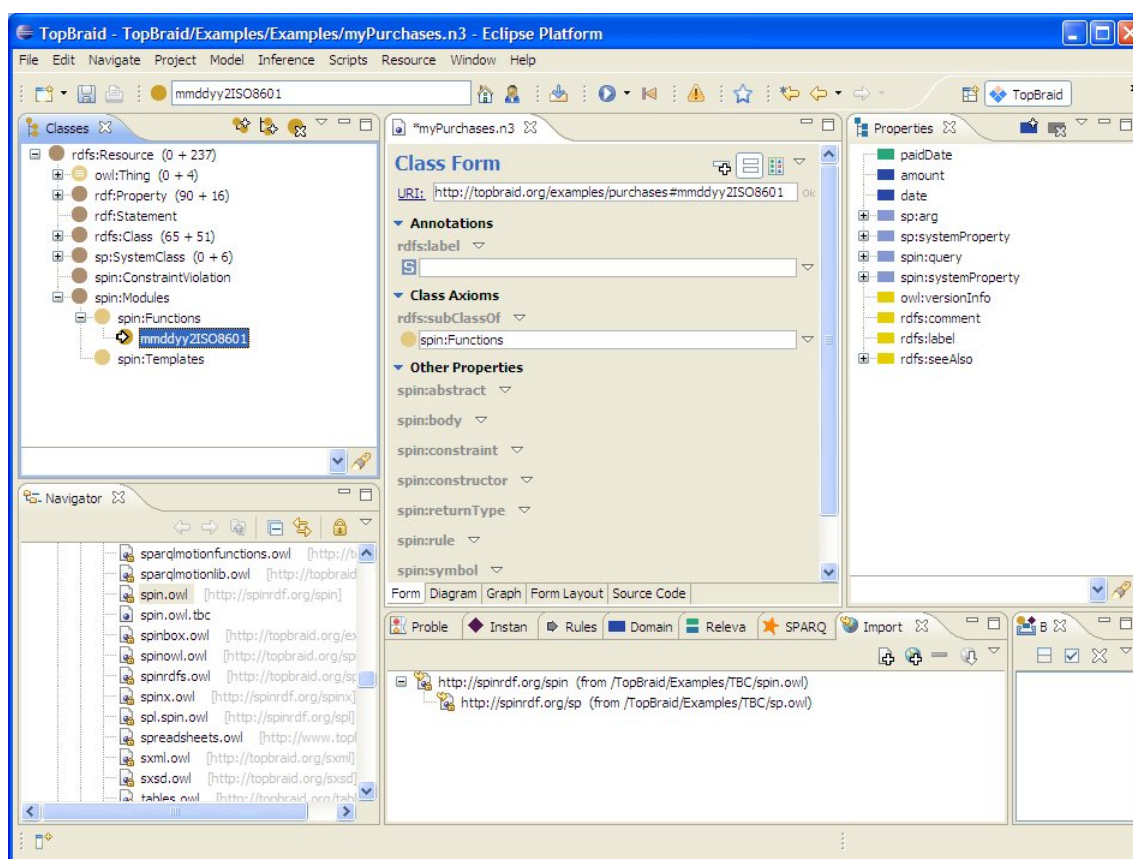


When creating a brand new model that will use the SPARQL Rules vocabulary, instead of picking **RDF/OWL File** from the **File -> New** menu, pick **RDF/OWL/SPIN File** to create an empty new file with the SPARQL Rules vocabulary already imported.

The **Classes** view also shows a few new classes. Click the plus sign next to **spin:Modules** to expand it, and you'll see that one of the subclasses is **spin:Functions**. (We'll use the other subclass, **spin:Templates**, later on in this tutorial.)

To create your new function,

1. Right-click on **spin:Functions** and pick **Create subclass** from the context menu.
2. On the **Create Classes** dialog box, click on the default new function name of "Functions_1", replace it with the name "mmdyy2ISO8601", and press Enter.
3. Click the **OK** button. You'll see that your new subclass of **spin:Functions** is currently selected in the **Classes** view, and the class form will have the **URI** and **rdfs:subClassOf** values already filled out.



Your new function will have an argument passed to it: the date string to convert. The body of the function must reference this argument, so let's define the argument before creating the function body. Like everything else in SPARQL Rules, this definition will ultimately be stored with a series of triples, but because argument definition is so common in application development, TopBraid Composer includes a template to make it easier.

1. SPARQL Rules model arguments to functions as constraints on those functions. On the `mmddyy2ISO8601` class form, click the **spin:constraint** widget (the small white triangle to the right of the name "spin:constraint") and pick "Create from SPIN template".
2. Select **sp:Argument** from the **Available Ask/Construct Templates** list, and then fill in the following two fields in the **Arguments** panel on the right of the **Create from SPIN template...** dialog box:
 - In the **comment** field, enter **Convert mm/dd/yy formatted date to xsd:date type and format**.



Remember to press the Enter key after entering a value on one of these forms. If you tab to another field and the **comment** field has a heavier border around it and the **Create from SPIN template** dialog box's **OK** button is grayed out so that you can't click it, go back to the **comment** field and press the Enter key.

- Instead of typing a value directly into the **predicate** field, click the plus sign to the right of it. This displays a **Select Resource...** dialog box; because we're defining our first (and only) argument to pass to the new `mmddyy2ISO8601` function, click **sp:arg1** on the right panel and then the **OK** button.



As an alternative to picking **Create from SPIN template** and clicking the plus sign to point to the **sp:arg1** property, you could also have simply dragged **sp:arg1** from the **Properties** view onto the **spin:constraint** property name on the form to display the same dialog box with the **predicate** already filled out for you. (You'll still want to fill out the **comment** field, though.)

3. Click the **OK** button to finish with the **Create from SPIN template** dialog box, and you'll see that the **spin:constraint** field of the class form for your new **mmddy2ISO8601** function has been filled out.
4. All that remains is to define the function body. Click the **spin:body** widget on the class form and select **Add empty row**.
5. Enter the following as the body:

```
SELECT ?x
WHERE {
  LET (?x := smf:parseDate(?arg1, "MM/dd/yy")) .
}
```



The body of this function uses **?x** as the variable name, but you can use any name you want. If you have more than one variable, and it binds to more than one value, the function will return the first value of the first variable, so there's no point in naming more than one variable after the **SELECT** keyword.



As before, don't forget to press **Enter** while the cursor is still in the **spin:body** field to tell the form that you're finished entering this value. You can also click the **Ok** button to the right of the field.

Note how, in this SPARQL query, the first argument to the **smf:parseDate()** function is **arg1**, which we defined on the **spin:constraint** part of the form earlier.

Your new functions should be all ready. Test it by entering and executing the following in the **SPARQL** view:

```
SELECT ?x
WHERE {
  LET (?x := :mmddy2ISO8601("9/6/09")) .
}
```



When you make calls to your own functions, don't forget the colon at the beginning of the function call and, if your function was not defined in the default namespace, the namespace prefix before that. All function calls are in a particular namespace (for example, **parseDate()** is in the namespace represented by the **smf** prefix), and your new **mmddy2ISO8601()** function is in the default namespace for this project. For production application development, it's better to identify the function's namespace with a specific prefix, because it will be easier to import the code into another file without confusion.

mmddy2ISO8601() is now a function that you can call anywhere, including from the body of new functions that you create. Test it further in the **SPARQL** view by passing other dates as a parameter.

This was a very simple function. There's a lot more you can do when defining your own functions, especially when you use the SPINx extensions, which let you write functions in JavaScript and call them from subclasses of **spin:Functions** like the **mmddy2ISO8601()** function that you just created.


1.5 Inferencing

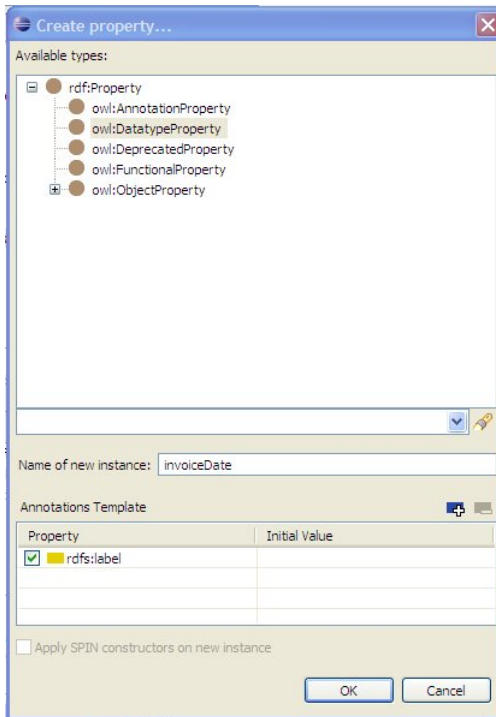
In a Semantic Web application, inferencing is the ability to create new information (that is, to generate new triples) based on information in existing triples. The incremental inferencing feature of the Standard and Maestro editions of TopBraid Composer makes it possible to derive new triples as data is edited, based on rules about what inferences to draw.

For our application, we want TopSPIN to check for invoices that are more than 90 days old, but we first must convert the dates stored with each purchase into a format that's easier for a computer to work with. We've already written a

function to do the conversion, but how do we tell the system to apply our `mmddy2ISO8601()` function to the **date** property of each purchase? With inferencing. We will tell TopSPIN that whenever it sees a triple with a predicate of `purchases:date`, it should create a new triple with the same subject as the first triple and a predicate of `:invoiceDate`. The new triple's object will be the result of passing the original date value to the `mmddy2ISO8601()` function.

First we need to declare our new **invoiceDate** property:

1. Click the **Create property** icon on the **Properties** view .
2. In the **Available types** section of the **Create property...** dialog box, select **owl:DatatypeProperty**, because our new property will not have an object resource as its value.
3. In the **Name of new instance** field, enter **invoiceDate**.



4. Click the **OK** button.
5. The **invoiceDate** values are not going to be just arbitrary strings, but dates of type `xsd:date`. Click the **rdfs:range** widget and select "Set to `xsd:date`".

Rules are attached to classes. We want our new inference rule to work for both the `MaterialsPurchase` and the `ServiceContract` classes, so we'll attach it to their parent class: `Purchase`.



The SPARQL Rules storage of all of these rules, constraints, and other kinds of stored procedures as triples under the covers are what makes it possible to connect them to specific classes and to therefore take an object-oriented approach to how they are organized and how they affect the data that you want them to affect.)

1. In the **Classes** view, click the yellowish-brown circle next to **purchases:Purchase** to display its class form. (If you don't see **purchases:Purchase**, expand the **owl:Thing** node.)
2. We're going to add a SPARQL CONSTRUCT statement in the **spin:rule** field to create our new triple, but first we'll test our CONSTRUCT statement as a regular SPARQL query. Enter the following in the **SPARQL** view and execute it:

```

CONSTRUCT {?s :invoiceDate ?idate }
WHERE {
  ?s purchases:date ?date .
  LET (?idate := :mmddyy2ISO8601(?date)) .
}

```

You should see the new triples displayed on the right panel of the **SPARQL** view.

- Once you're sure that this CONSTRUCT statement creates the triples that you want, click the **spin:rule** widget on the Purchase class's class form and select **Add empty row**. Paste the CONSTRUCT statement above over the default text in that field, which you'll notice has a similar structure to what you're pasting over it.
- You're not quite finished defining the rule. Replace the **?s** variable on the first and third lines of the CONSTRUCT statement with **?this**, a special variable that refers to the current instance of this class that is being evaluated by the inference rule.

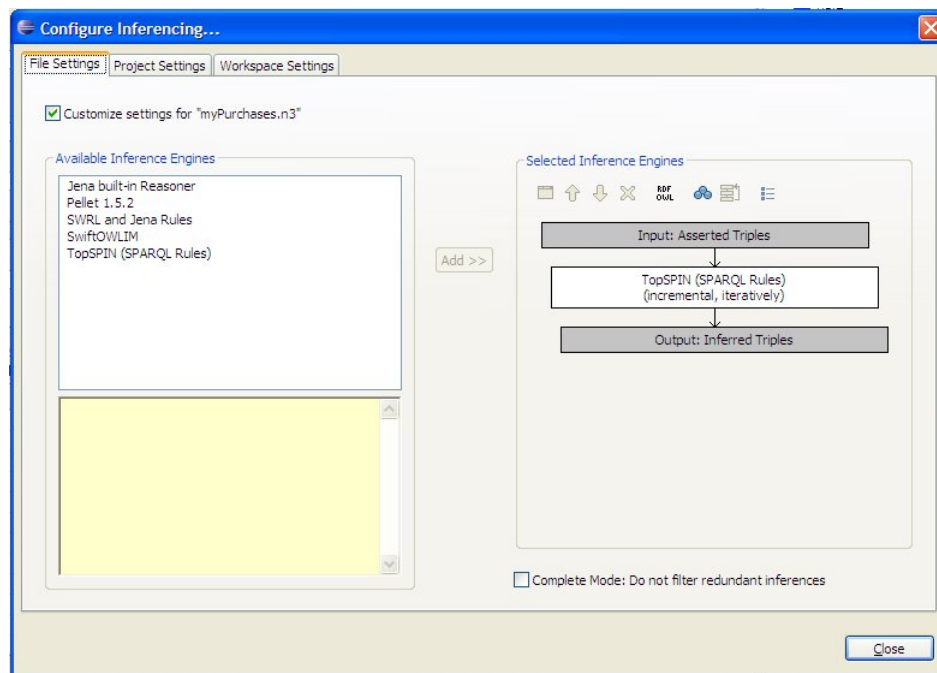


The **?this** variable is similar to the variable used in the for-each loop of many programming languages as they iterate through a collection performing the same steps on each member of the collection.

As you add each **?this** variable, TopBraid Composer bolds them in the editor, because it knows that they're special variables.

- After fixing the two instances of this variable, press Enter or click **Ok** to the right of the **spin:rule** field and you're finished entering the inference rule.

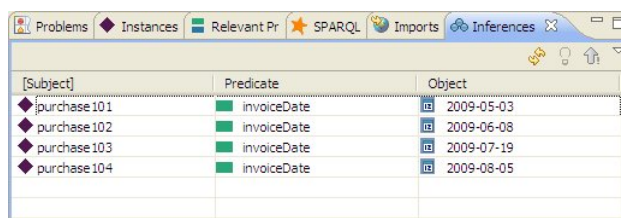
If you're using the Free edition of TopBraid Composer, you're all ready to run your inferencing. If you're using the Standard or Maestro editions, you may not be ready just yet, because you have several options for configuring what inference engine or engines do this work and how they go about it. To double-check that your copy of the Standard or Maestro edition is ready to run the inferencing that you've defined, select **Configure Inferencing...** from the **Inferences** menu and click the **File Settings** tab. It should be set up to use TopSPIN as the only inferencing engine and to do incremental inferencing, with the **Customize settings for "myPurchases"** checkbox checked, as shown here:



If not,

1. Click the **Customize settings for "myPurchases"** checkbox.
2. Delete any extra inferencing engines from the workflow on the right by selecting them and clicking the **X** ("Delete") icon above them. If TopSPIN (SPARQL Rules) is not included, select it from the **Available Inference Engines** panel on the right and click the **Add >>** button in the middle.
3. When you have TopSPIN in the workflow on the right, if it says "non-incremental", double-click it and check the **Incremental inferencing at edit time** checkbox on the **Configure TopSPIN** dialog box.
4. Click the **OK** button and then the **Close** button on the **Configure Inferencing...** dialog box.

Now, regardless of which edition you're using, you're ready to tell TopSPIN to go ahead and do the inferencing you asked for. Select **Run Inferences** from the **Inference** menu button. TopBraid Composer will display new triplets in the **Inferences** view.



[Subject]	Predicate	Object
purchase101	invoiceDate	2009-05-03
purchase102	invoiceDate	2009-06-08
purchase103	invoiceDate	2009-07-19
purchase104	invoiceDate	2009-08-05

When you ran a test CONSTRUCT statement earlier, TopBraid Composer displayed the constructed triples in the **SPARQL** view's results window. This time, though, TopBraid Composer added the inferred triples to the data you're working with. To see this:

1. Select **purchases:MaterialsPurchase** or **purchases:ServiceContract** from under the **purchases:Purchase** node in the **Classes** view.
2. Activate the **Instances** view and click the purple diamond next to any of the instances there to display its data on the Resource Form. You'll see that the new **invoiceDate** property has been added to it with a value calculated according to your inference rule.



To recalculate inferred values using the Free edition of TopBraid Composer, select **Reset Inferences** from the **Inference** menu to remove inferred values and then **Run Inferences** again.



The **mmddy2ISO8601()** function that you created and the CONSTRUCT statement that called it were actually quite simple. With the other built-in functions available to you, the ability to define and call your own functions, and the ability to create much more complex CONSTRUCT statements, TopSPIN inferencing can do far more with your data.



You attached your inferencing rule to the **Purchase** class, and because **purchases:MaterialsPurchase** and **purchases:ServiceContract** are subclasses of that, TopSPIN knew to perform this inferencing on individuals in those classes—in other words, they inherited this rule. In a larger, more complex system, this ability to define different inferencing rules for different classes (and, if necessary during the design process, to move them from one class to a more appropriate one) lets you organize your rules and their relationships to your data in a structured, well-organized way.


1.6 Adding a constructor

We'd also like to record the date and time that each set of purchase data was added to our database, and we can automate this so that no data entry is required.

1. First, we need a **postingDate** property of type `xsd:date` to store this data. Instead of creating it from scratch, right-click **paidDate** or **invoiceDate** in the **Properties** view and select **Create clone**. Right-click the new property, pick **Rename**, and call it **postingDate**.
2. On the Purchase class form, select **Add empty row** from the **spin:constructor** widget and paste the following in, replacing the default skeleton for a CONSTRUCT statement in the **spin:constructor** field:

```
CONSTRUCT {
  ?this :postingDate ?date .
}
WHERE {
  LET (?date := afn:now()) .
}
```

It's pretty simple, calling the **afn:now** extension function get the current date and time and then using that value as the object of a triple about the posting date. Like the CONSTRUCT statement that we used to infer the **invoiceDate** value, this also uses the special variable name **?this** to refer to the Purchase instance being evaluated.

3. Press Return or click **OK** and your simple constructor is all set.
4. To test it, select **MaterialsPurchase** or **ServiceContract** in the **Classes** view by clicking on the yellowish-brown icon next to either one and then go to that class's **Instances** view. (You defined your constraint for the Purchases class, but these subclasses will inherit it.)
5. Click the **Instance** view's **Create instance** icon  on the Instances view, assign the name **purchase105** to your new purchase, and click the **OK** button. The **postingDate** value will automatically appear on your new purchase's Resource Form, and you can then drag additional properties on the form and fill them out.

1.7 Adding constraints

A SPARQL Rules constraint defines a condition that, if true, should be brought to the user's attention. The SPARQL ASK statement, which returns a boolean true or false value, is ideal for this.

We're going to put a constraint on materials purchases so that TopSPIN alerts us to unpaid materials purchases that are more than 90 days old.

1. Display the class form for the **MaterialsPurchase** class in the **Classes** view. Click the **spin:constraint** widget and select **Add empty row**.
2. The default value has a skeleton for a SPARQL ASK statement; replace it with the following:

```
# an invoice with no paidDate is > 90 days old
ASK WHERE {
  ?this :invoiceDate ?invoiceDate .
  OPTIONAL {
    ?this purchases:paidDate ?paidDate .
  } .
  FILTER ((!bound(?paidDate)) && (smf:duration("d", ?invoiceDate, afn:now()) >
90)) .
}
```

The first line is a SPARQL comment explaining what the constraint is checking for.



Remember to always add a comment at the beginning of a SPARQL query used to define a constraint. It will be used to alert the user to the problem when TopSPIN finds data that has violated the constraint.


Finish entering the query by pressing Return or clicking **Ok**.



Instead of an ASK query, a SPARQL rule can use a CONSTRUCT query to create triples about the violated constraint. These triples can be used in applications built using the TopBraid platform. See spinrdf.org for more information.

In our model, a paid invoice will have a **paidDate** value and an unpaid one will not. If it doesn't (that is, if the **?paidDate** variable is not bound) and the **?invoiceDate** is more than 90 days ago, we want the query above to return a boolean value of "true". Like the **random()** and **parseDate()** functions we saw earlier, the **duration()** function is from the SPARQLMotion namespace, which includes many other useful extension functions.

To test whether you've set up this constraint correctly, you want to have at least one purchase that is more than 90 days old and one that isn't. To do this,

1. Make sure that MaterialsPurchase is selected in the **Classes** view and open the **Instances** view.
2. Create another new MaterialsPurchase instance, and call this one **purchase106**. Drag a **purchases:date** property from the **Properties** view onto the purchase106 form and enter date from about a month ago in mm/dd/yy format. When you press Enter, you'll see **invoiceDate** get automatically updated with the new value.
3. Change the date you just entered to one from four months ago and press Enter.
4. Click the **Display constraint violation warnings** icon at the top of the screen  and you'll see smaller versions of the same icon appear next to the headings for the form's **invoiceDate** and **paidDate** properties. You'll see another version of the icon appear near the top of the form with a small white "1" on a red background to indicate that one constraint violation has been found for this instance. (Clicking this repeatedly toggles it on and off.)

myPurchases.n3

Resource Form

UR: <http://topbraid.org/myPurchases#purchases106> OK

Annotations

Other Properties

purchases:amount

purchases:date 9/28/10

purchases:paidDate ⚠

invoiceDate ⚠ 2010-09-28

postingDate 2011-02-28T10:31:34.78-05:00

rdf:type purchases:MaterialsPurchase

Incoming References

Form | Browser | Graph | Source Code

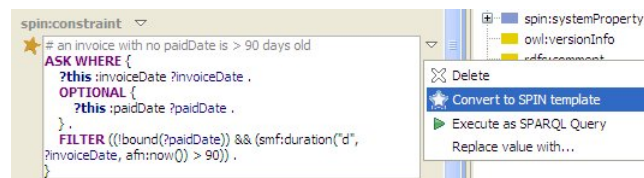
5. For a summary of all the problems that TopBraid Composer can find in the current file, display the **Problems** view (**Window->Show View->Problems** and click the **Refresh all problems of current TopBraid file** icon. 🔄)
6. To correct the error, select **Add empty row** from the **paidDate** widget, enter today's date in yyyy-mm-dd format and press Enter. Even though the invoice date is more than 90 days ago, the invoice is marked as paid, so it's no longer a problem that needs flagging. At the top of the form, you'll see that the white "1" on a red square is gone, but a grayed-out version of the warning icon ⚠️ remains to show you that TopBraid Composer has been set to check for constraint violations; it just hasn't found any.

1.8 Defining and calling a template

Let's implement a new business rule: we want to be alerted of unpaid service contract invoices that are more than 60 days old. Using the steps above, but substituting a "60" for a "90", this would all be quite straightforward, but as the size of a system scales up, it's more maintainable if common code is shared and re-used instead of copied and pasted from one place to another.

When using SPARQL Rules, you can create re-usable queries known as templates. These are flexible enough that you can pass parameters to them to customize their behavior, so we're going to create a **CheckInvoiceAge** template that checks the age of an invoice and alerts the user if the invoice is older than the value passed as a parameter: 90 days for materials purchases and 60 days for service contracts.

1. If you expand the **spin:Modules** node in the **Classes** view, you'll see that it has a subclass called **spin:Templates**. You could create a new template by right-clicking that and selecting "Create subclass", but you can save some time and trouble by doing the following:
 - a. Select the **purchases:MaterialsPurchase** node in the **Classes** view.
 - b. On its class form, click the triangle widget to the right of the **spin:constraint** that you've already created (not the similar widget above it) and select "Convert to SPIN template".



This replaces the SPARQL query in the **spin:constraint** field with a call to the new template. The yellow background shows that it's a call to another template, and the text shown in that field comes from the comment at the beginning of the original constraint query.

2. In the **Classes** view, you'll see a new subclass of the **spin:Templates** class's **spin:AskTemplates** class with a name obviously based on the comment in the original query: **aninvoicewithnopaidDateis90daysold**. Click its yellowish-brown icon in the **Classes** view to display its details on the class form. On this form, change its name from **aninvoicewithnopaidDateis90daysold** to **checkInvoiceAge**.
3. To make this template more flexible, we need to replace the "90" with a reference to an argument that gets passed and then pass that argument from calls to the template in the definitions for the **MaterialsPurchase** and **ServiceContract** classes.

The argument to the template is defined as a constraint on the template, so click the **spin:constraint** widget on the **checkInvoiceAge** class form.

4. As we did with the argument we defined to pass to the `mmddyy2ISO8601` function, we'll use a built-in template to define the argument, so pick "Create from SPIN template" and make sure that **sp:Argument** is selected in the **Available Ask/Construct Templates** list.
5. To fill out this form, click the plus sign to the right of the **predicate** field and select **sp:arg1** on the right of the **Select Resource** dialog box and click the **OK** button.



You can create your own argument names as well, which can improve readability.

6. To finish configuring the argument, on the **Create from SPIN template** dialog box:
 - Replace the current content of **rdfs:comment** field with **maximum age of invoice in days** and press Enter.
 - Click the plus sign next to the **valueType** field. On the **Select Resource** dialog box, click the **Datatypes** tab and select `xsd:int` as the data type for the argument to be passed. (If you forget this step, the value will be passed as a string, so when the template rule compares it with the numbers 60 or 90 we won't get the result we want.)
 - Click **OK** to return to the class form for the `checkInvoiceAge` template.
7. The **spin:body** value of this template is already filled out for you, because you created it from an existing constraint, but the **FILTER** condition is still hard coded to compare the number of days since the **invoiceDate** with the number 90. Replace "90" with `?arg1`, press Enter, and TopBraid Composer will bold this argument reference in the SPARQL code.

```
spin:body
# an invoice with no paidDate is > 90 days old
ASK WHERE {
  ?this :invoiceDate ?invoiceDate .
  OPTIONAL {
    ?this :paidDate ?paidDate .
  } .
  FILTER ((!bound(?paidDate)) && (smf:duration("d", ?invoiceDate, afn:now()) > ?arg1)) .
}
```

8. Just as an **rdfs:label** property provides a more readable way to refer to a URI, the template rule's **spin:labelTemplate** property further down on the class form provides a more readable way to refer to the template. It also provides a way to pass an argument value into the template: replace 90 in the **spin:labelTemplate** value with `{?arg1}` and press Enter.
9. We're done defining the template, and now it's time to add calls to it in the constraint fields of the class forms for the `MaterialsPurchase` and `ServiceContract` classes. Go to the `MaterialsPurchase` form by clicking its yellowish-brown icon in the **Classes** view and you'll see that its form already has a call to to the template—it was created when you told TopBraid Composer to create a template from the original constraint query that was there.
10. Instead of editing it, it will be simpler to just replace it. Click the widget to the right of the yellow field and pick **Delete**, and then pick **Create from SPIN template** from the **spin:constraint** widget.
11. This time, instead of picking a built-in template, you're going to pick the one that you just created: `checkInvoiceAge`.
12. When you click it to select it, the **Arguments** part of the **Create from SPIN template** dialog box asks for the **arg1** value. Enter 90, press Enter, and click **OK**. You'll see that the **spin:constraint** property has a value of the `checkInvoiceAge`'s **spin:labelTemplate** value with "90" substituted for the "{?arg1}" part that you originally entered there.



13. Next, add a constraint to the ServiceContract class form that alerts the user to unpaid invoices that are more than 60 days old. Instead of defining a constraint, you'll create it from your new **checkInvoiceAge** template. The steps will be almost identical to your addition of the similar constraint to the MaterialsPurchase class form.

Now, test your constraints. Select `purchases106` from the **purchases:MaterialsPurchase** class's **Instances** view, click the white triangle next to the `purchases:paidDate` property, and delete it. Change to `purchases:date` value to recent and older dates and watch the **invoiceDate** values adjust and the constraint violation warnings appear and disappear.



If you wanted to add the constraint that unpaid service contracts should be flagged when they were 60 days old, you could use the same template, just passing it the different value as the `arg1` value.

1.9 Summing up

In this tutorial, you defined inferencing rules, constructors, and constraints around a set of data about a series of purchases.

Remember, the SPARQL Rules property that you use to store your SPARQL query affects when that query is run, and on which data:

- **spin:rule** When running inferences, apply the query in this property to each member of the class the rule is defined on and all subclass instances.
- **spin:constraint** Define a constraint violation to be applied to all instances of a class, invoked by turning on "Display constraint violation warnings" (the 🚩 icon).
- **spin:constructor** Invoked on a new instance of a class when it's created.

If you click the **spin:constructor** and **spin:rule** widgets on any class form, you'll see that they also offer "Create from SPIN template" as a choice. Templates can be called from many different places, and like the customized functions that you can define, these are a valuable building block for creating applications.

You defined all of these using SPARQL Inferencing Notation, an RDF vocabulary that adds a layer of new features on top of SPARQL, but instead of writing out SPARQL Rules using RDF syntax (which is certainly possible) you used TopBraid Composer as a front end to define the rules with a GUI interface that made it much easier to navigate the model, data, and rules.

To actually see the effect of the rules—newly inferred triples, constraint violations, and more—you used TopQuadrant's TopSPIN inferencing engine. As with the specification of the rules, you used Composer as a front end to run TopSPIN and see its results, but you can take advantage of these same rules after creating them by using them in applications developed with TopBraid Ensemble and SPARQLMotion and then deploy them for use by multiple users with TopBraid Live.

These application development tools help you to build on your data and business rules to create customized user interfaces for use by end users who may know nothing about RDF, triples, or SPARQL, but need to use the data in the applications that you're designing for them. Before you even begin to develop and deploy user interfaces for these applications, though, SPARQL Rules (SPIN) gives you extra capabilities in modeling your data and business rules that can let the same set of data bring more value to multiple different applications.